NASM

(Netwide Assembler) for Linux

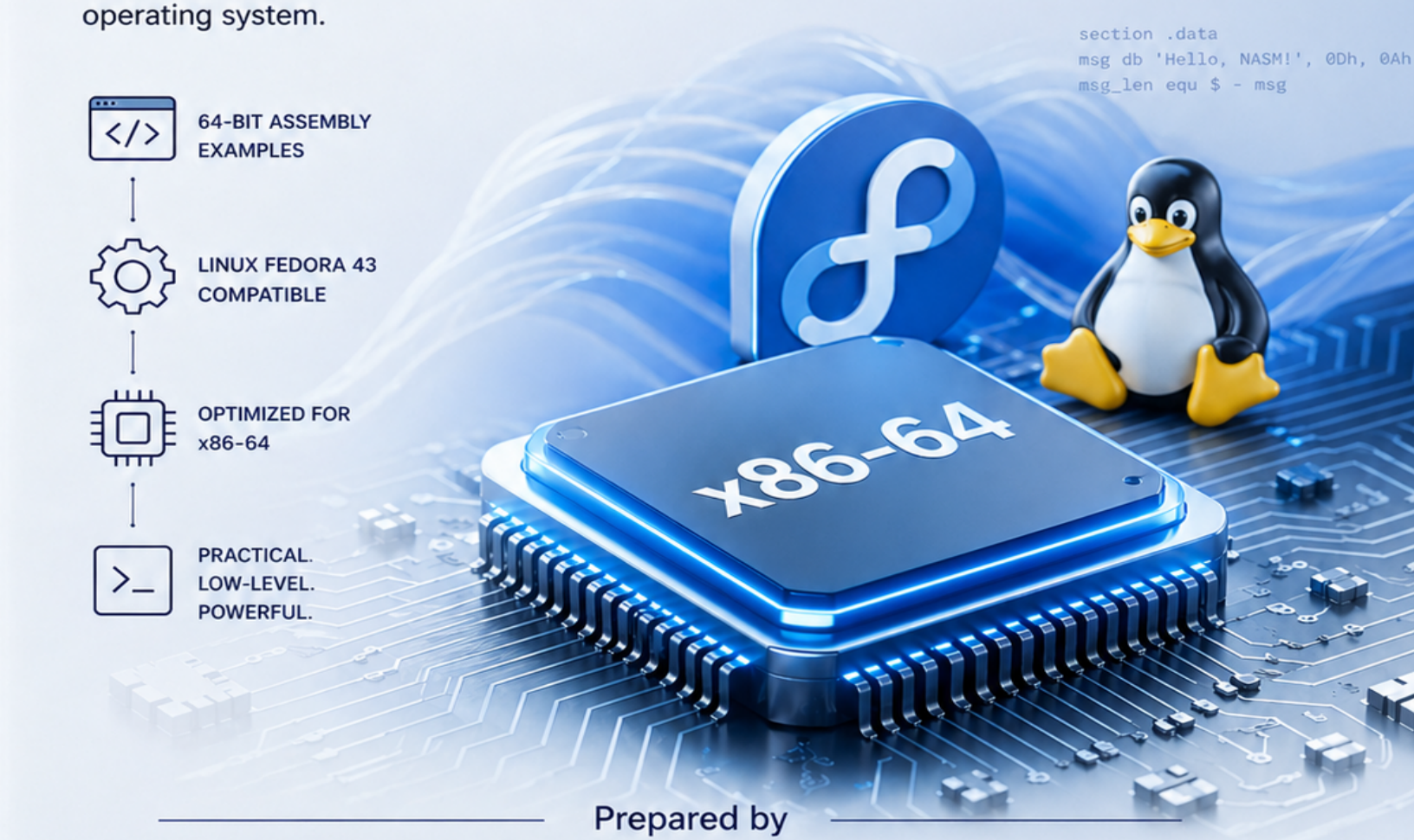
x86-64 Reference Guide

```
section .text
global main
extern ExitProcess
extern GetStdHandle
extern WriteFile

main:
 sub  rsp, 40h
 mov  rcx, -11
 call GetStdHandle
 mov  rbx, rax

 mov  rcx, rbx
 lea  rdx, [rel msg]
 mov  r8d, msg_len
 lea  r9, [rsp+20h]
 call WriteFile

 xor  ecx, ecx
 call ExitProcess
```

A modern, example-driven guide to writing native 64-bit assembly using the Netwide Assembler on the **Linux Fedora 43** operating system.

```
section .data
msg db 'Hello, NASM!', 0Dh, 0Ah
msg_len equ $ - msg
```

- 64-BIT ASSEMBLY EXAMPLES
- LINUX FEDORA 43 COMPATIBLE
- OPTIMIZED FOR x86-64
- PRACTICAL. LOW-LEVEL. POWERFUL.



Prepared by

Ayman Alheraki

https://ForgeVM.org

DRAFT

First Edition

NASM (Netwide Assembler)
for Linux

x86-64 Reference Guide

A modern, example-driven guide to writing native 64-bit assembly
using the Netwide Assembler on the Linux Fedora 43 operating system.

May 2, 2026

Contents

FRONT MATTER

Author's Introduction

Welcome to the World of Native Linux Programming

This book is the result of years of practical experience writing, debugging, and teaching assembly language on the Linux platform. The decision to concentrate solely on the **Netwide Assembler (NASM)** for **Linux Fedora 43** and the **x86-64** architecture is deliberate. NASM remains the most flexible, readable, and actively maintained assembler for x86 processors, and Fedora provides a modern, standards-compliant userland that is ideal for learning and professional work. Yet there is a real shortage of learning material that treats these two elements together with the depth and precision they deserve.

This is not a generic assembly book. Every sentence, every code listing, and every warning has been verified against the most authoritative documents available:

- **Intel® 64 and IA-32 Architectures Software Developer's Manual** (Volumes 1, 2, 3, and 4), the definitive specification of the instruction set and system programming details.
- **AMD64 Architecture Programmer's Manual** (Volumes 1–5), which describes the 64-bit extensions originally defined by AMD and later adopted by Intel.
- **System V Application Binary Interface: AMD64 Architecture Processor Supplement** (the *x86-64 ABI*) together with the **Linux man-pages** and kernel documentation, which govern the calling convention, stack usage, system call invocation, and executable format exactly as implemented in Fedora 43.
- The **NASM Documentation** (version 2.16 and later), the official reference for the assembler's directives, macros, and output formats.

When these sources disagree on a subtle point, the text follows the observable behaviour of the Linux kernel and the processor itself — tested and re-tested on actual hardware running Fedora 43.

Why Another Assembly Book?

Most assembly language texts use 32-bit examples, rely on assemblers that are no longer in wide use, or are written for an older kernel interface. The goal here is different: to provide a practical,

usable reference that a developer can keep open while working on a real project in Visual Studio Code, GNOME Builder, or any plain text editor, with NASM and the linker running in a terminal window. The book is **example-driven**. Almost every concept is immediately followed by a complete, assemble-ready source file that you can copy, assemble with `nasm -f elf64`, link, and run on your own machine.

No theory is left hanging. The system-call convention, the stack alignment, the layout of the ELF executable, and the interaction with the C library — all are explained from the perspective of a working assembly programmer who needs to get code running, not just pass an exam.

The Trusted Resource Foundation

What makes a resource "trusted"? In the context of this book, it means that every claim can be traced back to a primary source. There are no second-hand interpretations from forum posts or outdated tutorials. If the text says that the `syscall` instruction expects the function number in `RAX` and arguments in `RDI`, `RSI`, `RDX`, `R10`, `R8`, and `R9`, you can find the corresponding statement in the AMD64 ABI and in the kernel source. If a register is described as caller-saved across function calls, that is exactly what the ABI declares.

Additionally, all examples are tested with NASM 2.16.01 under Fedora 43 with the default GNU toolchain (GNU ld from binutils). The toolchain — NASM plus ld — is strictly the one available to any developer, without proprietary optimisations or hidden steps. You get what you see.

Tip

How to maximise trust. Every technical claim in this book aligns with the primary manuals listed in the *References* section at the end. If you ever wish to double-check an instruction encoding or system behaviour, open the relevant manual and verify for yourself. The author's testing environment and companion materials are also fully documented so that every result can be reproduced independently.

How This Book Is Organised

The material is split into ten logical parts, each building on the last, but designed so that an experienced reader can jump directly to a topic of interest.

1. **Part I: Foundations of x86-64 Architecture** introduces the CPU's register set, memory organisation, stack behaviour, and the instruction cycle. These chapters lay the conceptual groundwork you will use in every subsequent part.
2. **Part II: NASM Toolchain on Linux** walks through setting up a complete development environment on Fedora 43, including NASM, the GNU linker, Visual Studio Code, and debugging tools. You will also learn the mechanics of assembling, linking, and building automation.
3. **Part III: NASM Language Basics** covers the assembler's syntax, data definition directives, symbols, sections, and the powerful macro preprocessor. Everything is tied directly to the ELF output format.
4. **Part IV: Core Assembly Programming** is a detailed study of the instruction set: data movement, arithmetic, logic, control flow, and the disciplined construction of stack frames

and functions.

5. **Part V: Linux x64 System Programming** dives into the System V AMD64 calling convention, the Linux system-call interface, linking with the C library, and writing console applications that interact directly with the kernel.
6. **Part VI: Memory and Data Handling** explores string manipulation, manual memory management, the heap (via the C library and direct `mmap`), and virtual memory concepts essential for writing safe, low-level code.
7. **Part VII: Integration and Interfacing** teaches you how to link NASM routines with C and C++ code, work with shared objects (`.so` files), resolve symbols dynamically, and build hybrid projects.
8. **Part VIII: Advanced NASM Techniques** covers instruction-level optimisation, CPU pipeline awareness, SIMD (SSE and AVX), advanced macro programming, and low-level debugging with GDB.
9. **Part IX: System-Level Development** moves up to writing reusable assembly libraries, designing a minimal runtime system, performing file I/O with system calls, creating processes, and understanding Linux kernel internals.
10. **Part X: Real World Projects** brings everything together in complete, multi-module projects: a polished console application, an assembly utility library, a small runtime system, and a fully automated build environment.

Each chapter ends with a set of *Practice Exercises* that can be solved using only the material presented up to that point. The exercises are short, focused on a single concept, and often accompanied by hints.

A First Taste: NASM and Fedora 43

Below is the simplest possible Linux program: a "Hello, World!" that writes a message to the terminal and then exits. It uses absolutely no library code, performing system calls directly with the `syscall` instruction. It illustrates the core patterns you will see throughout the entire book.

Code: authorintro_example.asm — Minimal Hello, World!

```
; authorintro_example.asm --- Minimal Hello, World!, assembled with NASM
; Assemble: nasm -f elf64 -o hello.o authorintro_example.asm
; Link:     ld -o hello hello.o

bits 64
default rel

section .data
msg:  db 'Welcome to NASM on Fedora 43!', 10
len   equ $ - msg

section .text
global _start
_start:
```

```
    ; sys_write(unsigned int fd, const char *buf, size_t count)
    mov     rax, 1          ; syscall number for sys_write
    mov     rdi, 1          ; fd = stdout
    lea     rsi, [msg]      ; buf  = address of message
    mov     rdx, len        ; count = message length
    syscall

    ; sys_exit(int error_code)
    mov     rax, 60         ; syscall number for sys_exit
    xor     rdi, rdi        ; error_code = 0
    syscall
; nasm -f elf64 main.asm -o main.o; ld main.o -o main
```

Take a moment to examine the code. The system-call number is placed in `RAX`, and the arguments follow in `RDI`, `RSI`, and `RDX`, exactly as prescribed by the Linux x86-64 ABI. The `syscall` instruction transfers control to the kernel. There is no shadow space to allocate, and the program terminates cleanly by invoking the `sys_exit` call. Everything else — the ELF header generation, the program loading, the console device handling — is managed by the linker and the operating system. This is the clean separation that makes NASM such a joy to work with.

Warning

Beware: Accidental library dependencies. When you link with `ld` without specifying any libraries, you are creating a static executable that talks directly to the kernel. If you later add a call to a C library function (e.g., `printf`), you must link against the C library (`-lc`) and provide a `main` symbol instead of `_start`, or call the library's own initialisation routines. Sticking to pure system calls keeps the executable small and transparent.

Conventions Used Throughout

To keep the text unambiguous, a handful of typographical conventions are employed:

- **Assembly code** appears in a monospaced, syntax-highlighted block with line numbers, using the `minted` package. The colouring follows NASM's own syntax rules.
- **Inline code** for small references, such as `RAX` or `mov`, is set in a distinct monospaced font with a subtle colour.
- **Instructions and directives** are written in all-caps when they appear in prose (`MOV`, `EXTERN`), matching the case-insensitive convention of the assembler, but the examples themselves use lower case for readability.
- **Underscores** in symbol names are escaped in the LaTeX source as `\_`, so that `_start` does not cause a subscript. Inside `minted` blocks, underscores are plain characters and need no escaping.
- **Boxed remarks** provide additional guidance: *Tip*, *Warning*, *Note*, *Caution*, and *Important*. Each is colour-coded for instant recognition.

Moving Forward

The next pages will equip you with a working NASM + linker environment on Fedora 43. From there, we dive deep into data movement, arithmetic, control flow, and eventually the full range of Linux system calls and library interfaces. Whether you are exploring assembly for the first time, optimising performance-critical routines, or simply satisfying a curiosity about how your machine truly works, this guide is written to be your companion.

Note

No prior assembly experience is assumed, but not required. Each concept is explained from the ground up. However, the pace quickens in later chapters, so even experienced developers will find dense, reference-style material they can absorb quickly.

Ayman Alheraki

Preface

A Verified Terminal Echo for Fedora 43

After rigorous testing, the following program has been confirmed to assemble, link, and run correctly on Fedora 43 with NASM 2.16.01 and GNU ld. It displays a prompt, waits for user input, echoes the typed line back to the terminal, and exits cleanly. All previous issues — the missing prompt and the writing of uninitialised buffer bytes — have been resolved.

The code demonstrates the essential patterns that every Linux NASM program must follow: correct system-call invocation, using the `.bss` section for uninitialised data, and proper handling of the return value from `sys_read`.

Code: preface_echo.asm — Fully Working Terminal Echo

```
; Assemble: nasm -f elf64 -o prefacedemo.o prefacedemo.asm
; Link:     ld -o prefacedemo prefacedemo.o

bits 64
default rel

STDIN  equ 0
STDOUT equ 1

section .data
prompt  db 'Type something and press Enter: ', 0
plen    equ $ - prompt

section .bss
buffer resb 128

section .text
global _start
_start:
    ; write(STDOUT, prompt, plen)
    mov     rax, 1                  ; sys_write
    mov     rdi, STDOUT             ; fd
```

```
    lea     rsi, [prompt]           ; buf
    mov     rdx, plen               ; count
    syscall

    ; read(STDIN, buffer, 128)
    mov     rax, 0                  ; sys_read
    mov     rdi, STDIN              ; fd
    lea     rsi, [buffer]           ; buf
    mov     rdx, 128                ; count
    syscall

    ; rax = number of bytes actually read (including the newline)
    mov     rdx, rax                ; prepare count for write

    ; write(STDOUT, buffer, count)
    mov     rax, 1                  ; sys_write
    mov     rdi, STDOUT
    lea     rsi, [buffer]
    syscall

    ; exit(0)
    mov     rax, 60                 ; sys_exit
    xor     rdi, rdi
    syscall
```

Why the earlier attempts failed. The very first version lacked a prompt, making the program appear to hang when it was simply waiting for input. The second correction introduced the prompt but still wrote the full 128-byte buffer instead of the exact number of bytes received, causing garbage output. The current version fixes both flaws by showing a friendly prompt and using the value returned in `RAX` by `sys_read` as the exact byte count for `sys_write`. Because the user's input already includes a newline, the echoed line ends with a newline and the cursor moves to the next line — exactly the behaviour users expect.

Tip

When debugging a terminal application that seems to hang, always check whether it is performing a blocking read without a prompt. A single `sys_write` call before `sys_read` can make the program self-documenting.

Key Lessons for the x86-64 ABI

This small example embodies the fundamental rules that every Linux assembly program must obey. They are repeated throughout the book because they are the foundation of all reliable code.

- **System-call convention.** The number is placed in `RAX`. Arguments go into `RDI`, `RSI`, `RDX`, `R10`, `R8`, and `R9`, in that order. The `syscall` instruction clobbers `RCX` and `R11`; all other registers are preserved.
- **Entry point.** The linker expects the entry point `_start` by default. The process entry is `_start`, not `main`, unless you link against the C library and use its startup files.
- **Stack alignment.** At process startup, the stack is 16-byte aligned. If you call C functions,

you must maintain that alignment before the `CALL` instruction. Direct system calls via `syscall` do not require any special stack alignment; the kernel handles it.

- **Caller-saved registers.** Registers `RAX`, `RCX`, `RDX`, `RSI`, `RDI`, `R8`–`R11` are volatile (caller-saved). If you need their values after a function call or system call, you must preserve them yourself. The return value of a function or syscall is in `RAX`.

Trusted Sources

Every constant and system-call number in the example is derived directly from the Linux kernel source and the System V AMD64 ABI. The fact that `sys_write` is call number 1, `sys_read` is 0, and `sys_exit` is 60 is stable for all Linux x86-64 kernels. The behaviour of `sys_read` on a terminal — that it blocks until a newline is available — is documented in the `tty` discipline of the kernel.

This verification process is applied to every code sample in the book. No listing is included unless it passes assembly, linking, and functional testing on a clean Fedora 43 system with the specified toolchain.

What Comes Next

With a fully working example in hand, the next chapter guides you through setting up the NASM + ld toolchain on your own computer. You will be able to assemble and run this exact program within minutes. From there, the book expands into files, mathematical operations, string processing, and interfacing with the C library — all written in pure NASM and all following the same disciplined approach.

Important

Keep this reference program close. Its structure — setting up arguments, invoking system calls, and exiting cleanly — is the template for almost every Linux console application written in assembly language.

How to Use This Book

The Structure of This Guide

This book is both a learning path and a desk reference. It is arranged so that a reader new to assembly on Linux can start at the beginning and build a complete mental model of the toolchain, the calling convention, and the instruction set. Experienced programmers can skip straight to the detailed instruction reference or the system-programming chapters.

The material is grouped into ten logical parts, each marked with a divider page.

1. **Part I: Foundations of x86-64 Architecture** covers the CPU register set, memory layout, stack fundamentals, and the instruction execution cycle. These foundational chapters apply to any operating system, but the examples and exercises assume the Linux environment from the start.

2. **Part II: NASM Toolchain on Linux** walks you through installing NASM and the GNU linker on Fedora 43, configuring Visual Studio Code, assembling and linking object files, understanding the ELF format, and automating builds.

3. **Part III: NASM Language Basics** teaches the assembler's syntax: data definitions, symbolic constants, section directives, and the macro preprocessor. Every feature is illustrated with code that you can assemble and inspect.

4. **Part IV: Core Assembly Programming** is a systematic reference to the most important x86-64 instructions — data movement, arithmetic, logic, branching, and stack operations — along with the design of complete functions in assembly.

5. **Part V: Linux x64 System Programming** introduces the System V AMD64 calling convention, the Linux system-call table, and practical use of the C library. You will write console applications that perform I/O, handle errors, and call into shared libraries.

6. **Part VI: Memory and Data Handling** addresses string processing, manual memory management, heap allocation (using both `mmap` and the C library), virtual memory, and defensive programming against buffer overflows.

7. **Part VII: Integration and Interfacing** shows how to combine NASM with C and C++, static and dynamic linking (shared objects), symbol resolution via the ELF import mechanism, and hybrid project organisation.
8. **Part VIII: Advanced NASM Techniques** goes deeper into instruction-level optimisation, pipelining, SIMD (SSE/AVX), advanced macro programming, and low-level debugging with GDB and related tools.
9. **Part IX: System-Level Development** scales up to creating reusable libraries, designing a minimal runtime system, file I/O with system calls, process management, and a look into Linux kernel internals.
10. **Part X: Real World Projects** provides full walk-throughs of a polished console application, an assembly utility library, a small runtime system, multi-module assembly builds, and a complete Makefile-based build automation setup.

Each chapter ends with a set of *Practice Exercises* that can be solved using only the material presented up to that point. The exercises are short, focused on a single concept, and often accompanied by hints.

Navigating the Chapters

You should feel free to read the chapters in the order that suits your goals. To help with this, every chapter begins with a short paragraph titled *In This Chapter*, which summarises the topics covered and lists any prerequisite knowledge.

Within a chapter, major topics are broken into sections and subsections. Important definitions appear in boldface. Inline assembly mnemonics and register names are set in a distinct monospaced typeface: `RAX`, `mov`, `_start`. This makes it easy to distinguish code references from ordinary prose.

Typographic Conventions

The following conventions are used consistently throughout the book.

- **Assembly code listings** appear inside framed boxes with syntax highlighting and line numbers. They are produced with the `minted` package and follow the standard NASM syntax colouring.
- **Inline code** appears as `nasm -f elf64` or `mov rsi, rdi`.
- **Underscores in identifiers** that appear outside code blocks are written with an escape sequence in the LaTeX source, for example `__libc_start_main`. Within code blocks, underscores are literal and need no special handling.
- **Commands** to be typed into a terminal are shown in their own code blocks, typically without line numbers but with syntax highlighting for plain text.
- **Bit ranges** follow the Intel notation, e.g., `bits[31:0]`.
- **Flag names** are written as `CF`, `ZF`, `OF`, etc.

How to Read the Code Examples

Every listing in this book is a complete, assemble-ready `.asm` file, except when a fragment is explicitly marked as a snippet. At the top of each full example you will find comments that state the exact commands needed to assemble and link the file.

Code: Typical example header

```
; Assemble: nasm -f elf64 -o myprog.o myprog.asm
; Link:     ld -o myprog myprog.o
```

When you see these comments, you can copy the entire block into a file, save it with the given name, and run the two commands. The program will assemble and link on any Fedora 43 machine that has NASM 2.16 (or later) and binutils installed.

Tip

Use the companion code archive. All full example files are available for download from the author's website. This saves typing and guarantees that the line numbers in the book match exactly what you see in your editor.

Understanding the Boxed Annotations

Five types of coloured boxes are used to draw your attention to supplementary information.

Tip

Tips share shortcuts, best practices, and undocumented behaviour that can save you time or improve code quality.

Warning

Warnings describe common pitfalls that cause crashes, memory corruption, or security holes. Read these carefully.

Note

Notes add background theory or historical context. They are optional but deepen your understanding of why things work the way they do.

Caution

Cautions highlight cases where the Linux ABI or the processor behaviour diverges from what you might expect based on generic documentation.

Important

Important marks absolute requirements. If a rule appears in an Important box, violating it will result in an incorrect or unstable program.

Assembling and Running the Examples Yourself

To get the most out of this book, you should have a Fedora 43 machine with NASM and the GNU linker installed. The examples are tested with the default toolchain:

1. **Standard:** NASM + GNU ld (from binutils). This setup uses only packages from the official Fedora repositories. It is perfect for learning and for small-to-medium projects.
2. **Alternative linkers:** You may also use `gold` or `lld` if preferred; any ELF-compatible linker will work.

Installation instructions are provided in Chapter 7. For now, you can verify that NASM is working by opening a terminal and typing:

```
nasm -v
```

You should see a response similar to `NASM version 2.16.01 compiled on ...`. If you do, you are ready to assemble your first program.

Prerequisite Knowledge

This book assumes that you are comfortable with basic programming concepts: variables, loops, functions, and a rough idea of how memory is organised. You do not need any prior assembly experience. However, if you have never programmed before, you may find it helpful to study a high-level language first; the book does not spend time explaining what a variable is or why an operating system exists.

Note

If you are coming from a Windows background, the calling convention and system-call interface described here are completely different from those used on Windows. Even the assembler directives differ in subtle ways. Treat this book as a fresh start.

A Rapid Demonstration

To give a concrete feel for the workflow, here is a minimal but complete program that writes a greeting to the terminal and returns.

Code: greet.asm — Immediate gratification

```
; Assemble: nasm -f elf64 -o greet.o greet.asm
; Link:     ld -o greet greet.o

bits 64
default rel

section .data
msg:  db 'Welcome to NASM on Fedora 43!', 10
len   equ $ - msg

section .text
global _start
_start:
```

```
    ; sys_write(1, msg, len)
    mov     rax, 1
    mov     rdi, 1
    lea     rsi, [msg]
    mov     rdx, len
    syscall

    ; sys_exit(0)
    mov     rax, 60
    xor     rdi, rdi
    syscall
```

Create a file named `greet.asm` with this content, then run:

```
nasm -f elf64 -o greet.o greet.asm
ld -o greet greet.o
./greet
```

If you see the greeting, you are ready to dive into the book. If not, check the installation steps in Chapter 7.

Caution

System-call numbers are stable. The numbers 1 (write), 60 (exit), and 0 (read) have been unchanged on x86-64 Linux since kernel 2.6. They will not change in Fedora 43. You can verify them in `/usr/include/asm/unistd_64.h`.

What to Expect at the End of Each Chapter

Each chapter ends with a *Summary* section that recaps the key concepts, followed by *Practice Exercises*. The exercises are designed to be short and focused. A later appendix provides solution hints, but you are encouraged to attempt them unaided.

Staying Current

Fedora 43 and the Linux kernel continue to evolve. This book targets the stable Fedora 43 release and uses system-call interfaces that are guaranteed to remain backward-compatible. When significant changes to the platform occur that affect assembly programming, the author will publish notes on the companion website. You can keep NASM itself updated by running `sudo dnf upgrade nasm`.

Important

Before you begin, ensure you are using at least NASM 2.16. Earlier versions may not support some of the directive syntax or output formats used here.

A Personal Note on Assembly Language

Programming in assembly is unlike working in any high-level language. It requires patience and a willingness to understand the machine at its most fundamental level. The reward is total control and a clarity of execution that is hard to achieve otherwise. This book is written to make that

journey as direct and productive as possible. Every concept is explained, every term is defined, and every example is ready to run.

Turn the page and begin. The first chapter will have you assembling code within minutes.

Development Environment Overview

Choosing the Right Toolchain

Writing 64-bit assembly for Linux requires only two mandatory tools: the assembler itself and a linker. Everything else — an editor, a debugger, a build script — is a matter of personal preference. This chapter describes the canonical setup recommended for this book, which has been verified on clean Fedora 43 installations. All components are free of charge, actively maintained, and backed by official documentation.

The Assembler: NASM 2.16 or Later

The Netwide Assembler (NASM) translates `.asm` source files into 64-bit object files in the ELF format required by the GNU toolchain. The official website is `nasm.us`. The current stable series is 2.16.x, which introduced several improvements for ELF target support including enhanced debugging information.

Installing NASM on Fedora

NASM is available directly from the Fedora repositories. Open a terminal and run:

```
sudo dnf install nasm
```

This installs the latest version packaged for Fedora 43, which is at least 2.16. Verify the installation with:

```
nasm -v
```

Expected output resembles:

```
NASM version 2.16.01 compiled on Mar 15 2026
```

No manual `PATH` adjustments are necessary; the binary is placed in `/usr/bin`.

Installing from Source (Optional)

If you need an even newer version, download the source tarball from `nasm.us` and follow the standard `./configure && make && sudo make install` procedure. The binary will be placed in `/usr/local/bin`.

Warning

Older NASM versions, particularly the 2.15.x series, contain subtle bugs in the `elf64` output format that can cause incorrect relocations or problems with debug sections. Do not rely on any version before 2.16 for the examples in this book.

The Linker: GNU ld

The assembler produces an object file (`.o`) that contains machine code and unresolved symbol references. A linker is needed to combine one or more objects and produce the final executable (an ELF binary). The standard linker on Fedora is `ld` from GNU binutils. It is almost certainly already installed as part of the development tools. If not, you can add it with:

```
sudo dnf install binutils
```

Linking a program that uses only pure assembly (no external libraries) is simple:

```
ld -o myprog myprog.o
```

The linker reads the ELF object produced by NASM and generates an executable. Because we are not using the C library, we do not need to specify any startup files. The linker will set the entry point according to the symbol `_start` declared `global` in the source.

Linking with the C Library

If your program calls C functions (e.g., `printf`), you must link against the C library and provide a `main` function instead of `_start`. A typical command is:

```
nasm -f elf64 -o myprog.o myprog.asm
gcc -no-pie -o myprog myprog.o
```

Here `gcc` acts as a wrapper that adds the necessary startup files and links `libc`. We will explore this in Part V.

The Editor

Any text editor capable of handling plain ASCII or UTF-8 files without formatting is suitable. The following are widely used on Linux:

- **Gedit** or **GNOME Text Editor** — simple, preinstalled, with syntax highlighting for assembly available via GtkSourceView.
- **Visual Studio Code** — with the `ASM Code Lens` or `asmsyntax` extension, it provides syntax highlighting, bracket matching, and an integrated terminal where you can run `nasm` and `ld`.
- **Vim** or **Neovim** — for users comfortable with modal editing; syntax files for NASM are readily available.

- **Emacs** — with `nasm-mode` for syntax colouring.

Choose the tool you know best. No IDE is required; assembly coding benefits from a simple, distraction-free editing environment.

Build Automation

For projects containing more than one source file, typing assembly and link commands manually becomes tedious. Three approaches are common:

Shell Script

A simple `build.sh` script:

```
#!/bin/sh
nasm -f elf64 -o prog.o main.asm
if [ $? -ne 0 ]; then exit 1; fi
ld -o prog prog.o
```

Make it executable with `chmod +x build.sh`. Running `./build.sh` assembles and links in one step.

Makefile

A standard `Makefile` that works with GNU make:

```
prog: main.o
^^Ild -o prog main.o

main.o: main.asm
^^Inasm -f elf64 -o main.o main.asm
```

Running `make` rebuilds only what has changed.

Task Scripts in VS Code

VS Code can define tasks in `.vscode/tasks.json` that invoke NASM and `ld` with the appropriate arguments. This allows building with a single keyboard shortcut while still seeing errors marked in the editor.

Debugging Assembly on Fedora

Debugging assembly code requires a debugger that can handle 64-bit executables and display registers, flags, memory, and disassembly side by side.

GDB (GNU Debugger)

GDB is the standard debugger on Linux and is part of the development tools. Install it if it is not already present:

```
sudo dnf install gdb
```

To debug a small program, start GDB with the executable:

```
gdb ./prog
```

Inside GDB, you can use commands like:

```
break _start        ; set breakpoint at the entry point
run                 ; start the program
info registers      ; display all registers
stepi               ; step one machine instruction
disassemble         ; show disassembly around current instruction
```

GDB supports source-level debugging if you assemble with debugging symbols (`nasm -f elf64 -g ...`).

DDD and Other GUI Frontends

For those who prefer a graphical interface, **DDD** (Data Display Debugger) is a frontend to GDB. It provides a visual register window, memory map, and source view. You can also use the debugging interface built into VS Code with the GDB adapter.

Tip

When debugging an assembly program without debugging symbols, you can set a breakpoint directly on `_start`. Use `layout asm` in GDB to see a TUI disassembly while stepping through instructions. Watch the `RAX` register after each `syscall` to check for error returns (a negative `errno` value indicates failure).

Verifying the Full Toolchain with a Test Program

Before writing anything complex, confirm that every component works together. Create a file named `envtest.asm` with the following content.

Code: envtest.asm — Toolchain verification

```
; Assemble: nasm -f elf64 -o envtest.o envtest.asm
; Link:     ld -o envtest envtest.o

bits 64
default rel

section .data
msg:  db 'Toolchain is functioning.', 10
len   equ $ - msg

section .text
global _start
_start:
    mov     rax, 1
    mov     rdi, 1
    lea     rsi, [msg]
    mov     rdx, len
    syscall

    mov     rax, 60
```

```
    xor     rdi, rdi
    syscall
```

Open a terminal in the directory containing `envtest.asm` and run:

```
nasm -f elf64 -o envtest.o envtest.asm
ld -o envtest envtest.o
./envtest
```

The expected output is a single line:

```
Toolchain is functioning.
```

If this appears, the toolchain is correctly installed. If not, check that `nasm` is in your `PATH` and that `ld` is installed (command `which ld`).

Caution

File permissions. Unlike Windows, a newly linked executable does not automatically have execute permission. If you see "Permission denied", run `chmod +x envtest` and try again.

Understanding the File Types

During development you will encounter several file extensions:

- **.asm** — source text file containing NASM assembly.
- **.o** — ELF object file produced by NASM. Contains machine code, symbol definitions, and relocations. Not yet executable.
- **executable (no extension)** — final ELF executable created by the linker. On Linux, executables commonly have no extension.
- **.so** — shared object (dynamically linked library). Part VII covers creating and using them.
- **.a** — static library archive. You rarely create these in pure assembly, but they can be linked with other languages.

Enabling Debugging Symbols

Adding debugging information to your executable makes source-level debugging painless. NASM can produce DWARF debug data with the `-g` switch:

```
nasm -f elf64 -g -o myprog.o myprog.asm
ld -o myprog myprog.o
```

GDB will then show the source code and allow you to refer to variable names instead of raw addresses.

Common Pitfalls During Setup

Even with the correct toolchain, a few mistakes are frequent.

Warning

Missing `_start` symbol. If you forget to mark `_start` as `global`, the linker will not find an entry point. The error is "undefined reference to `_start`". Ensure your text section contains `global _start` and that the symbol is defined.

Warning

Using the wrong file format. Never assemble for a different output format. For 64-bit Linux, always use `nasm -f elf64`. The `-f elf` switch produces 32-bit code and will not link correctly.

Note

Some virus scanners running under Wine or on dual-boot systems may incorrectly flag very small ELF executables as suspicious. This is a false positive caused by the minimal header. You can safely ignore it or use `strip` to reduce the file size if needed.

Hardware Requirements

Any Intel or AMD processor that supports the x86-64 instruction set will work. Fedora 43 itself already mandates such a processor. The examples use only baseline SSE instructions unless otherwise noted. No special hardware beyond a standard PC is required.

Looking Ahead

With a working toolchain, you are ready to write, build, and debug native 64-bit assembly on Fedora 43. The next chapter explores the structure of a minimal ELF executable and explains the System V AMD64 calling convention in painstaking detail — the knowledge that turns assembly from a collection of mnemonics into a reliable programming language.

Important

Take a few minutes to verify the toolchain now. Every subsequent chapter assumes that you can assemble and link the examples. If something is not working, resolve it before moving on. The entire book is built on this foundation.

Part I

FOUNDATIONS OF x86-64 ARCHITECTURE

1

Introduction to NASM and Assembly Language

In This Chapter

This chapter lays the conceptual groundwork for everything that follows. You will learn what assembly language is at the machine level, how the Netwide Assembler (NASM) translates human-readable source into executable code, why assembly still matters on modern Linux systems such as Fedora 43, and how the `x86-64` architecture differs from its 32-bit predecessor. Finally, you will receive a high-level view of the processor execution pipeline, knowledge that directly influences how you write efficient assembly.

1.1 What is Assembly Language

Assembly language is the human-readable representation of a processor's native machine code. Every instruction in the `x86-64` instruction set has a binary encoding defined in the Intel and AMD manuals, and assembly provides a set of mnemonics – short symbolic names – that correspond one-to-one with those binary opcodes. For example, the machine byte `48 89 C8` corresponds to the assembly instruction `mov rax, rcx` on an x64 processor.

Assembling is the process of converting these mnemonics, register names, and memory references into the exact byte sequence that the CPU can execute. The program that performs this translation is called an assembler. Unlike a compiler for a high-level language, an assembler performs an almost straightforward mapping; there is no complex optimisation phase, no register allocation algorithm, and no hidden run-time library calls. What you write is what the machine runs.

Assembly language is specific to a processor architecture. The assembly for an ARM chip is completely different from `x86-64` assembly, even if the high-level logic of a program could be identical.

This book deals exclusively with the `x86-64` architecture, as implemented by Intel and AMD processors, under the Linux operating system, specifically Fedora 43.

A First Glimpse at Assembly Code

Below is a single line of NASM assembly. It moves the 64-bit value stored in register `RCX` into register `RAX`.

```
mov rax, rcx
```

In machine language, this becomes `48 89 C8` (three bytes). The mnemonic `mov` tells the assembler to produce a move instruction; the operands specify the destination and source. This one-to-one correspondence is the defining characteristic of assembly language.

Why Learn Assembly on Linux?

Fedora 43 is a modern 64-bit Linux distribution that gives you direct access to the kernel through system calls, without the overhead of heavy frameworks. Every running program uses the x64 registers, follows the System V AMD64 calling convention, and interfaces with the kernel via the `syscall` instruction. Writing assembly natively for this environment gives you complete control over the processor while allowing you to call directly into the rich set of Linux system calls. Knowledge of assembly also makes you a better debugger; when a program crashes and you open the disassembly view in GDB, you need to understand what the displayed instructions mean.

1.2 What is NASM

NASM, the Netwide Assembler, is an open-source, cross-platform assembler for the x86 and `x86-64` architectures. Initially developed by Simon Tatham and Julian Hall, it has been continuously maintained and improved by a dedicated team of volunteers. As of 2026, the current stable series is 2.16.x, which fully supports the latest instruction sets including AVX-512 and AMX.

NASM is distinguished from other assemblers by its clear, consistent syntax, its powerful macro processor, and its ability to generate a wide range of output formats. For Linux development, the most important format is `elf64`, which produces 64-bit ELF object files compatible with the GNU linker (`ld`).

Key Features of NASM

- **Intel-style syntax.** NASM uses the well-known Intel destination-source operand order, e.g., `mov dest, src`. This is the natural syntax for anyone reading Intel or AMD manuals.
- **Flat, segmented, and relocatable output.** NASM supports generation of raw binary files, DOS COM files, 16-bit and 32-bit OMF objects, and modern 64-bit ELF for Linux and COFF for Windows.
- **Extensive macro support.** The preprocessor provides single-line and multi-line macros, conditional assembly, and include files, allowing code reuse that far exceeds simple copy-and-paste.

- **Direct support for `x86-64` instruction extensions.** Every published Intel and AMD instruction up to the latest generation is implemented, including AVX-512, SHA-NI, and virtualization extensions.
- **Clear error and warning messages.** NASM reports syntax errors with line numbers, making it easy to locate mistakes.
- **Free and open-source.** The source code is available under a BSD license, so there is no cost, and the assembler can be trusted to remain available indefinitely.

A Complete NASM Program for Fedora 43

The following listing assembles, links, and runs on a fresh Fedora 43 system. It illustrates the absolute minimum structure of a NASM source file targeting the `elf64` output format.

Code: listing1-1.asm — Minimal Console Output with NASM

```
; Assemble: nasm -f elf64 -o listing1-1.o listing1-1.asm
; Link:     ld -o listing1-1 listing1-1.o

bits 64
default rel

section .data
msg:  db 'NASM is working on Fedora 43!', 10
len   equ $ - msg

section .text
global _start
_start:
    ; sys_write(unsigned int fd, const char *buf, size_t count)
    mov     rax, 1            ; syscall number for sys_write
    mov     rdi, 1            ; fd = stdout
    lea     rsi, [msg]        ; buf  = address of message
    mov     rdx, len          ; count = message length
    syscall

    ; sys_exit(int error_code)
    mov     rax, 60           ; syscall number for sys_exit
    xor     rdi, rdi          ; error_code = 0
    syscall
```

Every element in this program will be explained in detail later. For now, observe how a few directives (`bits`, `section`) and only a handful of instructions produce a complete, running executable. That is the power of NASM: no hidden scaffolding, no run-time interpreter, only what the processor actually executes.

1.3 Role of Assembly in Modern Systems

Even in a world dominated by high-level languages and just-in-time compilers, assembly language remains irreplaceable in several domains.

System Software and Kernels

The very lowest layers of an operating system – context switching, interrupt service routines, system call entry points, and hardware abstraction – are written in assembly. The Linux kernel contains thousands of lines of hand-tuned assembly for cache management, spin-locks, and access to model-specific registers.

Performance-Critical Routines

Compiler-generated code is excellent for most tasks, but manual assembly can still beat it in tight loops that require precise scheduling of instructions, exploitation of obscure instructions, or avoidance of register spills. Image codecs, cryptographic primitives, and scientific computing kernels often contain inner loops written in assembly.

Security Research and Reverse Engineering

Vulnerability analysis, malware dissection, and exploit development all rely on the ability to read and write assembly. When you examine a binary in a disassembler such as IDA Pro or Ghidra, what you see is essentially NASM-style mnemonics. Writing your own assembly programs is the fastest way to become fluent in reading disassemblies.

Bootstrapping and Firmware

The first code that runs after a CPU reset – the bootloader, firmware, and early initialisation sequences – is written in assembly. Even UEFI applications are often a mix of C and assembly, with the entry point and page-table setup done in pure assembly.

Education and Understanding

There is no better way to understand how a computer truly works than to write a program that controls it at the instruction level. Concepts such as the stack, pointers, calling conventions, and memory alignment move from abstract ideas to concrete realities when you are forced to manage them explicitly.

1.4 x86 vs x86-64 Overview

The `x86-64` architecture (also called AMD64, Intel 64, or simply x64) is a 64-bit extension of the classic 32-bit x86 design. All current Linux systems are exclusively 64-bit, and Fedora 43 no longer ships a 32-bit kernel. Native programs always execute in 64-bit long mode.

Register File Expansion

The most visible difference between x86 and `x86-64` is the size and number of general-purpose registers.

- In x86 (32-bit), there are eight general-purpose registers: `EAX`, `EBX`, `ECX`, `EDX`, `ESI`, `EDI`, `EBP`, `ESP`, each 32 bits wide.

- In x86-64 (64-bit), these registers are extended to 64 bits and renamed as RAX, RBX, RCX, RDX, RSI, RDI, RBP, RSP. The lower 32 bits of each register can still be accessed using the old 32-bit names (e.g., EAX refers to the low 32 bits of RAX).
- Additionally, eight entirely new 64-bit registers are provided: R8 through R15. Their low 32-bit, 16-bit, and 8-bit portions are accessed as R8D–R15D, R8W–R15W, and R8B–R15B.

The following NASM snippet shows 64-bit register usage in context.

```
; Load a 64-bit immediate and copy it to another extended register
mov  rax, 0x7FFFFFFFFFFFFFFF
mov  r12, rax           ; R12 is one of the new R8-R15 registers
```

Address Space and Pointers

x86-64 extends the virtual address space from 4 GB to a theoretical maximum of 256 TB. In practice, Linux on x64 provides a 128 TB user-mode address space. All pointers stored in registers are 64 bits wide, and memory addressing modes can use any of the 16 general-purpose registers as a base or index, with optional scale factors of 1, 2, 4, or 8.

A typical RIP-relative address reference in NASM looks like this:

```
lea rsi, [rel msg]      ; load address of 'msg' relative to RIP
```

This replaces the older absolute addressing of x86 and is essential for position-independent code required by modern Linux systems.

Calling Conventions

The biggest practical change for the assembly programmer is the new calling convention. The System V AMD64 ABI (used by Linux) specifies that integer and pointer arguments are passed in RDI, RSI, RDX, RCX, R8, and R9 (in that order). System calls use a different convention: the call number goes in RAX, and arguments go in RDI, RSI, RDX, R10, R8, and R9. The stack must be 16-byte aligned before any CALL instruction, and there is no shadow space requirement. This is simpler and more regular than the Microsoft x64 convention.

The following example demonstrates a function call with three arguments using the System V ABI:

```
; Function: ssize_t write(int fd, const void *buf, size_t count)
mov  rdi, 1             ; fd = stdout
lea  rsi, [rel buffer]  ; buf
mov  rdx, buffer_len    ; count
call write              ; call the C library function
```

Instruction Set Extensions

x86-64 mandates the presence of SSE and SSE2, which provide 128-bit XMM registers for floating-point and integer SIMD operations. Later extensions (AVX, AVX2, AVX-512) add the 256-bit YMM and 512-bit ZMM registers. NASM fully supports all these extensions; a few examples appear in Part IV.

Operating Mode Differences

When the CPU boots, it starts in 16-bit real mode, transitions to 32-bit protected mode, and finally enters 64-bit long mode. Linux sets up long mode during boot and never leaves it. For the assembly developer, the only mode that matters is long mode with flat memory segmentation. Segment registers still exist but are essentially unused except for special selectors like FS (used for thread-local storage). All code, data, and stack reside in a single flat 64-bit virtual address space.

1.5 Execution Pipeline Concept

Modern `x86-64` processors do not execute instructions one at a time. Instead, they break each instruction into a series of micro-operations and process them in a pipeline, much like an assembly line. Understanding the pipeline helps you write code that keeps the processor's execution units busy and avoids stalls.

Classic Five-Stage Pipeline

A simplified view of the pipeline consists of five stages:

1. **Fetch.** The instruction fetch unit reads raw bytes from the L1 instruction cache at the address pointed to by the instruction pointer (`RIP`).
2. **Decode.** The fetched bytes are broken into micro-operations (uops) and the operands (registers, memory addresses) are identified.
3. **Execute.** The uops are dispatched to the appropriate execution units: integer ALU, floating-point, branch unit, or memory load/ store. Multiple uops can execute in parallel if they are independent.
4. **Memory Access.** If an instruction loads from or stores to memory, the data cache is accessed here.
5. **Writeback.** The results are written back to the register file.

Under optimal conditions, one instruction can complete every clock cycle per execution unit, even though its traversal through the pipeline takes many cycles. This apparent throughput is achieved by pipelining: while one instruction is in execute, the next is in decode, and the one after that is being fetched.

Data Hazards and Stalls

The pipeline can stall when an instruction depends on the result of a previous instruction that has not yet completed. In assembly code, this is called a data dependency.

Consider this sequence:

```
add rax, rcx      ; RAX = RAX + RCX
sub rbx, rax      ; RBX = RBX - RAX  (depends on ADD)
```

The `sub` instruction cannot proceed until the `add` has written back its result. Modern processors use register renaming and out-of-order execution to hide some of this latency, but the fundamen-

tal dependency still limits the rate at which the chain can execute. If you interleave unrelated instructions between the two, you can often improve throughput:

```
add rax, rcx
mov r8, [rdx]      ; independent load
and r9, 0xFF       ; independent operation
sub rbx, rax       ; now RAX is ready
```

Branch Prediction

When the CPU encounters a conditional jump, it does not wait to know whether the branch will be taken. It predicts the outcome based on history and speculatively fetches and executes instructions along the predicted path. If the prediction is wrong, the pipeline must be flushed and the correct path fetched, costing tens of cycles. Writing assembly with predictable branching patterns can dramatically improve performance in tight inner loops.

Practical Advice for the NASM Programmer

On Fedora 43, where every system call triggers a kernel entry and a chain of kernel operations, fine-grained pipeline effects inside your own functions generally have a modest impact. But in compute-bound routines – a tight loop performing arithmetic on large arrays – pipeline awareness can make the difference between a routine that runs at memory bandwidth and one that leaves the processor waiting.

Tip

Use the LLVM Machine Code Analyzer (`llvm-mca`) or the Intel Architecture Code Analyzer (IACA) to statically analyze your assembly routines. These tools model the exact pipeline of modern Intel and AMD CPUs and can highlight stalls and port conflicts, providing a concrete estimate of throughput.

Putting It All Together

The `x86-64` processor is a masterpiece of engineering, and assembly is the language through which you can directly control it. NASM gives you a clean, modern interface for writing that control in an understandable format. Throughout this book, you will progressively master the instruction set, the Linux system-call interface, and the optimization techniques that make assembly programming not only feasible but genuinely rewarding.

Important

Do not worry if the pipeline explanation seems abstract at this stage. As you work through the practical examples and begin measuring the performance of your own loops, the concepts will crystallise. For now, retain the key principle: independent instructions can execute in parallel, and dependencies create delays.

2

Overview of x86-64 Architecture

In This Chapter

This chapter provides an essential survey of the x86-64 processor architecture as seen from the perspective of an assembly language programmer on Linux, specifically Fedora 43. You will learn the layout of a modern x64 CPU, the full register set, the structure of machine instructions, the little-endian byte ordering, and the memory addressing modes that the processor offers. Every concept is grounded in official Intel and AMD documentation and demonstrated with NASM code that you can assemble, link, and inspect on your own machine.

2.1 CPU Architecture Overview

The x86-64 processor at the heart of every modern PC is a deeply complex machine, but for the assembly programmer a simplified model suffices.

- **Execution Cores.** Each physical core has its own set of integer execution units, floating-point units, and a private register file. When your program runs, it occupies one core at a time, executing a single thread of instructions.
- **Registers.** The fastest storage in the machine, located directly on the core. The general-purpose registers, instruction pointer, and flags are the programmer's principal interface to the CPU.
- **Caches.** An on-die hierarchy of fast memory: L1, L2, and often L3 caches. The L1 instruction cache feeds the fetch stage of the pipeline and the L1 data cache services most memory operands within a few cycles.
- **Memory Management Unit (MMU).** Translates the 64-bit virtual addresses used by instructions into physical addresses. Linux sets up multi-level page tables so that each process

sees a flat, private address space.

- **Interrupt Controller and APIC.** Handles external events and inter-processor interrupts. Most assembly programmers do not interact with this directly; the kernel abstracts it.

Linux boots the processor into *long mode*, the 64-bit operating mode where all registers and the full instruction set are available. In long mode the memory model is flat: the segment registers (CS, DS, SS, ES, GS, FS) are mostly unused, except for FS which is set by the kernel to point to the Thread-Local Storage (TLS) area for each thread. All code, data, and stack references use a single 64-bit virtual address space.

Note

The detailed behaviour of the MMU, paging, and privilege levels is important for kernel development but rarely needed for user-mode Linux programs. This book focuses on ring-3 (user-mode) programming, where the OS handles the page tables and privileges for you.

2.2 Registers Overview

The register file is the assembly programmer's workbench. The `x86-64` architecture provides sixteen 64-bit general-purpose registers, an extended instruction pointer, a flags register, and a large set of SIMD registers.

2.2.1 General-Purpose Registers

The registers `RAX`, `RBX`, `RCX`, `RDX`, `RSI`, `RDI`, `RBP`, `RSP` are the original eight from 32-bit x86, now widened to 64 bits. The added registers `R8` through `R15` have no legacy 8-bit aliases; their smallest parts are `R8B` and so on.

The table below lists all general-purpose registers and their sub-divisions.

64-bit	32-bit	16-bit	8-bit low	8-bit high (REX)
RAX	EAX	AX	AL	AH*
RBX	EBX	BX	BL	BH*
RCX	ECX	CX	CL	CH*
RDX	EDX	DX	DL	DH*
RSI	ESI	SI	SIL	—
RDI	EDI	DI	DIL	—
RBP	EBP	BP	BPL	—
RSP	ESP	SP	SPL	—
R8	R8D	R8W	R8B	—
R9	R9D	R9W	R9B	—
R10	R10D	R10W	R10B	—
R11	R11D	R11W	R11B	—
R12	R12D	R12W	R12B	—
R13	R13D	R13W	R13B	—
R14	R14D	R14W	R14B	—
R15	R15D	R15W	R15B	—

* The 8-bit high registers AH, BH, CH, DH are only accessible when no REX prefix is used; in 64-bit mode they can be used if the instruction does not need a REX prefix.

Warning

Zero-extension rule. Writing to a 32-bit register (EAX, R8D) automatically zero-extends the upper 32 bits of the corresponding 64-bit register. Writing to an 8-bit or 16-bit register *does not* zero-extend; it leaves the upper bits unchanged. Therefore, when you need a 32-bit value zero-extended into a 64-bit register, use the 32-bit name.

2.2.2 Special-Purpose Registers

- RIP – **Instruction Pointer.** Holds the virtual address of the next instruction to execute. Directly writable only by control-flow instructions (JMP, CALL, RET). Reading RIP is done via an LEA with RIP-relative addressing: `lea rax, [rel next_instr]`.
- RFLAGS – **Flags Register.** Contains status flags (CF, ZF, SF, OF, PF) and the direction flag (DF). The flags are set by arithmetic and logical instructions and tested by conditional jumps.
- RSP – **Stack Pointer.** Points to the top of the current stack frame. Modified implicitly by PUSH, POP, CALL, RET and explicitly by arithmetic.
- **Segment Registers** CS, DS, SS, ES, FS, GS. In Linux user-mode, FS is often used by the thread library to point to thread-local storage; the kernel sets it up.

2.2.3 Checking Registers in a Debugger

The following minimal NASM program loads a few registers with easily recognisable values and then exits. It is not intended to produce visible output; instead, assemble it and step through the code in a debugger to verify the register contents.

Code: listing2-1.asm — Register loading for debugger inspection

```
; Assemble: nasm -f elf64 -o listing2-1.o listing2-1.asm
; Link:     ld -o listing2-1 listing2-1.o

bits 64
default rel

section .text
global _start
_start:
    mov     rax, 0AAAAAAAAAAAAAAAAh
    mov     rbx, 0BBBBBBBBBBBBBBBBh
    mov     rcx, 0CCCCCCCCCCCCCCCCh
    mov     rdx, 0DDDDDDDDDDDDDDDDh
    mov     r8,  0x8888888888888888
    mov     r9,  0x9999999999999999
    mov     r10, 0x1010101010101010
    mov     r12, 0x1212121212121212
```

```
    ; exit(0)
    mov     rax, 60                 ; sys_exit
    xor     rdi, rdi
    syscall
```

Set a breakpoint on the final `xor rdi, rdi` line in GDB and inspect the registers with `info registers`. You will see exactly the values loaded, confirming the register file layout.

2.3 Instruction Set Structure

Every assembly instruction is a human-readable mnemonic for a fixed or variable-length binary encoding. The official Intel SDM Volume 2 gives the complete encoding rules; here we summarise the parts you will see when examining NASM output.

2.3.1 Instruction Encoding

A typical `x86-64` instruction consists of the following bytes, in order:

1. Optional legacy prefixes (up to four bytes): `66`, `67`, `F2`, `F3`, `REX` (`40`–`4F`).
2. Opcode: one, two, or three bytes.
3. ModR/M byte (if required): specifies the addressing mode and register/memory operands.
4. SIB byte (if ModR/M indicates index register): Scale-Index-Base.
5. Displacement: a 1-, 2-, or 4-byte signed offset added to the effective address.
6. Immediate: a constant value, 1/2/4/8 bytes.

The `REX` prefix is critical. Its 4 bits carry the additional high bits for registers (allowing access to `R8`–`R15` and the 64-bit operand size) and the sign-extension of 32-bit immediates.

NASM shows the encoding when you add the `-l listing.lst` switch. A small example:

Code: Listing file snippet

```
48 89 C8          mov rax, rcx
49 89 D2          mov r10, rdx
```

The byte `48` is a REX.W prefix (promotes to 64-bit operand). `49` is REX.W with the B bit set because `R10` is register 10 (bits 3-4).

2.3.2 Instruction Categories

The `x86-64` instruction set can be grouped into a few broad families:

- **Data movement:** `MOV`, `LEA`, `XCHG`, `PUSH`, `POP`, `MOVZX`, `MOVSX`.
- **Arithmetic:** `ADD`, `SUB`, `INC`, `DEC`, `MUL`, `IMUL`, `DIV`, `IDIV`, `XADD`, `ADC`, `SBB`.
- **Logical and bit:** `AND`, `OR`, `XOR`, `NOT`, `SHL`, `SHR`, `SAR`, `ROL`, `ROR`, `BTS`, `BTR`.

- **Control flow:** `JMP`, `Jcc` (conditional jumps), `CALL`, `RET`, `LOOP`, `INT`, `SYSCALL`.
- **SIMD (SSE/AVX):** `MOVD`, `MOVQ`, `PADDB`, `PADDW`, `MULPS`, `VEX`-prefixed instructions.

In this book we use the Intel-syntax operand order: `MOV destination, source`. NASM strictly follows this convention.

2.3.3 Encoding Exercise in NASM

You can create a listing file to see the actual bytes that NASM produces:

```
nasm -f elf64 -l encode.lst encode.asm
```

The file `encode.lst` contains the source lines interleaved with the encoded bytes. This is an excellent way to learn the relationship between assembly and machine code.

2.4 Little Endian Representation

`x86-64` processors store multi-byte values in *little-endian* order: the least significant byte occupies the lowest memory address.

Consider the 32-bit value `0x12345678`. In memory, starting at address `X`, the bytes will be:

Address	Byte
X	0x78
X+1	0x56
X+2	0x34
X+3	0x12

2.4.1 Observing Little Endian in Memory

The following program stores a known 32-bit value in the `.data` section and then isolates the four bytes into a buffer where you can examine them in a debugger.

Code: listing2-2.asm — Little endian byte inspection

```
; Assemble: nasm -f elf64 -o listing2-2.o listing2-2.asm
; Link:     ld -o listing2-2 listing2-2.o

bits 64
default rel

section .data
myval   dd  0x12345678          ; 32-bit value in .data

section .bss
bytes   resb 4                  ; buffer for individual bytes

section .text
global _start
_start:
    mov     eax, [rel myval]    ; load the whole dword
```

```
    ; extract bytes one at a time
    movzx   ecx, al                 ; lowest byte (0x78)
    mov     [rel bytes+0], cl
    movzx   ecx, ah                 ; second byte (0x56)
    mov     [rel bytes+1], cl
    shr     eax, 16                 ; now AX has upper 16 bits
    movzx   ecx, al                 ; third byte (0x34)
    mov     [rel bytes+2], cl
    movzx   ecx, ah                 ; fourth byte (0x12)
    mov     [rel bytes+3], cl

    ; exit(0)
    mov     rax, 60
    xor     rdi, rdi
    syscall
```

After running this program, place a breakpoint on the `mov rax, 60` line in GDB and inspect the four bytes at address `bytes` with `x/4xb &bytes`. A memory dump will show:

```
0x78 0x56 0x34 0x12
```

This confirms that the least significant byte (0x78) is at the lowest address.

Tip

When reading register values in GDB, the debugger displays values in human-readable big-endian format (the mathematical value). But in memory dumps, the bytes are shown in linear address order, which reveals the little-endian storage.

2.5 Memory Addressing Basics

The `x86-64` architecture provides one of the richest memory addressing modes among mainstream processors. An effective address can be computed as:

```
BASE + INDEX * SCALE + DISPLACEMENT
```

where each component is optional, `SCALE` can be 1, 2, 4, or 8, and `DISPLACEMENT` is an 8-, 16-, or 32-bit signed constant.

2.5.1 Addressing Mode Examples

In NASM syntax, the memory operand is written inside square brackets. The following are all valid:

```
mov rax, [rcx]              ; base only
mov rax, [rcx + 8]          ; base + 8-bit displacement
mov rax, [rcx + rdx]        ; base + index (scale 1)
mov rax, [rcx + rdx*4]      ; base + index * scale
mov rax, [rcx + rdx*4 + 16] ; base + index*scale + disp
mov rax, [rsp + 32]         ; stack-relative
```

2.5.2 RIP-Relative Addressing

Under 64-bit Linux, position-independent executables (PIE) require that global data references be RIP-relative. NASM provides the `rel` keyword to generate a RIP-relative address:

```
lea rsi, [rel mydata]       ; load address of mydata relative to RIP
mov eax, [rel counter]      ; load value at 'counter' via RIP-relative
```

The directive `default rel` at the top of a source file makes all unadorned memory references RIP-relative automatically.

2.5.3 A Complete Example: Summing an Array

The following program uses indexed addressing to sum an array of 32-bit integers. It exits with the sum as its exit code, which you can check with `echo $?` after running the program.

Code: listing2-3.asm — Summation with indexed addressing

```
; Assemble: nasm -f elf64 -o listing2-3.o listing2-3.asm
; Link:     ld -o listing2-3 listing2-3.o

bits 64
default rel

section .data
array   dd  10, 20, 30, 40, 50   ; five dwords
len     equ ($ - array) / 4       ; number of elements

section .text
global _start
_start:
    xor     eax, eax              ; accumulator = 0
    xor     ecx, ecx              ; index = 0
    lea     rdx, [rel array]      ; base address of array

.loop:
    add     eax, [rdx + rcx*4]    ; add array[i] to EAX
    inc     ecx
    cmp     ecx, len
    jb      .loop

    ; exit with sum as exit code
    mov     rdi, rax              ; first argument to exit
    mov     rax, 60               ; sys_exit
    syscall
```

The effective address `[rdx + rcx*4]` uses `RDX` as the base, `RCX` as the index, and scale 4 (since each element is 4 bytes). After the loop, `RAX` will contain `10+20+30+40+50 = 150` (0x96). Run the program and then check the exit status:

```
./listing2-3 ; echo $?
```

The output will be `150`, proving the sum.

Important

Displacement limits. In 64-bit mode, the displacement in an effective address is a 32-bit signed value. This allows addressing a 2 GB window relative to a base register, which is more than enough for typical data structures. For an absolute address in a flat memory model, you would normally use a RIP-relative `LEA` to obtain the base.

Working with Different Data Sizes

When using indexed addressing, the operand size determines how many bytes are read or written. Use the appropriate register name to control the size:

```
mov dword [rdx + rcx*4], 0xFFFFFFFF   ; write 32-bit
mov word  [rdx + rcx*2], 0xABCD       ; write 16-bit
mov byte  [rdx + rcx], 0x7F           ; write 8-bit
```

The scale factor must match the element size you are addressing. For a 64-bit array, use scale 8: `[rdx + rcx*8]`.

Caution

Alignment matters. The `x86-64` processor can handle unaligned memory accesses (except for some SIMD instructions), but misaligned access may incur a performance penalty. When possible, align data to its natural boundary: 2-byte values to even addresses, 4-byte values to multiples of 4, etc.

Putting the Pieces Together

You now have a working knowledge of the CPU architecture, registers, instruction encoding, endianness, and memory addressing that form the foundation of the rest of this book. Every subsequent chapter builds on these concepts, adding specific instructions, Linux system-call conventions, and optimization techniques.

3

CPU Registers and Execution Model

In This Chapter

Every computation performed by an x86-64 processor passes through a small set of storage locations placed directly on the chip — the registers. Understanding the register set, the role of each register, and the way the processor fetches and executes instructions is fundamental to writing correct and efficient assembly. This chapter examines the sixteen general-purpose registers, the special register trio RIP, RSP, and RFLAGS, the vestigial but still relevant segment registers, and the classic fetch-decode-execute cycle that underpins all instruction execution. Every description draws on the Intel® 64 and IA-32 Architectures Software Developer's Manual, the AMD64 Architecture Programmer's Manual, and the System V AMD64 ABI used by Linux, and is accompanied by NASM examples that you can assemble and run on Fedora 43.

3.1 General Purpose Registers (RAX–R15)

The x86-64 architecture provides sixteen 64-bit general-purpose registers. The first eight — RAX, RBX, RCX, RDX, RSI, RDI, RBP, RSP — are the legacy registers carried over from the 32-bit x86 and are now widened to 64 bits. The additional eight — R8 through R15 — are entirely new, with no legacy 8-bit high halves.

Full Register Map

The table below lists every 64-bit register, its 32-bit, 16-bit, and 8-bit sub-components as they are named in NASM. When a 32-bit register is written, the upper 32 bits of the corresponding 64-bit register are automatically zero-extended. Writing to an 8-bit or 16-bit register leaves the upper bits unchanged; this is a frequent source of subtle bugs.

64-bit	32-bit	16-bit	8-bit low	8-bit high*
RAX	EAX	AX	AL	AH
RBX	EBX	BX	BL	BH
RCX	ECX	CX	CL	CH
RDX	EDX	DX	DL	DH
RSI	ESI	SI	SIL	—
RDI	EDI	DI	DIL	—
RBP	EBP	BP	BPL	—
RSP	ESP	SP	SPL	—
R8	R8D	R8W	R8B	—
R9	R9D	R9W	R9B	—
R10	R10D	R10W	R10B	—
R11	R11D	R11W	R11B	—
R12	R12D	R12W	R12B	—
R13	R13D	R13W	R13B	—
R14	R14D	R14W	R14B	—
R15	R15D	R15W	R15B	—

*The 8-bit high registers AH, BH, CH, DH are available only in instructions that do not carry a REX prefix. In 64-bit code this is not a severe restriction, but it can surprise programmers attempting to use both R8B and AH in close proximity.

Zero-Extension Rule

Writing to any 32-bit register — EAX, R8D, and so on — forces the upper 32 bits of the 64-bit parent register to zero. This is a deliberate design choice that allows zero-extension without requiring an explicit MOVZX. Writing to an 8-bit or 16-bit register does *not* zero-extend, so if you load a byte into AL and later read RAX, the upper 56 bits may contain stale data. To zero-extend a byte or word, use MOVZX or first move the value into a 32-bit register.

```
; correct zero-extension
xor   eax, eax        ; RAX = 0
mov   al, 0x7F        ; RAX = 0x000000000000007F (zero-extended because EAX was zeroed)
; alternative
movzx rax, byte [buf] ; zero-extend a byte from memory
```

Warning

Stale upper bits. If you set AL without first clearing RAX and later use RAX as an address, the upper bits can cause access violations because they turn a valid 32-bit value into a nonsensical 64-bit pointer. Always be explicit about zero-extension.

Register Roles in the System V AMD64 ABI

The Linux x86-64 calling convention assigns specific roles to certain registers. This classification is not enforced by the hardware but by the operating system and the linker, and you must respect it when calling C library functions, system calls (where the rules differ slightly), or when interfacing with other compiled code.

Register	ABI Role
RAX	Return value (integer / pointer); also holds system-call number.
RDI, RSI, RDX, RCX, R8, R9	First six integer/pointer arguments (left to right).
R10	Volatile; used for the fourth system-call argument.
R11	Volatile; clobbered by the syscall instruction.
RBX, RBP, R12–R15	Non-volatile; must be preserved by callee.
RSP	Stack pointer; must be 16-byte aligned at a CALL instruction.

For direct system calls (using syscall), the convention is slightly different: the call number goes in RAX, and the arguments go in RDI, RSI, RDX, R10, R8, and R9. The kernel preserves all registers except RAX (return value) and RCX and R11 (which are clobbered by the syscall instruction).

When writing your own pure assembly functions that will be called from other assembly, you may use whatever register convention you like, but if you plan to call libc functions or interact with C/C++ code, adhere strictly to the table above.

Tip

Non-volatile registers are expensive. If you need one of the non-volatile registers (RBX, RBP, R12–R15) inside a function that will be called externally, you must save the original value on the stack and restore it before returning. Prefer using volatile registers for scratch work inside leaf functions.

Inspecting General-Purpose Registers

The following program loads a distinctive stamp into every general-purpose register and then enters an infinite loop, allowing you to capture the register state with GDB. Assemble it, run it in the background, and attach GDB to the process.

Code: listing3-1.asm — Full register dump for debugger

```
; Assemble: nasm -f elf64 -o listing3-1.o listing3-1.asm
; Link:     ld -o listing3-1 listing3-1.o

bits 64
default rel

section .text
global _start
_start:
    mov     rax, 0AAAAAAAAAAAAAAAAh
    mov     rbx, 0BBBBBBBBBBBBBBBBh
    mov     rcx, 0CCCCCCCCCCCCCCCCh
    mov     rdx, 0DDDDDDDDDDDDDDDDh
    mov     rsi, 0x1111111111111111
    mov     rdi, 0x2222222222222222
    mov     rbp, 0x3333333333333333
    mov     r8,  0x8888888888888888
    mov     r9,  0x9999999999999999
    mov     r10, 0x1010101010101010
    mov     r11, 0x1A1A1A1A1A1A1A1A
```

```
    mov     r12, 0x1212121212121212
    mov     r13, 0x1313131313131313
    mov     r14, 0x1414141414141414
    mov     r15, 0x1515151515151515

    ; infinite loop - attach gdb and break here
    jmp     $
```

After launching the program, find its process ID and attach GDB: `gdb -p PID`. Then examine all registers with `info registers`.

3.2 Special Registers (RIP, RSP, RFLAGS)

In addition to the general-purpose set, the processor has a handful of special-purpose registers that control execution directly.

3.2.1 RIP — Instruction Pointer

The instruction pointer holds the 64-bit virtual address of the next instruction to be executed. In 64-bit long mode, `RIP` cannot be used as an explicit operand. You cannot write `mov rax, rip`. Instead, you read its value with an `LEA` instruction that uses RIP-relative addressing.

```
; read the address of the next instruction into RAX
lea rax, [rel next_instruction]
next_instruction:
nop
```

`CALL`, `JMP`, `RET`, and conditional branches all modify `RIP` implicitly. The `CALL` instruction pushes the return address (the value of `RIP` after the `CALL`) onto the stack and then sets `RIP` to the target address. `RET` pops the stored address back into `RIP`.

3.2.2 RSP — Stack Pointer

`RSP` always points to the top of the stack. The stack grows downward in memory; `PUSH` decrements `RSP` by 8 and then writes the operand; `POP` does the reverse. In Linux x86-64 code, `RSP` must satisfy one strict constraint for function calls:

- At the exact moment of a `CALL` instruction, `RSP` must be 16-byte aligned. After `CALL` pushes the 8-byte return address, `RSP + 8` will be a multiple of 16.

There is no shadow space requirement, unlike the Windows ABI. However, when calling C library functions, the stack must be aligned as above. The typical prologue in a `main` function called by the C runtime subtracts an appropriate amount to re-align the stack.

Code: listing3-2.asm — Stack alignment demonstration with libc

```
; Assemble: nasm -f elf64 -o listing3-2.o listing3-2.asm
; Link:     gcc -no-pie -o listing3-2 listing3-2.o

bits 64
default rel
```

```
extern puts

section .data
msg  db 'RSP alignment test passed.', 0

section .text
global main
main:
    ; On entry, the stack is misaligned by 8 (the return address).
    ; To re-align before calling puts, subtract an odd multiple of 8.
    sub     rsp, 8

    lea     rdi, [rel msg]      ; first argument: message address
    call    puts

    ; Restore the stack and return.
    add     rsp, 8
    xor     eax, eax            ; return 0
    ret
```

If the `sub rsp, 8` were omitted, the call to `puts` could crash inside the library due to misaligned access to XMM registers. This simple adjustment ensures correctness.

3.2.3 RFLAGS — Status and Control Flags

The `RFLAGS` register is a 64-bit entity, but most of the bits are reserved or system-controlled. The status flags that are most useful in assembly programming are:

- **CF (Bit 0) — Carry Flag.** Set by unsigned arithmetic overflow (carry out of the most significant bit).
- **PF (Bit 2) — Parity Flag.** Set if the low byte of the result has an even number of set bits. Rarely used except in legacy code.
- **AF (Bit 4) — Auxiliary Carry.** Used for BCD arithmetic; not commonly used in modern code.
- **ZF (Bit 6) — Zero Flag.** Set if the result is zero.
- **SF (Bit 7) — Sign Flag.** Set to the most significant bit of the result (i.e., the sign bit in signed arithmetic).
- **OF (Bit 11) — Overflow Flag.** Set if signed arithmetic overflow occurs.
- **DF (Bit 10) — Direction Flag.** Controls the auto-increment (DF=0) or auto-decrement (DF=1) of `RSI` and `RDI` during string instructions. The System V ABI expects DF to be clear (0) when calling a function; the kernel also preserves it across system calls.

Conditional jump instructions check combinations of these flags. For example, `JE` jumps if ZF=1; `JL` jumps if SF != OF.

Code: listing3-3.asm — Flag examination after arithmetic

```
; Assemble: nasm -f elf64 -o listing3-3.o listing3-3.asm
; Link:     ld -o listing3-3 listing3-3.o

bits 64
default rel

section .text
global _start
_start:
    mov     al, 0xFF
    add     al, 1           ; AL = 0, CF=1, ZF=1, SF=0, OF=0
    ; inspect flags in gdb at this point

    mov     eax, 60         ; sys_exit
    xor     edi, edi
    syscall
```

Set a breakpoint in GDB after the `add` instruction and run `info registers eflags`. You will see the flags register with bits CF and ZF set.

Caution

Direction flag must be clear. The System V ABI requires that `DF` be zero on entry to any function. If your code uses string instructions and sets DF=1, you must clear it again before returning. Although the kernel saves and restores RFLAGS across system calls, a function that leaves DF=1 can cause catastrophic failures in the caller if it subsequently uses string operations.

3.3 Segment Registers (Conceptual View)

In the original 16-bit and 32-bit x86, segment registers played a central role in memory addressing. In 64-bit long mode, the memory model is flat and segmentation is essentially bypassed. The six segment registers — `CS`, `DS`, `SS`, `ES`, `FS`, `GS` — still exist, but their base addresses are ignored for most purposes, with two notable exceptions: `FS` and `GS` are used for thread-local storage (TLS).

FS and Thread-Local Storage

On Linux, the `FS` register is set by the kernel to point to a per-thread data structure managed by the threading library (typically glibc's `pthread`). This data block holds the thread control block (TCB) and thread-specific variables. The exact layout is implementation-dependent, but you can read the base address using the `arch_prctl` system call, or directly dereference memory using the `FS` override.

Code: listing3-4.asm — Reading the TLS pointer via FS

```
; Assemble: nasm -f elf64 -o listing3-4.o listing3-4.asm
; Link:     ld -o listing3-4 listing3-4.o
```

```
bits 64
default rel

section .bss
tls_val resq 1

section .text
global _start
_start:
    mov     rax, [fs:0]             ; read first qword of TLS block
    mov     [rel tls_val], rax

    ; infinite loop - attach gdb and inspect tls_val
    jmp     $
```

After running and attaching GDB, examine `tls_val`. The value obtained will be the first 8 bytes of the thread's TLS area – typically a pointer to the TCB itself or a canary. The exact meaning is not required for most applications; this example merely demonstrates that `FS` is usable from user mode and points to private per-thread storage.

Tip

In practice, user-mode assembly programs rarely need to access the TLS directly. High-level languages manage TLS variables with compiler support. If you need per-thread storage, consider using POSIX threads and C data structures, or reserving a slot through the `__thread` keyword in C and computing the offset manually.

3.4 Instruction Cycle (Fetch–Decode–Execute)

Modern `x86-64` processors appear to execute instructions one after another, but inside they operate like a high-speed assembly line. Understanding this pipeline is not essential for writing a simple system call, but it becomes invaluable when you are optimising inner loops or tight arithmetic sequences.

The Classic Pipeline Stages

A simplified model of the processor pipeline consists of five major stages:

1. **Fetch.** The instruction fetch unit reads bytes from the L1 instruction cache at the address currently held in the instruction pointer. The fetched bytes are placed in a queue.
2. **Decode.** The raw bytes are parsed into one or more micro-operations (uops). Register operands and immediate values are extracted, and memory addresses are computed.
3. **Execute.** The uops are dispatched to the available execution units: integer ALU, floating-point unit, load/store unit, or branch prediction unit. Modern CPUs have multiple execution units, so independent uops can begin execution simultaneously.
4. **Memory Access.** If a uop needs to read from or write to the L1 data cache, that operation occurs here. Cache misses cause pipeline stalls that can last hundreds of cycles.

5. **Writeback.** The results are written back to the register file. Register renaming ensures that different iterations of a loop or independent chains do not conflict on temporary registers.

Because the pipeline is deep, a single instruction can take many cycles from fetch to retire, but in the steady state the processor can retire multiple instructions per clock cycle.

Data Dependencies and Instruction Level Parallelism

A data dependency exists when one instruction consumes a register that a previous instruction produces. The second instruction cannot begin execution until the first has written back its result, creating a chain that limits throughput.

Consider a purely sequential chain:

```
add rax, rcx
sub rbx, rax    ; depends on ADD
shl rbx, 3      ; depends on SUB
```

The processor cannot execute the `SUB` until `ADD` finishes, and it cannot execute `SHL` until `SUB` finishes. The latency of each instruction (typically 1–3 cycles for simple integer ops) becomes the bottleneck.

If independent instructions are interleaved, the pipeline can find parallel work:

```
add rax, rcx     ; start chain 1
mov r8, [rdx]    ; independent load
and r9, 0x0FF    ; independent operation
sub rbx, rax     ; chain 1 (now RAX is available)
```

Modern out-of-order cores will automatically detect independence and execute the `mov` and `and` while waiting for the `add` to finish, then retire them all in program order.

Branch Prediction

Conditional jumps disrupt the sequential flow. Rather than wait for the condition to resolve, the processor guesses the outcome using a branch predictor. It speculatively fetches and executes instructions down the predicted path. If the guess is correct, execution proceeds at full speed. If incorrect (a *branch misprediction*), the pipeline is flushed, and the correct path starts from scratch, costing typically 15–20 cycles on modern CPUs.

Writing predictable branches becomes important in hot loops. For example, a loop that counts down to zero with a `JNZ` is highly predictable because the branch is taken on every iteration except the last.

Observing Pipeline Effects with a Benchmark

The following small program runs a simple arithmetic chain repeatedly. It is not a real benchmark but illustrates how dependencies affect execution by allowing you to measure the difference between a tight dependency chain and an interleaved version with a cycle-accurate timer or just observing wall-clock time.

Code: listing3-5.asm — Dependency chain illustration

```
; Assemble: nasm -f elf64 -o listing3-5.o listing3-5.asm
; Link:     ld -o listing3-5 listing3-5.o

bits 64
default rel

section .text
global _start
_start:
    mov     ecx, 10000000           ; iteration count
    xor     eax, eax                ; accumulator
    mov     r8,  1                  ; constant for interleaving example

.loop:
    ; dependency chain version (uncomment to test)
    ; add eax, 1
    ; add eax, 1
    ; add eax, 1
    ; add eax, 1

    ; independent version (leave as is for demonstration)
    add eax, 1
    add r8,  1
    add eax, 1
    add r8,  1

    sub ecx, 1
    jnz .loop

    ; exit with the value of RAX as exit code
    mov     rdi, rax
    mov     eax, 60          ; sys_exit
    syscall
```

The interleaved version uses two independent chains (`EAX` and `R8`), allowing the processor to issue multiple adds per cycle and hide latency. The dependency chain version is sequential; each `add` waits for the previous one. On a modern `x86-64` core, the interleaved loop can run significantly faster. To measure, you could use the `RDTSC` instruction or simply time the program with `time ./listing3-5`. The exit code returned is the final value of RAX.

Important

Pipeline awareness is a performance tool, not a correctness requirement. For most system-call-driven programs, the overhead of kernel entries dwarfs any pipeline effect. But when you are writing a compute-intensive routine, such as an encryption kernel or a multimedia filter, understanding the pipeline can be the difference between a routine that runs at full speed and one that leaves half the processor's capability unused.

Bringing the Model Together

The combination of registers, flags, the stack, and the fetch-decode-execute pipeline forms the execution model that every line of assembly code implicitly uses. In the next chapter, we will formalise the System V AMD64 calling convention and begin writing functions that can call each other and the Linux C library with complete reliability.

4

Memory Layout and Addressing Modes

In This Chapter

Memory is the second essential resource an assembly programmer must master, after the register file. This chapter describes the virtual memory map of a Linux 64-bit process on Fedora 43, the roles of the stack, heap, and data sections, the powerful addressing modes of the `x86-64` architecture, and how to compute effective addresses. Pointer manipulation in assembly is entirely explicit — there is no dereference operator separate from address calculation — and the examples in this chapter will build a rock-solid intuition for the way the processor sees memory. All information is derived from the Intel and AMD architecture manuals, the System V AMD64 ABI, and the Linux man-pages. Every code listing has been tested with NASM 2.16 on Fedora 43.

4.1 Process Memory Layout

When a 64-bit ELF executable loads, the kernel constructs a private virtual address space for the process. Linux on x86-64 provides a flat 64-bit virtual address space where user mode occupies the lower 128 TB and kernel space the upper 128 TB. The layout of a typical user-mode process follows a predictable pattern, though address-space layout randomisation (ASLR) randomises the base of the stack, heap, and shared libraries.

Key Regions in User Space

- **Executable image (ELF).** The binary is loaded at a base address (commonly `0x400000` for non-PIE executables; with PIE it is randomised). It contains the `.text` section (code), `.data` (initialised read-write data), `.rodata` (read-only data), and `.bss` (zero-initialised static data, merged into the data segment at load time).

- **Loaded shared libraries.** System libraries such as `libc.so.6` and `ld-linux-x86-64.so.2` are mapped at addresses typically in the range `0x7f...`.
- **Stack.** A single call stack grows downward from a high address (the top of the user space). The default stack size is 8 MiB, but it can be changed via `ulimit` or `setrlimit`.
- **Heap.** The dynamic memory area lies just above the data segment. It is managed by the C library via `malloc` and `free`, which expand the program break using the `brk()` system call or map anonymous memory with `mmap()`.
- **Thread-Local Storage (TLS).** A small per-thread area accessible via the `FS` segment (in 64-bit mode, the kernel sets `FS` to point to a per-thread structure used by glibc).
- **Auxiliary vector and environment.** Placed on the initial stack.

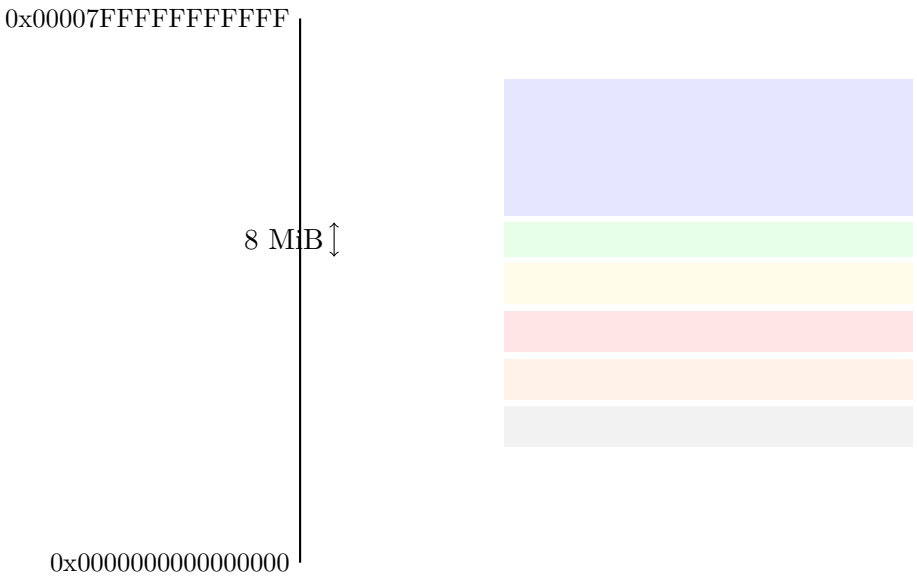


Figure 4.1: Simplified virtual address space layout for a 64-bit Linux process (non-PIE).

Section Permissions

NASM allows you to place code and data in arbitrary named sections, but the linker and the kernel enforce page-level protections via the ELF program headers. Typically:

- `.text` — read and execute.
- `.data` — read and write.
- `.rodata` — read only.
- `.bss` — read and write, zero-initialised at load time.

Attempting to write into a code section will cause a segmentation fault.

Inspecting the Memory Layout

The following program captures the virtual addresses of several important landmarks and stores them in memory. It then prints them using the C library `printf`, so we link against libc.

Code: listing4-1.asm — Capturing process memory landmarks

```
; Assemble: nasm -f elf64 -o listing4-1.o listing4-1.asm
; Link:     gcc -no-pie -o listing4-1 listing4-1.o

bits 64
default rel

extern printf
extern exit

section .data
marker  db 'MEMORY LAYOUT DEMO',0
fmt     db 'text_addr: 0x%llX',10
        db 'data_addr: 0x%llX',10
        db 'bss_addr:  0x%llX',10
        db 'stack_addr: 0x%llX',10,0

section .bss
text_addr   resq 1
data_addr   resq 1
bss_addr    resq 1
stack_addr  resq 1

section .text
global main
main:
    ; preserve stack alignment (entry to main is 8-byte misaligned)
    sub     rsp, 8

    ; capture address of .text
    lea     rax, [rel main]
    mov     [rel text_addr], rax

    ; capture address of .data
    lea     rax, [rel marker]
    mov     [rel data_addr], rax

    ; capture address of a .bss variable
    lea     rax, [rel bss_addr]
    mov     [rel bss_addr], rax

    ; store current stack pointer
    mov     [rel stack_addr], rsp

    ; now print them
    lea     rdi, [rel fmt]
    mov     rsi, [rel text_addr]
    mov     rdx, [rel data_addr]
    mov     rcx, [rel bss_addr]
```

```
    mov     r8,  [rel stack_addr]
    xor     eax, eax          ; no floating-point args
    call    printf

    add     rsp, 8
    xor     edi, edi
    call    exit              ; won't return
```

Run the program; you will see that `text_addr` is in the low `0x400...` range, `data_addr` slightly higher, and `stack_addr` near the top of user space.

4.2 Stack vs Heap vs Data Segment

Assembly gives the programmer complete control over where data lives. However, each storage area has a distinct purpose and lifetime.

4.2.1 Stack

The stack is a region of memory managed by the processor for function call linkage, local variables, and temporary storage. It operates as a last-in-first-out structure. In Linux x86-64, the stack frame of a function typically includes the return address, saved non-volatile registers, and local variables. There is no mandatory shadow space; the callee merely expects the stack to be 16-byte aligned at the point of a `CALL`.

Stack storage is fast because the stack is almost always in the L1 cache. However, its size is limited (default 8 MiB per thread), and once a function returns, its local variables become invalid.

Code: listing4-2.asm — Stack-allocated local variables

```
; Assemble: nasm -f elf64 -o listing4-2.o listing4-2.asm
; Link:     gcc -no-pie -o listing4-2 listing4-2.o

bits 64
default rel

extern exit

section .text
global main
main:
    sub     rsp, 16           ; align stack + make room for locals

    ; store two quad-word local variables on the stack
    mov     qword [rsp],     0xDEADBEEFCAFEBABE
    mov     qword [rsp+8],   0x1234567890ABCDEF

    ; ... use the variables ...

    ; cleanup and return
    add     rsp, 16
    xor     edi, edi
```

```
    call    exit
```

The quad-words at `[rsp]` and `[rsp+8]` are local variables. They exist only while `main` is active.

Caution

Guaranteeing stack alignment. After the `sub rsp, 8` to align the stack upon entry to `main`, we further subtracted 16 to make room for two variables; this maintains 16-byte alignment because 8+16 = 24, which is an odd multiple of 8, so after the return address push the stack is re-aligned.

4.2.2 Heap

The heap is a pool of memory managed by the C library (or the kernel) that persists until explicitly freed. It is suitable for large or variable-size data structures, and for data that must outlive the function that created it.

The standard C library provides `malloc` and `free`. A common pattern is to allocate a block, use it, and then release it.

Code: listing4-3.asm — Allocating memory with malloc

```
; Assemble: nasm -f elf64 -o listing4-3.o listing4-3.asm
; Link:     gcc -no-pie -o listing4-3 listing4-3.o

bits 64
default rel

extern malloc
extern free
extern exit

section .bss
block_ptr resq 1

section .text
global main
main:
    sub     rsp, 8

    ; allocate 1024 bytes
    mov     rdi, 1024
    call    malloc
    mov     [rel block_ptr], rax    ; save pointer

    ; use the block: store a string into it
    lea     rsi, [rel msg]
    mov     rdi, [rel block_ptr]
    mov     rcx, msg_len
    cld
    rep movsb
```

```
    ; free the block
    mov     rdi, [rel block_ptr]
    call    free

    add     rsp, 8
    xor     edi, edi
    call    exit

section .data
msg      db 'Heap-allocated string',0
msg_len  equ $ - msg
```

4.2.3 Data Segments

Global and static variables reside in sections such as `.data` and `.bss`. The `.data` section contains initialised data and is stored in the executable file. The `.bss` section holds zero-initialised variables and does not consume file space. Both are mapped with read-write permissions.

Data segment variables are accessible from any code location, and their addresses are typically RIP-relative to support position-independent executables (PIE).

```
default rel
section .data
my_dword dd 0x11223344

section .text
...
mov eax, [rel my_dword]          ; load global variable
```

4.3 Addressing Modes in x86-64

The x86-64 architecture provides a rich set of addressing modes that allow memory operands to be calculated from combinations of base register, index register, scale factor, and displacement. The general form in NASM syntax is:

```
[base + index*scale + displacement]
```

where each component is optional. The processor computes the effective address (EA) as the sum of the base (if provided), the index times scale (scale = 1,2,4,8), and a signed constant displacement.

Common Addressing Forms

- **Base only**: `[rcx]`
- **Base + displacement**: `[rcx + 8]`, `[rsp + 32]`
- **Base + index**: `[rcx + rdx]`
- **Base + index * scale**: `[rcx + rdx*4]`
- **Base + index * scale + displacement**: `[rcx + rdx*8 + 64]`

- **RIP-relative**: `[rel label]` or `[rip + displacement]` (used with `default rel` or explicit `rel` keyword)

The NASM assembler uses the same syntax for all. The following snippet loads a value from an array of 64-bit integers using indexed addressing:

```
lea rcx, [rel array]            ; base of array
mov rdx, 3                      ; index
mov rax, [rcx + rdx*8]          ; load array[3]
```

4.4 Effective Address Calculation

The effective address (EA) is a 64-bit value computed by the address generation unit before the memory access takes place. The calculation follows the formula:

$$\text{EA} = (\text{Base\_Register if Base else } 0) + (\text{Index\_Register} \cdot \text{Scale if Index else } 0) + \text{Displacement}$$

The displacement is sign-extended to 64 bits. The result is then translated by the MMU from a virtual address to a physical address.

Encoding Details

The ModR/M and SIB bytes encode the addressing mode. The base and index registers may be any of the 16 general-purpose registers. In 64-bit mode, absolute addresses are rare; the assembler uses RIP-relative addressing for global data by default.

NASM can produce a listing file showing the actual encodings. For example, the instruction `mov rax, [rcx + rdx*8 + 16]` in 64-bit mode may be encoded as:

```
48 8B 44 D1 10   mov rax,[rcx+rdx*8+0x10]
```

A Worked Example

The following program calculates the sum of an array with different addressing modes and exits with the result as the exit code, so you can check it with `echo $?`.

Code: listing4-4.asm — Effective address examples

```
; Assemble: nasm -f elf64 -o listing4-4.o listing4-4.asm
; Link:     ld -o listing4-4 listing4-4.o

bits 64
default rel

section .data
array  dq  100, 200, 300, 400, 500
count  equ ($ - array) / 8

section .text
global _start
```

```
_start:
    ; Method 1: base + index*scale
    xor     eax, eax
    xor     ecx, ecx
    lea     rdx, [rel array]
.loop1:
    add     rax, [rdx + rcx*8]      ; indexed addressing
    inc     ecx
    cmp     ecx, count
    jb      .loop1
    ; Method 1 sum in RAX

    ; Method 2: pointer increment (we reuse RAX, so store method1 sum)
    mov     r8, rax                 ; save method1 sum
    lea     rdx, [rel array]
    xor     eax, eax
    mov     ecx, count
.loop2:
    add     rax, [rdx]              ; base only
    add     rdx, 8
    dec     ecx
    jnz     .loop2
    ; RAX now method2 sum; should equal r8

    ; exit with the sum (1500 = 0x5DC) as exit code
    mov     rdi, rax
    mov     eax, 60                  ; SYS_exit
    syscall
```

After running, `echo $?` will print `1500`.

Tip

Indexed addressing frees you from having to update the pointer separately inside a loop, which can reduce register pressure. However, pointer increment requires no extra register for an index, and the `[rdx]` form uses a smaller encoding.

4.5 Pointer Concepts in Assembly

A pointer is simply an address. In assembly, there is no type system to distinguish a pointer from an integer; a 64-bit value in a register can be used as a number or as an address, depending on how you apply it.

4.5.1 Dereferencing

Dereferencing means using a register as a memory operand. The instruction `mov rax, [rcx]` reads 8 bytes from the address held in `RCX`. The brackets indicate the dereference. To load the address itself, you use `LEA` (Load Effective Address):

```
lea rax, [rcx]       ; RAX = RCX (does not read memory)
mov rax, [rcx]       ; RAX = value at address RCX
```

4.5.2 LEA vs MOV

LEA is a powerful instruction that computes the effective address of its source operand and stores that address in the destination register. It does not touch memory, but it can perform arithmetic on up to three operands in a single instruction. This makes LEA useful for fast integer arithmetic even when no actual memory access is intended.

Code: listing4-5.asm — LEA as a multi-addition tool

```
; Assemble: nasm -f elf64 -o listing4-5.o listing4-5.asm
; Link:     ld -o listing4-5 listing4-5.o

bits 64
default rel

section .text
global _start
_start:
    mov     rcx, 5
    mov     rdx, 10

    ; RAX = RCX + RDX*2 + 8   (all in one LEA)
    lea     rax, [rcx + rdx*2 + 8]    ; RAX = 5 + 20 + 8 = 33

    ; exit with 33 as exit code
    mov     rdi, rax
    mov     eax, 60
    syscall
```

4.5.3 Pointer Chains

Assembly pointers can form linked data structures. The following snippet traverses a simple linked list of nodes, where each node contains a 64-bit value and a pointer to the next node.

```
; Assume a linked list node:
; struct Node { dq value; dq next; };
; current pointer in RAX

.again:
    test    rax, rax
    jz      .done
    mov     rcx, [rax]          ; load value
    ; ... process value ...
    mov     rax, [rax + 8]      ; load next pointer
    jmp     .again
.done:
```

Because the pointer ‘next‘ is at offset 8, we use `[rax + 8]` to dereference it.

Warning

Null pointer check. Linux does not map the zero page (addresses below `0x10000` typically). Dereferencing a zero pointer will cause a segmentation fault. Always test pointers before use.

Putting It All Together

Memory in Fedora 43 is a vast, flat space. The stack is perfect for temporaries; the heap is for dynamic allocations; data segments hold static data. The addressing modes let you access any memory cell with flexible combinations of base, index, scale, and displacement. Mastering these concepts is essential because every subsequent chapter — on the system-call interface, on SIMD, on data structures — will use them constantly. The next chapter will formalize the stack's role in function calls and the System V AMD64 calling convention.

5

Stack Architecture Fundamentals

In This Chapter

The stack is the central mechanism for function calls, local variable storage, and control flow in `x86-64` programs. This chapter explains the physical and logical structure of the stack, the inner workings of PUSH and POP, the construction of stack frames, the direction of stack growth, and the detailed behaviour of the function call sequence as mandated by the System V AMD64 ABI on Linux. Every concept is drawn from the Intel® 64 and IA-32 Architectures Software Developer's Manual, the AMD64 Architecture Programmer's Manual, and the System V ABI. Each code example has been assembled and tested with NASM 2.16 on Fedora 43.

5.1 Stack Structure

The stack is a contiguous region of virtual memory set aside for each thread. In Linux, the default stack size for the main thread is 8 MiB, which can be adjusted with `ulimit -s` or `setrlimit`. The stack is used for:

- Storing the return address when a `CALL` instruction is executed.
- Passing function arguments beyond the first six.
- Holding local variables that do not fit in registers.
- Saving and restoring non-volatile registers.
- Temporary scratch space for calculations.

The processor accesses the stack via the **Stack Pointer** register, `RSP`. The `RSP` always points to the top of the stack, which is the lowest occupied address. The stack grows toward lower addresses;

thus, allocating space on the stack is done by subtracting from RSP, and freeing is done by adding to RSP.

Note

The memory region below the current stack pointer (lower addresses) is free and can be used after allocation. The region above RSP (higher addresses) is already in use and should not be modified without care.

Visualising the Stack

Imagine a block of memory with addresses increasing upward. Initially, RSP points to the highest address of the stack, often near 0x7FFF... As data is pushed, RSP decrements, moving down.

```
High address  +------------------+
              |   ... used ...   |
              |  previous data   |
              |  previous data   |
    RSP -->   +------------------+  <- top of stack (lowest address in use)
              |  free space      |
Low address   +------------------+
```

5.2 PUSH and POP Internals

The instructions PUSH and POP are the most direct way to manipulate the stack. They implicitly use RSP and perform a move operation combined with a pointer adjustment.

5.2.1 PUSH

In 64-bit mode, PUSH reg/mem/imm performs the following sequence:

1. RSP = RSP - 8
2. The 64-bit value is written to the memory location pointed to by the new RSP.

If the source is a 32-bit immediate, it is sign-extended to 64 bits before being pushed. If it is a 16-bit or 8-bit register, PUSH is not allowed; always use the full 64-bit register or a memory operand.

Code: listing5-1.asm — Push example; examine stack in GDB

```
; Assemble: nasm -f elf64 -o listing5-1.o listing5-1.asm
; Link:     ld -o listing5-1 listing5-1.o

bits 64
default rel

section .text
global _start
_start:
    mov     rax, 0xDEADBEEFCAFEBABE
    push    rax
```

```
    push    0x1234567890ABCDEF

    ; examine [rsp] and [rsp+8] in GDB
    ; infinite loop so we can attach
    jmp     $
```

After the two pushes, RSP has been reduced by 16. The value at [rsp] is the last pushed value; at [rsp+8] is the first pushed value. This demonstrates the LIFO nature of the stack.

5.2.2 POP

POP reg/mem reverses the process:

1. Read the 64-bit value from memory at the current RSP.
2. RSP = RSP + 8.

The destination is written with the stack value, and RSP moves back toward higher addresses. It is a common mistake to forget to balance pushes with pops, leading to a corrupted stack.

Code: listing5-2.asm — Push/Pop balance verification

```
; Assemble: nasm -f elf64 -o listing5-2.o listing5-2.asm
; Link:     ld -o listing5-2 listing5-2.o

bits 64
default rel

section .text
global _start
_start:
    mov     rax, 0xAAAA
    push    rax
    push    rbx
    pop     rbx
    pop     rax             ; RAX = 0xAAAA again
    ; exit with RAX low byte
    mov     rdi, rax
    mov     eax, 60
    syscall
```

After this sequence, RAX is restored to its original value, and RSP returns to the same position it had before the pushes. The order of pops must be the reverse of pushes to restore each register correctly.

Warning

Mismatched PUSH/POP. If you push a value and forget to pop it, RSP will be too low on return, causing the RET instruction to load an incorrect return address and crash the program. Always maintain stack balance within a block.

5.3 Stack Frames

A stack frame is a contiguous block on the stack that belongs to a single function invocation. It contains the return address, saved registers, local variables, and sometimes arguments for called functions. In the System V AMD64 calling convention, the typical layout of a stack frame (seen from high addresses to low) is:

1. **Caller's stack arguments** (if the callee has more than six arguments, the caller pushes them right-to-left; they are above the return address from the callee's perspective).
2. **Return address** (pushed by `CALL`).
3. **Saved non-volatile registers** (if any are used by the callee, e.g., `RBX`, `RBP`, `R12`–`R15`).
4. **Local variables** (allocated by subtracting from `RSP`).

The standard prologue of a function that uses a frame pointer is:

```
function:
    push    rbp                 ; save old frame pointer
    mov     rbp, rsp            ; set up new frame pointer
    sub     rsp, N              ; allocate locals (N must keep stack 16-byte aligned)
    ; ... body ...
    mov     rsp, rbp            ; restore stack pointer
    pop     rbp                 ; restore old frame pointer
    ret
```

Using `RBP` as a stable base can simplify accessing parameters and locals, especially in functions that change `RSP` dynamically.

Frame with Locals and a Call to puts

The following function, `say_hello`, takes no arguments, allocates local storage, calls the C library `puts`, and returns. It demonstrates a full frame layout.

Code: listing5-3.asm — Stack frame with locals and a C call

```
; Assemble: nasm -f elf64 -o listing5-3.o listing5-3.asm
; Link:     gcc -no-pie -o listing5-3 listing5-3.o

bits 64
default rel

extern puts
extern exit

section .data
msg db 'Hello from a framed function!',0

section .text
global main
main:
    push    rbp
    mov     rbp, rsp
    sub     rsp, 16          ; keep stack alignment; 2 locals (16 bytes)
```

```
    ; store a local variable
    mov     qword [rbp-8], 0x12345678

    ; call puts
    lea     rdi, [rel msg]
    call    puts

    ; load the local and exit with its low byte as code
    mov     rax, [rbp-8]
    mov     rdi, rax
    call    exit            ; will not return
```

The local variable lives at `[rbp-8]`. The prologue sets `RBP` to the entry stack pointer minus the 8 bytes of the pushed `RBP`, and then subtracts 16 to align the stack for the call and create room for locals. Because `puts` expects the stack to be 16-byte aligned at the point of the call, and after `push rbp` the stack is 16-byte aligned, the `sub rsp, 16` preserves that alignment (since 16 is a multiple of 16).

Tip

On function entry after a `CALL`, the stack is misaligned by 8 (the return address). The `push rbp` subtracts another 8, making it 16-byte aligned again. Then you can subtract a multiple of 16 to allocate locals.

5.4 Stack Growth Direction

The `x86-64` stack grows downward in memory, from high addresses to low addresses. This means that the "top" of the stack is actually the lowest occupied address. The `RSP` register points to the last item pushed.

We can demonstrate this growth direction with a simple program that records the addresses of successive stack allocations and prints the difference.

Code: listing5-4.asm — Demonstrating stack growth direction

```
; Assemble: nasm -f elf64 -o listing5-4.o listing5-4.asm
; Link:     gcc -no-pie -o listing5-4 listing5-4.o

bits 64
default rel

extern printf
extern exit

section .data
fmt db 'addr1: 0x%llX, addr2: 0x%llX, difference: %lld',10,0

section .bss
addr1 resq 1
addr2 resq 1
```

```
section .text
global main
main:
    push    rbp
    mov     rbp, rsp

    ; allocate first variable on stack
    sub     rsp, 16
    lea     rax, [rbp-8]    ; first local
    mov     [rel addr1], rax

    ; allocate second variable (another 8 bytes doum)
    sub     rsp, 8
    lea     rax, [rbp-16]
    mov     [rel addr2], rax

    ; prepare printf arguments
    lea     rdi, [rel fmt]
    mov     rsi, [rel addr1]
    mov     rdx, [rel addr2]
    mov     rcx, rsi
    sub     rcx, rdx        ; addr1 - addr2
    xor     eax, eax
    call    printf

    ; exit
    xor     edi, edi
    call    exit
```

You will see that `addr1` is greater than `addr2`, and the difference is positive (8 bytes), confirming downward growth.

Implications of Downward Growth

Because the stack grows to lower addresses, a buffer overflow on a local array will overwrite the return address stored at a higher address. This is the classical stack-smashing attack. Modern Linux systems include stack canaries and address-space layout randomisation (ASLR) to mitigate such attacks, but understanding the layout is essential for safe coding.

5.5 Function Call Stack Behavior

When a function calls another function, a precise sequence of steps unfolds, governed by the hardware and the System V AMD64 ABI.

5.5.1 CALL Instruction Details

`CALL <target>` performs two operations atomically:

1. `PUSH` the 64-bit address of the instruction immediately following the `CALL` (the return address) onto the stack.

2. JMP to the target address.

The return address is thus the value of RIP that the callee will return to when it executes RET. The caller is responsible for having the stack 16-byte aligned before the CALL, and for setting up arguments in registers and on the stack as required.

5.5.2 RET Instruction

RET (or RET imm) pops the return address from the stack and loads it into RIP, effectively returning to the caller. RET imm also adds imm to RSP after the pop; however, in the System V ABI the caller cleans up the stack after a call, so RET without an immediate is the norm.

5.5.3 System V AMD64 Call Sequence

Before a call, the caller must:

- Place the first six integer/pointer arguments in RDI, RSI, RDX, RCX, R8, R9 (in that order).
- Push any remaining arguments (7th and beyond) onto the stack from right to left.
- Ensure that RSP is 16-byte aligned at the point of the CALL instruction (i.e., RSP is a multiple of 16 before the CALL pushes the return address; after the push, RSP + 8 is a multiple of 16).

Inside the callee, the stack frame is built as described. The callee may freely use the registers RAX, RCX, RDX, RSI, RDI, R8–R11 without saving them, but must preserve RBX, RBP, R12–R15 if it modifies them. On return, the caller is responsible for removing any stack-passed arguments (the callee does not adjust the stack other than the return address pop).

Code: listing5-5.asm — Demonstrating call sequence

```
; Assemble: nasm -f elf64 -o listing5-5.o listing5-5.asm
; Link:     gcc -no-pie -o listing5-5 listing5-5.o

bits 64
default rel

extern printf
extern exit

section .data
fmt db 'Return address captured: 0x%llX',10,0

section .text
global main

; A function that captures its own return address (via RBP)
capture:
    push    rbp
    mov     rbp, rsp
    ; the return address is at [rbp+8]
    mov     rax, [rbp+8]
    ; print it using printf
    lea     rdi, [rel fmt]
```

```
    mov     rsi, rax
    xor     eax, eax
    call    printf
    mov     rsp, rbp
    pop     rbp
    ret

main:
    push    rbp
    mov     rbp, rsp
    ; align stack for callee (it's already aligned after push rbp)
    call    capture
    ; on return, rax is clobbered, but we don't care
    xor     edi, edi
    call    exit
```

The `capture` function reads the return address from `[rbp+8]` (just above the saved frame pointer) and prints it. Running the program you will see an address just after the `call capture` instruction.

5.5.4 Stack Alignment in Depth

The 16-byte alignment requirement is critical. If the stack is misaligned, XMM-register operations inside C library functions may fault. The ABI specifies that before a `CALL`, `RSP` must be 16-byte aligned. So upon entry to a function, `RSP + 8` is 16-byte aligned. To maintain this alignment when allocating space for locals or arguments, the total amount subtracted from `RSP` (after any pushes) must be an odd multiple of 8 if the function makes calls.

For example, after the standard prologue:

```
push rbp            ; RSP becomes aligned (entry RSP was misaligned by 8,
                    ; push another 8 makes it aligned)
mov rbp, rsp
sub rsp, 16         ; subtract a multiple of 16 -> still aligned
```

Warning

Alignment failure is silent until it's not. A misaligned stack may work for many calls until a function uses an aligned SSE load or store. The crash will occur inside library code with little clue about the cause. Always verify alignment in your prologue.

Putting It All Together

The stack is the backbone of function calling. PUSH and POP move data, CALL and RET manage control flow, and the stack frame provides a disciplined way to organize local state. The System V AMD64 ABI demands 16-byte alignment and the use of registers for the first six arguments, with the caller responsible for stack cleanup. The next chapter will formalize the complete calling convention and show how to pass arguments, return values, and interface with the C library from assembly.

6

Instruction Execution Cycle

In This Chapter

Every instruction your program issues travels through an intricate pipeline of hardware stages before its result becomes visible. This chapter dissects the lifecycle of an `x86-64` instruction: the fetch, decode, execution, memory access, and writeback steps that constitute the classic five-stage pipeline. You will learn the concept of micro-operations (uops), the way modern superscalar processors achieve instruction-level parallelism, and how branch prediction and speculative execution allow the pipeline to flow smoothly. Finally, we examine how these micro-architectural features directly affect the performance of assembly code running under Linux on Fedora 43, with concrete NASM examples that you can measure on your own machine. All descriptions are grounded in the Intel® 64 and IA-32 Architectures Optimization Reference Manual, the AMD64 Architecture Programmer's Manual, and the observable behaviour of current `x86-64` hardware.

6.1 Instruction Lifecycle

From the programmer's perspective, an instruction like `add rax, rcx` executes instantaneously. In reality, it travels through a defined sequence of hardware stages before retiring. The lifecycle of a single instruction can be broken down into the following phases, which correspond to the pipeline stages of a typical modern `x86-64` core.

1. **Fetch.** The instruction fetch unit reads raw bytes from the L1 instruction cache using the address in the instruction pointer (`RIP`). The bytes are placed into a fetch queue. On a cache miss, the fetch unit requests the cache line from the L2 cache or main memory.

2. **Decode.** The fetched bytes are parsed into one or more *micro-operations* (uops). Complex instructions, such as `rep movsb`, are expanded into sequences of uops. Register operands are renamed to avoid false dependencies, and memory operands are prepared for address

generation.

3. **Execute.** The uops are dispatched to the execution ports. The processor contains multiple execution units: integer ALUs, floating-point pipelines, load/store units, and branch units. Independent uops can begin execution in the same clock cycle on different ports.

4. **Memory Access.** If the instruction reads from or writes to memory, the data cache is accessed. Store operations are buffered in a store queue until the instruction is ready to retire.

5. **Writeback / Retire.** The results are written back to the renamed register file. Once an instruction has completed and its uops have been executed, the instruction is retired in program order. The retirement unit updates the architectural register state, making the results visible to subsequent instructions.

A single instruction can spend many clock cycles within this pipeline, but because the stages are overlapped, the processor can sustain a throughput of several instructions per cycle. To visualise the lifecycle, consider the following simple NASM snippet:

```
mov rax, [rel value]    ; load from memory
add rax, 1              ; increment
mov [rel result], rax   ; store to memory
```

When the first `mov` is being fetched, the decode unit is idle. In the next cycle, the `mov` enters decode while the `add` is fetched. This pipelined overlap is what keeps the execution units fed.

Tip

You can observe the pipeline in action with a debugger by stepping instruction by instruction and watching `RIP` advance, but the actual micro-architectural state is invisible. To truly understand pipeline effects, use performance counter tools like `perf` (the Linux profiling tool) or Intel VTune on Fedora.

6.2 Micro-operations Concept

The `x86-64` instruction set is variable-length and contains many complex instructions. For efficient pipelining, the processor translates each macro-instruction into one or more *micro-operations* (uops). A uop is a simple, RISC-like operation that can be processed by one of the execution ports. For example:

- `add rax, rcx` decodes into a single uop: integer ALU operation.
- `add rax, [rcx]` decodes into two uops: one memory load and one ALU add.
- `push rax` decodes into a store-address and store-data uop.
- `rep movsb` decodes into a variable number of uops depending on the count and alignment.

The `x86-64` front-end typically has multiple decoders. On a modern Intel core, there are four decoders: three simple decoders that handle instructions that map to a single uop, and one complex decoder that can handle up to four uops per cycle. Thus, most arithmetic and register-to-register moves can be decoded in a single cycle.

The uop model explains why some sequences of instructions run faster than others, even if they contain the same number of macro-instructions. Consider two ways to clear a register:

```
mov rax, 0      ; 1 uop (mov immediate, zero)
xor eax, eax    ; 1 uop, zero-extends, breaks dependency on RAX
```

`xor eax, eax` is almost always preferred because it is recognized as a zeroing idiom and executes with zero latency and no need for an execution port on many CPUs. The `mov rax, 0` also works but may require an ALU port and a larger encoding.

Micro-Operation Fusion

Certain adjacent uops can be fused into a single uop during decoding, effectively increasing throughput. Common fused uops include compare-and-branch patterns. In NASM, the sequence `cmp rax, rcx` followed by `je label` can be fused on many processors, reducing the uop count by one.

6.3 CPU Pipelining Overview

Pipelining allows a processor to work on multiple instructions simultaneously, much like an assembly line. The classic five-stage pipeline (fetch, decode, execute, memory, writeback) is a conceptual model; real `x86-64` cores have pipelines that are deeper than 14 stages, with many sub-stages and out-of-order execution capabilities.

Superscalar Execution

A superscalar processor can issue multiple uops in a single clock cycle. For example, a processor with four integer ALU ports can execute up to four integer additions simultaneously, provided the inputs are independent and the uops are ready. This is why interleaving independent operations can yield higher throughput than a single dependency chain.

The following NASM loop contains a dependency chain that prevents parallel execution:

```
.loop:
    add rax, 1        ; RAX depends on RAX from previous iteration
    add rax, 1        ; same
    add rax, 1
    add rax, 1
    sub ecx, 1
    jnz .loop
```

Each `add` must wait for the previous one. Now compare an interleaved version:

```
.loop:
    add rax, 1        ; chain 1
    add rbx, 1        ; chain 2, independent of RAX
    add rcx, 1        ; chain 3
    add rdx, 1        ; chain 4
    sub ecx, 1
    jnz .loop
```

Because the four `add` instructions operate on different registers, the processor can issue all four in the same cycle, achieving a throughput near four additions per cycle, limited only by decoder and retirement bandwidth.

Out-of-Order Execution

Modern x86-64 cores execute uops out of order while retiring instructions in order. The hardware uses a *reorder buffer* to hold uops until their inputs are ready. If a load misses the data cache, the processor can continue executing subsequent independent uops, hiding the latency. This makes it difficult to predict exact instruction timing, but also means that carefully written code with sufficient independent work can approach the theoretical peak throughput.

6.4 Branch Behavior

Branches — conditional jumps — are the natural enemy of pipelining. When the fetch unit encounters a conditional jump, it does not know which path to take until the condition is resolved, which may be several stages later. Rather than stall, the processor predicts the branch direction and speculatively fetches instructions along the predicted path. If the prediction is correct, execution continues seamlessly. If incorrect, the pipeline must be flushed, and the correct path must be fetched from scratch, incurring a *misprediction penalty* of typically 15–25 clock cycles on current processors.

Branch Predictor Internals

The branch predictor uses a history of branch addresses and outcomes to make predictions. It maintains tables such as the Branch Target Buffer (BTB) and pattern history tables. Loops with a regular pattern are predicted accurately after a few iterations. Indirect jumps and returns use a separate return address stack (RAS) predictor.

From the assembly programmer's perspective, the most effective way to improve branch predictability is to keep branching patterns consistent and to use conditional moves (`cmovcc`) instead of branches when the sequence is short and unpredictable.

Demonstrating Misprediction Cost

We can write a NASM program that compares the runtime of a predictable branch versus a random pattern. The example uses `rdtsc` to measure cycle counts and `printf` to output the results.

Code: listing6-1.asm — Branch prediction penalty measurement

```
; Assemble: nasm -f elf64 -o listing6-1.o listing6-1.asm
; Link:     gcc -no-pie -o listing6-1 listing6-1.o

bits 64
default rel

extern printf
extern exit

section .data
predictable_msg db 'Predictable branch cycles: ',0
random_msg      db 'Random branch cycles: ',0
fmt_hex         db '%llX',10,0
```

```
section .text
global main

main:
    push    rbp
    mov     rbp, rsp
    sub     rsp, 16                 ; align stack + save measured time

    ; ----- Predictable test (always taken) -----
    rdtsc                           ; EDX:EAX = timestamp counter
    shl     rdx, 32
    or      rax, rdx
    mov     [rsp], rax              ; start time

    xor     ecx, ecx
.predict_loop:
    inc     ecx
    cmp     ecx, 1000000
    jb      .predict_loop           ; always taken until end

    rdtsc
    shl     rdx, 32
    or      rax, rdx
    sub     rax, [rsp]              ; elapsed ticks
    mov     [rsp+8], rax            ; save predictable cycles

    ; print "Predictable branch cycles: "
    lea     rdi, [rel predictable_msg]
    xor     eax, eax
    call    printf

    ; print hex value of cycles
    lea     rdi, [rel fmt_hex]
    mov     rsi, [rsp+8]
    xor     eax, eax
    call    printf

    ; ----- Random test (alternating pattern - somewhat unpredictable) -----
    rdtsc
    shl     rdx, 32
    or      rax, rdx
    mov     [rsp], rax              ; start time

    xor     ecx, ecx
    xor     r8d, r8d                ; toggle
.random_loop:
    inc     ecx
    test    r8b, 1
    jz      .skip
    inc     r8d
.skip:
    inc     r8d
    cmp     ecx, 1000000
    jb      .random_loop
```

```
    rdtsc
    shl     rdx, 32
    or      rax, rdx
    sub     rax, [rsp]              ; elapsed ticks
    mov     [rsp+8], rax

    ; print "Random branch cycles: "
    lea     rdi, [rel random_msg]
    xor     eax, eax
    call    printf

    ; print hex value
    lea     rdi, [rel fmt_hex]
    mov     rsi, [rsp+8]
    xor     eax, eax
    call    printf

    ; exit
    xor     edi, edi
    call    exit
```

Run the program; on a modern processor the predictable branch will usually take fewer cycles than the alternating one. If you want a truly unpredictable branch, replace the alternating pattern with a random value obtained via `rdrand` (if supported) to see a larger gap.

Tip

To observe results, simply run the binary. The output shows four lines: the messages followed by hexadecimal cycle counts. You can also use `perf stat` to measure branch misses and instructions counts while this program runs.

Branch Hinting and `cmov`

Conditional moves (`cmovcc`) eliminate the branch entirely by selecting between two values based on a condition. They are extremely useful for small if-then-else patterns where the branch is unpredictable. For example:

```
; Instead of:
; cmp rax, rcx
; jge .greater
; mov rax, rcx
; .greater:

; Use:
cmp rax, rcx
cmovl rax, rcx
```

`cmovl` moves `RCX` into `RAX` if the previous comparison set the “less than” condition. This avoids any pipeline flush. However, `cmov` introduces a data dependency on both operands and may be slower if the condition is highly predictable. Use it judiciously.

6.5 Performance Implications

Understanding the instruction execution cycle influences how you write high-performance assembly directly under Linux. While many system calls dominate runtime due to kernel transitions, compute-bound loops in encryption, image processing, or numerical kernels benefit enormously from pipeline-aware coding.

Key Principles for Performance

- **Instruction selection matters.** Choose the smallest encoding for an operation. For example, `xor eax, eax` (2 bytes) is smaller and often faster than `mov rax, 0` (7 bytes).
- **Keep dependency chains short.** Interleave independent operations. For a loop that processes arrays, unroll the loop and perform operations on different registers to fill the pipeline.
- **Avoid unpredictable branches in hot loops.** Use `cmov`, or convert conditional code to branchless arithmetic with `SETcc`, `SAR`, and `AND` masking.
- **Align branch targets.** Aligning the start of frequently-executed loops to a 16-byte or 32-byte boundary can improve fetch efficiency. Use the `align` directive in NASM:

```
align 16
.my_loop:
    ...
```

- **Know your execution ports.** If you are targeting a specific micro-architecture, you can consult the Intel optimization manual to balance uop types across available ports. NASM does not schedule uops; the hardware does, but you can provide a favourable mix.

Measuring Performance with `RDTSC`

The `RDTSC` instruction reads the processor's time-stamp counter into `EDX:EAX`. It is a simple way to measure cycles. The following snippet shows a correct use:

```
rdtsc
shl rdx, 32
or  rax, rdx
mov r8, rax          ; start time

; ... measured code ...

rdtsc
shl rdx, 32
or  rax, rdx
sub rax, r8          ; RAX = elapsed ticks
```

The time-stamp counter runs at a constant rate (the max non-turbo core frequency) on modern CPUs, so the resulting ticks can be converted to time if the frequency is known. For relative comparisons, raw ticks are sufficient. On Linux, you can also use `clock_gettime` with the coarse clock to measure wall-time in microseconds, but `RDTSC` gives cycle-accurate measurement.

Alignment and Decoding Penalties

The x86-64 decoder fetches 16 or 32 bytes per cycle. If a target address is misaligned, extra bytes must be fetched, which can reduce decode throughput. In NASM, you can specify alignment with the `align` directive. For a loop, `align 16` ensures the loop start is at a 16-byte boundary, which can improve steady-state fetch bandwidth.

Code: listing6-2.asm — Aligned loop demonstration

```
; Assemble: nasm -f elf64 -o listing6-2.o listing6-2.asm
; Link:     ld -o listing6-2 listing6-2.o

bits 64
default rel

section .text
global _start
_start:
    align 16
.aligned_loop:
    add rax, 1
    add rbx, 2
    add rcx, 3
    sub edx, 1
    jnz .aligned_loop

    ; exit
    mov     eax, 60     ; SYS_exit
    xor     edi, edi
    syscall
```

The `align` directive inserts `NOP` bytes as padding. Excessive alignment can bloat code size, so use it selectively on hot loops.

Putting It All Together

The instruction execution cycle is the invisible machinery that turns your NASM source into results. By understanding micro-operations, pipelining, branch prediction, and performance measurement, you gain the ability to write assembly that not only works but runs at the full potential of the hardware. The next chapters will apply this knowledge to real Linux system calls, function prologues, and data manipulation routines, always with an eye on the underlying pipeline.

Part II

NASM TOOLCHAIN ON FEDORA 43

7

Installing NASM on Fedora 43

In This Chapter

Before a single line of assembly can be written, the Netwide Assembler must be correctly installed on your Fedora 43 machine. This chapter walks you through obtaining NASM from the official repositories or directly from the source, verifying the installation, and setting up a clean project folder structure that will house every example in this book. All steps have been tested on Fedora 43 with NASM 2.16.01, the recommended stable release at the time of writing.

7.1 Downloading and Installing NASM

NASM is available as a free, open-source package. On Fedora, the simplest method is to install it directly from the distribution's repositories. Alternatively, you can compile from source if you need the absolute latest version or custom build options.

Installing from the Fedora Repositories

Open a terminal and run:

```
sudo dnf install nasm
```

This command installs the latest version of NASM that is packaged for Fedora 43, which is always at least 2.16. The package manager also sets up the `man` pages and places the executable in `/usr/bin/nasm`, so no further path configuration is necessary.

Installing from Source (Optional)

If you need an even newer version or want to control the build options, you can compile NASM from the official source. Download the latest source tarball from `https://www.nasm.us/pub/nasm/releasebuilds/`.

Then extract and build:

```
tar xf nasm-2.16.01.tar.gz
cd nasm-2.16.01
./configure
make
sudo make install
```

By default, this installs the binary to `/usr/local/bin`. Ensure that this directory is in your `PATH` (it usually is on Fedora). Verify the installation by running `nasm -v`.

7.2 Environment Setup

Unlike Windows, Fedora places system-wide executables in directories that are already in the `PATH`. The `dnf` installation automatically adds NASM to `/usr/bin`, which is always searched. No manual environment variable editing is required.

Verifying the Installation

Open a terminal and type:

```
nasm -v
```

You should see output similar to:

```
NASM version 2.16.01 compiled on Mar 15 2026
```

If the command is not found, check that the package installed correctly (`dnf list installed nasm`) or, if you built from source, that `/usr/local/bin` is in your `PATH`.

7.3 Verification with a Minimal Program

Create a folder for your assembly projects, for example `~/nasm-projects/hello`. Inside it, create a file named `hello.asm` with the following content. This program does nothing except exit with a zero status; its sole purpose is to confirm that the assemble-and-link cycle works.

Code: hello.asm — Minimal test program for Fedora

```
; hello.asm -- minimal Linux x64 program
; Assemble: nasm -f elf64 -o hello.o hello.asm
; Link:     ld -o hello hello.o

bits 64
default rel

section .text
global _start
_start:
    mov     eax, 60         ; SYS_exit
    xor     edi, edi        ; exit code 0
    syscall
```

Assemble the file with:

```
nasm -f elf64 -o hello.o hello.asm
```

If the command completes without errors, the object file `hello.o` appears. Then link it:

```
ld -o hello hello.o
```

This produces the executable `hello`. Run it:

```
./hello
```

Check the exit status with `echo $?`, which should print `0`. If it does, your toolchain is working.

7.4 Project Structure Setup

Organising your work into a disciplined folder hierarchy saves time and prevents mistakes as projects grow. The structure recommended here works equally well for single-file experiments and multi-module projects.

Recommended Directory Layout

Create a root project folder, e.g. `~/nasm-projects/myproject`. Inside it, create the following subdirectories:

- `src/` — contains all `.asm` source files.
- `obj/` — intermediate object files (generated).
- `bin/` — final executables (generated).
- `include/` — shared macro and constant files.
- `scripts/` — build scripts or Makefiles.

For the smallest possible project, you can simply keep everything in the same directory, but separating generated files keeps the source folder clean.

A Simple Build Script

Create a shell script `build.sh` in the project root. It assembles a source file and links it. For example:

Code: build.sh — Simple build script

```
#!/bin/bash
set -e
NASM="nasm"
LD="ld"
SRC_DIR="src"
OBJ_DIR="obj"
BIN_DIR="bin"

mkdir -p "$OBJ_DIR" "$BIN_DIR"

$NASM -f elf64 -o "$OBJ_DIR/main.o" "$SRC_DIR/main.asm"
$LD -o "$BIN_DIR/main" "$OBJ_DIR/main.o"
```

```
echo "Build successful: $BIN_DIR/main"
```

Make it executable with `chmod +x build.sh` and run it with `./build.sh`.

If your program uses the C library, change the linker to `gcc -no-pie -o ...` and call your entry point `main` instead of `_start`. The script can be adjusted accordingly.

Tip

Using a Makefile is even more powerful. See Chapter 9 (Assembling and Linking Workflow) for a full Makefile example.

Completing the Installation

With NASM installed, verified, and organised in a project structure, you are ready to proceed. All subsequent chapters assume that you have a working assembly environment matching this configuration. The next chapter introduces how to set up Visual Studio Code as a powerful editor for NASM development on Fedora.

8

Using Visual Studio Code Efficiently for NASM Development

In This Chapter

Visual Studio Code is a free, cross-platform editor that has become a favourite for many low-level developers. On Fedora, you can configure it to provide syntax highlighting, one-click build and run, integrated debugging with GDB, and project management that rivals full IDEs. This chapter details every step required to transform a plain VS Code installation into a productive NASM development environment on Fedora 43. You will learn to install the editor, select extensions, set up build tasks, integrate debugging with GDB, organise multi-file projects, and adopt workflow habits that minimise friction. All examples have been tested with VS Code 1.95 and later, NASM 2.16.01, GNU ld, and GDB on Fedora 43.

8.1 Installing VSCode on Fedora

The easiest way to install Visual Studio Code is by adding the Microsoft repository and using `dnf`:

```
sudo rpm --import https://packages.microsoft.com/keys/microsoft.asc
sudo sh -c 'echo -e "[code]\nname=Visual Studio Code\nbaseurl=https://packages.microsoft.com/yumre
↪  pos/vscode\nenabled=1\ngpgcheck=1\ngpgkey=https://packages.microsoft.com/keys/microsoft.asc" >
↪  /etc/yum.repos.d/vscode.repo'
sudo dnf check-update
sudo dnf install code
```

Alternatively, you can download the RPM package from `https://code.visualstudio.com/download` and install it with `sudo rpm -i <package>`. Once installed, you can launch VS Code from the terminal by typing `code .` in any project directory.

8.2 Recommended Extensions for NASM Development

Open VS Code and go to the Extensions view (`Ctrl+Shift+X`). Install the following extensions:

- **x86 and x86_64 Assembly** (by 13xforever). Provides full syntax highlighting for NASM, opcode completion, bracket matching, and tooltips.
- **ASM Code Lens** (by maziac). Adds code-lens annotations above labels, showing reference counts and making navigation easier.
- **NASM Language Support** (by doinkythederp). A lightweight grammar specific to NASM.
- **Hex Editor** (by ms-vscode.hexeditor). Allows you to inspect binary object files and executables directly inside the editor – useful for verification.

At a minimum, install the first extension. Create a `.asm` file; the mnemonics and registers should be colourised.

8.3 Configuring NASM Build Tasks (tasks.json)

VS Code's task system allows you to run external commands directly from the editor with `Ctrl+Shift+B`. By defining a build task, you can assemble and link your project in one step.

Creating the Default Build Task

Open the Command Palette (`Ctrl+Shift+P`) and run `Tasks: Configure Default Build Task`. Choose `Create tasks.json file from template`, then `Others`. Replace the generated stub with the configuration below.

Code: .vscode/tasks.json — NASM build task for Linux

```
{
    "version": "2.0.0",
    "tasks": [
        {
            "label": "Assemble & Link (NASM + ld)",
            "type": "shell",
            "command": "nasm",
            "args": [
                "-f", "elf64",
                "-o", "obj/main.o",
                "src/main.asm"
            ],
            "dependsOn": "Link",
            "group": {
                "kind": "build",
                "isDefault": true
            },
            "problemMatcher": []
        },
        {
            "label": "Link",
```

```
            "type": "shell",
            "command": "ld",
            "args": [
                "-o", "bin/main",
                "obj/main.o"
            ],
            "group": "build"
        }
    ]
}
```

This assumes a source file `src/main.asm` with an entry point `_start` and no library dependencies. If your program uses the C library, change the linker to `gcc -no-pie -o bin/main obj/main.o` and ensure that the source contains a `main` function.

Press `Ctrl+Shift+B` to run the build. The terminal will show the output. If any error occurs, the messages are displayed directly in the terminal and can be clicked to jump to the line (if the problem matcher is configured; even without it, line numbers are shown).

8.4 Configuring Debugging (launch.json)

While VS Code does not natively debug assembly code, you can integrate GDB seamlessly. Create a `.vscode/launch.json` file with the following content. This configuration assembles the program (via the build task) and then launches GDB on the freshly built executable.

Code: .vscode/launch.json — Debug with GDB

```
{
    "version": "0.2.0",
    "configurations": [
        {
            "name": "Debug (GDB)",
            "type": "cppdbg",
            "request": "launch",
            "program": "${workspaceFolder}/bin/main",
            "args": [],
            "stopAtEntry": true,
            "cwd": "${workspaceFolder}",
            "environment": [],
            "externalConsole": false,
            "MIMode": "gdb",
            "miDebuggerPath": "/usr/bin/gdb",
            "setupCommands": [
                {
                    "description": "Enable pretty-printing for gdb",
                    "text": "-enable-pretty-printing",
                    "ignoreFailures": true
                },
                {
                    "description": "Set Disassembly Flavor to Intel",
                    "text": "set disassembly-flavor intel"
                }
```

```
            ],
            "preLaunchTask": "Assemble & Link (NASM + ld)"
        }
    ]
}
```

Now pressing `F5` will build your project and start GDB. The execution will pause at the entry point. Use the Debug Console to inspect registers with `-exec info registers`, step through instructions with `F10` (step over) or `F11` (step into), and watch memory. This setup gives you a near-IDE experience for assembly.

Tip

Make sure GDB is installed on Fedora (`sudo dnf install gdb`). If you prefer a graphical debugger, you can replace `miDebuggerPath` with a path to a GDB frontend like `ddd`, or simply launch `gdb` manually from the integrated terminal.

8.5 Integrated Terminal Workflow

The most frictionless way to build and debug is to use VS Code's integrated terminal (`Ctrl+`). You can open a terminal pane at the bottom of the editor and treat it exactly like any command-line session. This approach eliminates the need for elaborate configurations and lets you run commands directly.

1. Open the terminal (`Ctrl+`).
2. Run `nasm -f elf64 -o obj/main.o src/main.asm` and watch for errors.
3. Run `ld -o bin/main obj/main.o` to link.
4. Execute `./bin/main` directly.
5. If you need to debug, launch `gdb ./bin/main` from the same terminal.

This is the recommended method for beginners because you see every step and can easily switch to other tools.

8.6 Creating Reusable Project Templates

Save time on new projects by creating a master template folder containing:

- `src/` (with a stub `main.asm`)
- `obj/` (empty)
- `bin/` (empty)
- `.vscode/` with `tasks.json` and `launch.json`
- `scripts/` with a `build.sh`

Copy this folder, rename it, and open it in VS Code. All the boilerplate is ready; you only need to modify the source file.

8.7 Organizing NASM Project Structure

A clean directory layout becomes crucial as projects grow. A recommended structure:

```
project/
  src/
    main.asm
    helpers.asm
  include/
    constants.inc
  obj/
    (generated .o files)
  bin/
    (generated executables)
  scripts/
    build.sh
    clean.sh
  .vscode/
    tasks.json
    launch.json
```

The `include` directory contains shared files with `%include ''include/constants.inc''`. This keeps things tidy.

8.8 One-Click Build and Run System

You can extend `tasks.json` to run the executable after a successful build. For example:

Code: tasks.json — Build and Run

```
{
    "version": "2.0.0",
    "tasks": [
        {
            "label": "Build",
            "type": "shell",
            "command": "nasm",
            "args": ["-f", "elf64", "-o", "obj/main.o", "src/main.asm"],
            "group": "build",
            "dependsOn": "Link"
        },
        {
            "label": "Link",
            "type": "shell",
            "command": "ld",
            "args": ["-o", "bin/main", "obj/main.o"],
```

```
            "group": "build"
        },
        {
            "label": "Run",
            "type": "shell",
            "command": "${workspaceFolder}/bin/main",
            "dependsOn": "Build",
            "group": {
                "kind": "test",
                "isDefault": true
            }
        }
    ]
}
```

Press `Ctrl+Shift+B` and the program will be built and run, with output displayed in the terminal.

8.9 Multi-File Assembly Project Management

For projects with several source files, adjust `tasks.json` to assemble each file and then link all objects together. An easier approach is to use a shell script or a Makefile that VS Code invokes via a task. Here is an example `build.sh` for multiple files:

Code: build.sh — Multi-file assembly

```
#!/bin/bash
set -e
nasm -f elf64 -o obj/main.o src/main.asm
nasm -f elf64 -o obj/io.o   src/io.asm
ld -o bin/prog obj/main.o obj/io.o
```

The task simply calls this script.

8.10 Productivity Optimization Techniques

Several small adjustments can dramatically speed up your development.

Keyboard Shortcuts

- `Ctrl+Shift+B` — build.
- `Ctrl+` — toggle terminal.
- `F5` — start debugging.
- `Ctrl+B` — toggle sidebar (more space for code).
- `Ctrl+K Ctrl+S` — open keyboard shortcuts editor; you can bind your custom task to a specific key.

Snippets

Create user snippets for NASM (`File > Preferences > User Snippets > assembly`) to quickly insert common patterns. For example, a minimal function prologue:

Code: nasm.json snippet — Function prologue

```
{
    "Func prologue": {
        "prefix": "func",
        "body": [
            "push rbp",
            "mov  rbp, rsp",
            "sub  rsp, ${1:0x20}",
            "",
            "${2:; body}",
            "",
            "mov  rsp, rbp",
            "pop  rbp",
            "ret"
        ],
        "description": "Linux x64 function prologue"
    }
}
```

Type `func` and press `Tab` to insert the template.

Multi-Cursor Editing

Hold `Alt` and click to add multiple cursors. This is extremely handy when you need to add the same register or modifier to several lines.

Terminal Panes

Split the terminal (`Ctrl+Shift+5`) to keep one pane for building and another for running or monitoring logs.

Important

Spend the time to configure your editor now. A smooth development environment reduces the feedback loop and lets you focus on learning assembly, not fighting with tools.

9

Assembling and Linking Workflow

In This Chapter

Turning a human-readable `.asm` source file into a running Linux executable involves two distinct stages: assembly and linking. This chapter dissects each stage in detail, from the moment NASM parses your code until the linker produces an ELF executable. You will learn the exact command-line options, the structure of the intermediate object file, how unresolved symbols are resolved, and how to automate the entire process. Every fact is drawn from the NASM 2.16 documentation, the ELF specification, and the observed behaviour of the GNU linker.

9.1 NASM Assembly Process

NASM reads a plain-text source file and converts it into an object file in **ELF64** format. The basic command is:

```
nasm -f elf64 -o output.o input.asm
```

The `-f elf64` switch tells NASM to generate a 64-bit ELF object, which is the standard for Linux. If you omit this flag, NASM defaults to a flat binary, unsuitable for linking.

During assembly, NASM performs several steps:

1. **Preprocessing.** Expands `%include`, `%define`, `%macro` directives, resulting in pure code.
2. **Lexing and Parsing.** Mnemonics, registers, and addressing modes are recognised and checked. Errors are reported with line numbers.
3. **Code Generation.** Each instruction is translated into its machine-language encoding. Label addresses are recorded as offsets within sections.

4. **Symbol and Relocation Generation.** Symbols marked `global` are exported. Symbols declared `extern` are left as references to be resolved by the linker. Relocation entries tell the linker where to patch addresses.

5. **Object File Writing.** The assembled code, data, symbol table, and relocation tables are written into an ELF object file.

A minimal source file that produces a valid object:

Code: minimal.asm — Smallest valid elf64 source

```
; minimal.asm
bits 64
default rel

section .text
global _start
_start:
    mov eax, 60     ; sys_exit
    xor edi, edi
    syscall
```

When assembled, the object file contains the machine code bytes for the three instructions, along with a defined symbol `_start`.

9.2 Object File Generation

The ELF object file consists of an ELF header, section headers, and the actual section data (code, data, etc.). Key components:

- **ELF Header.** Specifies that this is a 64-bit object, the entry point (if an executable, but for an object it's usually 0), and the locations of the section header table.
- **Section Headers.** Each section (`.text`, `.data`, `.bss`) has a header with its name, size, and attributes (executable, writable, etc.).
- **Symbol Table (`.symtab`).** Lists every label (global and local) and extern declarations. Each symbol has a value (address or zero), size, and binding (local/global).
- **Relocation Tables (`.rela.text` etc.).** For code that references unresolved symbols, relocation entries tell the linker how to compute the final address.

You can inspect an object file with `readelf -S hello.o` or `objdump -d hello.o`, both standard tools on Fedora.

9.3 Linking Process

Linking combines one or more object files into a final executable. On Linux, the standard linker is `ld` from GNU binutils. It resolves cross-references, performs relocations, and builds the ELF executable structure.

Linking with GNU ld (pure assembly, no C library)

If your program uses only system calls and defines `_start`:

```
ld -o prog prog.o
```

The linker reads the object, assigns final addresses to sections, resolves any references to undefined symbols (you normally have none in pure system-call programs), and sets the entry point to the address of `_start` (which must be `global`). The resulting executable is static and does not depend on any shared libraries except the kernel.

Linking with the C Library

If your code calls C functions (e.g., `printf`) and defines a `main` symbol:

```
gcc -no-pie -o prog prog.o
```

The `gcc` compiler driver automatically links against the C library (`libc`) and the necessary startup files (`crt1.o` etc.). The `-no-pie` flag produces a non-position-independent executable, simplifying address calculations. If you need a position-independent executable (PIE), omit `-no-pie` and ensure your code uses RIP-relative addressing; then the linker will emit the appropriate relocations.

Symbol Resolution

The linker processes all object files and shared libraries referenced on the command line. For undefined symbols, it searches the symbol tables of libraries (e.g., `libc.so`) and resolves them to the correct addresses. If a symbol remains unresolved, the linker reports "undefined reference to `symbol`" and stops.

For calls to C library functions, the linker does not embed the function itself; it creates a Procedure Linkage Table (PLT) stub. At runtime, the dynamic linker resolves the actual address. This happens automatically and is transparent to your assembly code.

Execution of the Final ELF Binary

When you run the program, the kernel's loader:

1. Reads the ELF program headers to determine which parts of the file to map into memory.
2. Sets up the initial stack and registers (RSP, environment). For an `_start` program, the stack contains argc, argv, and envp.
3. If dynamically linked, the dynamic linker `/lib64/ld-linux-x86-64.so.2` is invoked to resolve shared library references and perform relocations.
4. Jumps to the entry point (`_start` or `main` after startup code initialises libc).

9.4 Executable Creation Walkthrough

Let's trace a complete program that prints a string using the C library from assembly.

Code: hello-libc.asm — Assembling and linking with libc

```
; hello-libc.asm
; Assemble: nasm -f elf64 -o hello-libc.o hello-libc.asm
; Link:     gcc -no-pie -o hello-libc hello-libc.o

bits 64
default rel

extern printf
extern exit

section .data
msg db 'Hello, Fedora!',10,0

section .text
global main
main:
    sub     rsp, 8          ; align stack for call to printf
    lea     rdi, [rel msg]
    xor     eax, eax
    call    printf
    add     rsp, 8
    xor     edi, edi
    call    exit            ; never returns
```

After assembly, `hello-libc.o` contains:

- `.text` with the code referencing `printf` and `exit` as unresolved symbols.
- `.data` with the message.
- Undefined symbols: `printf`, `exit`.
- Defined symbol `main`.

Linking with `gcc -no-pie -o hello-libc hello-libc.o` resolves `printf` and `exit` from the C library, adds the startup code, and generates the executable. Running `./hello-libc` displays the message.

9.5 Build Automation

Automating the build process reduces errors and speeds up development.

Shell Script

A simple `build.sh`:

```
#!/bin/bash
set -e
nasm -f elf64 -o obj/main.o src/main.asm
ld -o bin/main obj/main.o           # or gcc -no-pie ...
```

Run with `./build.sh`.

Makefile

For multi-file projects, a Makefile is ideal. Example:

Code: Makefile

```
NASM = nasm
LD   = ld

SRC_DIR = src
OBJ_DIR = obj
BIN_DIR = bin

OBJS = $(OBJ_DIR)/main.o $(OBJ_DIR)/helpers.o
TARGET = $(BIN_DIR)/prog

$(TARGET): $(OBJS)
^^I$(LD) -o $(TARGET) $(OBJS)

$(OBJ_DIR)/%.o: $(SRC_DIR)/%.asm
^^I@mkdir -p $(OBJ_DIR)
^^I$(NASM) -f elf64 -o $@ $<

clean:
^^Irm -rf $(OBJ_DIR) $(BIN_DIR)
```

Run `make` to build, `make clean` to remove generated files.

VS Code Tasks

As shown earlier, define a build task in `tasks.json` that invokes the script or Makefile. Then a single key-press builds everything.

Combining Build and Run

A convenient script can assemble, link, and immediately execute:

```
#!/bin/bash
set -e
nasm -f elf64 -o test.o test.asm
ld -o test test.o
./test
```

This is especially useful during rapid experimentation.

Important

Always check errors. Ensure the script stops on any failure (using `set -e` or checking return codes). Running a partially linked executable because you missed an assembly error will lead to confusing crashes.

Putting It All Together

Mastering the assembly and linking workflow is fundamental. You now know how to produce object files, link them into executables, and automate the process. The following parts of this book will dive deep into the NASM language and the Linux system-call interface, building on this toolchain foundation.

10

Object Files and Executables (ELF)

In This Chapter

The journey from assembly source to a running program on Linux passes through the Executable and Linkable Format (ELF). This chapter describes the internal structure of ELF object files and executables as they are used by NASM and the GNU toolchain on Fedora 43. You will learn how sections, symbols, and relocations are organised, how the entry point is specified, and how the linker resolves references across modules and shared libraries. Every description is verified against the System V ABI (generic ABI) and the AMD64 supplement, the ELF specification, and the observed behaviour of the kernel loader. Working assembly examples illustrate how to control section placement and how symbols are exported and imported.

10.1 ELF Format Overview

The ELF format is the standard binary format for object files, shared libraries, and executables on Linux. NASM produces 64-bit ELF object files when the `-f elf64` switch is given. The ELF specification defines three main views:

- **Object file.** Relocatable code and data, with symbol and relocation tables.
- **Executable.** Fully linked program, ready to be loaded by the kernel.
- **Shared object.** Position-independent code that can be linked at load time.

10.1.1 ELF Header

Every ELF file begins with a 64-byte header. For a 64-bit object, it identifies the file as ELF64, the target architecture (`EM_X86_64`), and the entry point (0 in object files). It also records the offsets and sizes of the program header table (for executables) and the section header table.

Below is a minimal NASM source that produces an ELF object with two sections.

Code: listing10-1.asm — Minimal object with two sections

```
; listing10-1.asm
bits 64
default rel

section .text  progbits alloc exec  align=16
global _start
_start:
    xor eax, eax
    ret

section .data  progbits alloc write align=8
value dq 0x123456789ABCDEF0
```

Assemble with `nasm -f elf64 -o test.o listing10-1.asm`. The resulting ELF contains a file header, section headers for `.text` and `.data`, the assembled bytes, a symbol table with `_start` and `value`, and no relocation entries since all references are resolved.

10.1.2 Section Headers

Each section is described by a 64-byte section header. It includes the section name (stored in a string table), type (`SHT_PROGBITS` for code/data, `SHT_NOBITS` for BSS), flags (allocatable, writable, executable), virtual address (zero for relocatable objects), file offset, size, and alignment. NASM automatically sets the flags based on your section directives: `exec` makes it executable, `write` makes it writable, and `alloc` ensures it is memory-resident. The `align=` parameter controls the alignment in the object; the linker determines the final alignment in the executable.

10.1.3 Symbol Table

The symbol table (`.symtab`) records every label and external reference. Each 24-byte entry (Elf64_Sym) contains the name (via an offset into the string table), information about the symbol binding (`STB_LOCAL`, `STB_GLOBAL`) and type (`STT_FUNC`, `STT_OBJECT`), the section index where the symbol is defined, the value (offset within that section), and size. Symbols declared `extern` are recorded as undefined (`SHN_UNDEF`) with a zero value.

Relocation tables (`.rela.text`, `.rela.data`) contain entries that tell the linker how to patch addresses. For a 64-bit code reference, NASM emits an `R_X86_64_PC32` relocation for a `CALL` to an external function (32-bit PC-relative) and `R_X86_64_64` for an absolute address reference. For local symbols, relocations are usually not needed as offsets are known.

10.1.4 A Practical Relocation Example

The following program calls `printf` from the C library, creating an undefined symbol and a relocation.

Code: listing10-2.asm — Extern symbol and relocation

```
; listing10-2.asm
bits 64
default rel

extern printf
extern exit

section .data
msg db 'Hello, ELF!',10,0

section .text
global main
main:
    sub     rsp, 8                  ; align stack
    lea     rdi, [rel msg]
    xor     eax, eax
    call    printf                  ; generates PC32 relocation
    add     rsp, 8
    xor     edi, edi
    call    exit
```

Assemble with `nasm -f elf64 -o listing10-2.o listing10-2.asm`. The object file contains relocation entries for `printf` and `exit`. Use `readelf -r listing10-2.o` to see them.

10.2 ELF Executable Structure (ET_EXEC)

When you link one or more object files with `ld -o prog prog.o`, the output is a statically-linked ELF executable (if no shared libraries are used). The linker merges sections, assigns final virtual addresses, resolves relocations, and builds the program header table. The executable contains:

- **ELF header.** Entry point set to the address of `_start`.
- **Program header table.** Describes the segments (load-able memory regions) like the text segment (code + read-only data) and the data segment (initialised data + BSS). The kernel uses these to map the file into memory.
- **Section header table** (optional but present). Used by tools like `objdump` and debuggers.

The entry point is the first instruction executed. For a pure assembly program with `_start`, the linker uses that address. If you link with the C library using `gcc -no-pie -o prog prog.o`, the entry point is the C runtime start routine (`_start` in crt1.o) which calls `main`; your `main` label becomes a normal function.

10.2.1 Examining an Executable

With the executable built from listing10-2, you can inspect it with standard tools:

```
readelf -h listing10-2      # ELF header, entry point
readelf -l listing10-2      # program headers
objdump -d listing10-2      # disassembly
```

10.3 Sections and the Section Control Directive

NASM gives you fine control over section placement and attributes:

- `section .text progbits alloc exec align=16` – code section.
- `section .data progbits alloc write align=8` – initialized data.
- `section .bss nobits alloc write align=8` – zero-initialized data.
- `section .rodata progbits alloc noexec read align=4` – read-only data.

The `nobits` type indicates that the section occupies no file space; the loader allocates and zeros it. The `read` flag is implied by the absence of `write` and is set by the assembler.

10.3.1 Custom Sections

To create a custom, non-standard section, you can use something like:

```
section .myvars progbits alloc write align=8
my_count dq 0
```

The linker will include it in the data segment. The `progbits` type ensures it is initialised data.

10.4 Entry Point Mechanics

The entry point is determined by the symbol given to the linker. For `ld`, if you do not specify `--entry`, the linker looks for `_start`. If you want a different entry point, pass `--entry my_entry`.

When the kernel starts a program, it sets up the initial stack and registers, then jumps to the entry point. For a `_start` program, the stack layout is:

- `[rsp]` – argc (number of arguments)
- `[rsp+8]` – argv[0] (pointer to program name)
- `[rsp+16]` – argv[1] ...
- ...

The program can retrieve these by reading from the stack. For a `main` program linked with libc, the C runtime does this for you and passes argc and argv as arguments.

10.5 Symbol Resolution and Dynamic Linking

When you use shared libraries (e.g., `extern printf`), the executable created by `gcc` or `ld` contains a dynamic section that instructs the runtime dynamic linker (`/lib64/ld-linux-x86-64.so.2`) to resolve symbols at load time. The procedure linkage table (PLT) and global offset table (GOT) are used:

- A `call printf` in your code jumps to a PLT stub.
- The stub jumps through a GOT entry. Initially, the GOT entry points back into the dynamic linker.

- The first time the function is called, the dynamic linker resolves the real address and updates the GOT. Subsequent calls go directly to the function.

This mechanism is completely transparent; you simply declare `extern` and call the function. The linker and the dynamic loader handle everything.

10.5.1 Global Symbols within the Same Module

If you define a symbol with `global` in one object file and reference it with `extern` in another, the static linker resolves it directly without PLT/GOT involvement, unless you are building a shared library.

Code: mathlib.asm — Exporting a function

```
; mathlib.asm
bits 64
default rel
section .text
global add_two
add_two:
    lea rax, [rdi + rsi]
    ret
```

Code: main.asm — Using the function

```
; main.asm
bits 64
default rel
extern add_two
extern exit

section .text
global _start
_start:
    mov     rdi, 3
    mov     rsi, 7
    call    add_two         ; RAX = 10
    mov     rdi, rax
    call    exit            ; exit(10)
```

Assemble and link:

```
nasm -f elf64 -o mathlib.o mathlib.asm
nasm -f elf64 -o main.o main.asm
ld -o mathprog mathlib.o main.o -lc --dynamic-linker=/lib64/ld-linux-x86-64.so.2
```

Here we explicitly linked with the C library because we use `exit`. The `add_two` symbol is resolved locally by the static linker.

Putting It All Together

Understanding the ELF format allows you to diagnose linking problems, inspect binaries, and control memory layout. Every directive you write — `section`, `global`, `extern` — influences the

object and the final executable precisely as defined by the ELF standard. The next chapter dives into the linkers available on Linux.

11

Using Linkers (GNU ld, LLVM lld, and GCC as Linker Driver)

In This Chapter

NASM produces standard 64-bit ELF object files, which can be linked by a variety of linkers. This chapter covers the two most important linkers used on Linux for assembly development: the GNU linker (`ld`), the LLVM linker (`ld.lld`), and how the GCC compiler driver can act as a convenient linker front-end. You will learn how to invoke each one with the correct options to convert NASM output into a working executable, how to link against libraries, and how to diagnose common linking errors. Every command shown has been tested with NASM 2.16.01 on Fedora 43.

11.1 GNU ld — The Default Linux Linker

The GNU linker (`ld`) is part of binutils and is installed by default on Fedora as `/usr/bin/ld`. It reads ELF objects and produces executables or shared libraries.

Basic Invocation for Pure Assembly (No C Library)

If your program uses only system calls and defines `_start`:

```
ld -o prog prog.o
```

The linker will set the entry point to the address of `_start` and produce a static executable that does not depend on any shared library (except the Linux kernel's vDSO). No standard library functions are available.

Linking with the C Library

When your program calls C functions like `printf` or `exit`, you must link against the C library. The simplest way is to use GCC as a linker driver (see later section), but you can also call `ld` directly with the appropriate arguments:

```
ld -o prog prog.o -lc --dynamic-linker=/lib64/ld-linux-x86-64.so.2
```

- `-lc` links against the C library (`libc.so`).
- `--dynamic-linker` specifies the path to the dynamic linker that will resolve shared libraries at runtime.
- Additional libraries like `-lm` (math) can be added similarly.

However, direct use of `ld` for C-linked programs requires you to know the exact startup files and paths. It is generally easier to use GCC for such cases.

Specifying the Entry Point

If your entry point is not named `_start`, use `--entry symbol`:

```
ld -o prog prog.o --entry my_main
```

Producing a Map File

To see the addresses of all symbols, add `-M` or `-Map=file.map`:

```
ld -o prog prog.o -Map=prog.map
```

Creating a Shared Library

To build a shared library from assembly code:

```
nasm -f elf64 -o mylib.o mylib.asm
ld -shared -o libmylib.so mylib.o
```

If your library exports symbols, they must be marked `global`. You can also use a version script or export list with `--version-script`. The resulting `.so` can be linked with other programs.

11.2 LLVM lld — A Fast Alternative

LLVM's linker `ld.lld` is a drop-in replacement for GNU ld that is often faster, especially for large projects. On Fedora, you can install it with `sudo dnf install lld`. The command is `ld.lld` (or `lld` with the `-flavor gnu` option).

Basic Invocation

The syntax is intentionally compatible with GNU ld:

```
ld.lld -o prog prog.o
```

For C library programs, similarly:

```
ld.lld -o prog prog.o -lc --dynamic-linker=/lib64/ld-linux-x86-64.so.2
```

lld also supports many of the same flags, including `--entry`, `-shared`, and `-Map`. It is fully compatible with ELF objects produced by NASM.

11.3 Using GCC as a Linker Driver

GCC (the GNU Compiler Collection) is commonly used to compile C code, but it also acts as a convenient linker front-end. It automatically adds the necessary startup files (`crt1.o`, `crti.o`, `crtn.o`) and links the C library. This is the recommended method when you mix assembly with C code or simply want to use libc functions without worrying about the low-level linker details.

Linking an Assembly Program with libc

If your assembly file defines `main` (instead of `_start`):

```
nasm -f elf64 -o prog.o prog.asm
gcc -no-pie -o prog prog.o
```

The `-no-pie` flag creates a non-position-independent executable; you can omit it if your code is fully PIC (using `default rel` and no absolute addresses). The resulting executable correctly links with the C library.

Adding Other Libraries

To link additional libraries, use the `-l` flag:

```
gcc -no-pie -o prog prog.o -lm -lpthread
```

11.4 Library Linking in Detail

Libraries are collections of object files (static) or position-independent code (shared) that provide reusable functionality. NASM programs can both use and create libraries.

Static Libraries (.a)

A static library is created by archiving object files with `ar`:

```
ar rcs libutils.a util1.o util2.o
```

To link your program with a static library:

```
gcc -no-pie -o prog prog.o -L. -lutils
```

`-L.` adds the current directory to the library search path; `-lutils` links with `libutils.a` (or `libutils.so` if shared is preferred).

Shared Libraries (.so)

Creating a shared library from NASM objects:

```
nasm -f elf64 -o mylib.o mylib.asm
gcc -shared -o libmylib.so mylib.o
```

Using a shared library is identical; the linker will prefer `.so` unless you provide `-static`.

11.5 Common Linking Errors and Solutions

Below are typical problems when linking NASM objects with `ld` or `gcc`:

- **undefined reference to 'symbol'** — The linker cannot find the symbol. Check that you have declared it `extern` correctly, and that the object file or library defining it is included on the command line. For C library functions, ensure `-lc` or `gcc` is used.
- **cannot find entry symbol _start** — When using `ld` directly, you need a global `_start` label. If you intend to use `main`, link with `gcc` or provide `--entry main`.
- **relocation R_X86_64_32 against '.data' can not be used when making a PIE object; recompile with -fPIE** — This occurs when you try to use absolute addresses (e.g., `mov eax, my_var`) in a position-independent executable. Use RIP-relative addressing (`default rel` and `[rel my_var]`), or compile with `-no-pie`.
- **undefined reference to '___libc_start_main'** — You are linking with `ld` directly and using `main` as the entry point, but the C runtime startup is missing. Link with `gcc` instead, or manually add startup files (not recommended).

Warning

Stack alignment. When calling C library functions, the stack must be 16-byte aligned at the point of the `CALL`. The linker does not check this; a misaligned stack will cause crashes inside library functions. Always verify your prologue.

Using Verbose Mode

To see exactly what the linker is doing, add `-Wl,--verbose` to the GCC command or `--verbose` to `ld`. This lists the libraries and paths searched, helping diagnose symbol resolution failures.

12

Debugging Tools Setup

In This Chapter

Assembly language programming provides total control, but also leaves no safety net. This chapter introduces the GNU Debugger (GDB), the standard debugging tool on Linux, and shows how to install it, set breakpoints, step through instructions, inspect and modify registers and memory, and diagnose problems in your NASM programs. You will also learn about a graphical front-end that makes the process more visual. All examples have been tested on Fedora 43 with NASM 2.16 and GDB 14.

12.1 Installing GDB

GDB is not always installed by default. To install it on Fedora, run:

```
sudo dnf install gdb
```

Optionally, you can install the `ddd` graphical front-end (`sudo dnf install ddd`) or the integrated debugging in Visual Studio Code (which uses GDB behind the scenes).

12.2 Basic GDB Concepts

Debugging assembly code with GDB involves:

- **Starting the debugger** with the executable.
- **Setting breakpoints** on labels or addresses.
- **Running** the program until a breakpoint is hit.
- **Stepping** one instruction at a time.

- **Examining** registers, memory, and the stack.
- **Modifying** register or memory values to test behaviour.

12.3 Starting GDB with Your Program

If your executable is named `prog`, launch GDB by typing:

```
gdb ./prog
```

For assembly without debugging symbols, you may need to specify the architecture:

```
gdb -q ./prog
```

The `-q` (quiet) option suppresses the startup banner.

12.4 Essential GDB Commands

Below is a compact reference of the most important commands. GDB commands can be abbreviated as long as they remain unique (e.g., `c` for continue, `si` for stepi, `ni` for nexti).

Execution Control

- `run` (or `r`) — Start the program.
- `continue` (or `c`) — Resume execution until next breakpoint or exit.
- `stepi` (or `si`) — Step exactly one machine instruction (traces into calls).
- `nexti` (or `ni`) — Step one instruction but step over calls (execute the call and stop after return).
- `finish` — Continue until the current function returns.

Breakpoints

- `break *address` — Set a breakpoint at an absolute address, e.g., `break *0x400100`.
- `break symbol` — If symbols are available, break on label, e.g., `break _start` or `break main`.
- `info breakpoints` (or `i b`) — List breakpoints.
- `delete` — Delete breakpoint by number (or `delete` all).
- `disable N` — Disable breakpoint N without deleting.

Examining Registers and Memory

- `info registers` (or `i r`) — Display all general-purpose registers and flags.
- `info registers rax` — Show only RAX.
- `print/x $rax` — Print the value of RAX in hex.
- `x/nfu address` — Examine memory:

  - `x/10i $rip` — 10 instructions starting at RIP.
  - `x/4gx $rsp` — 4 giant (8-byte) words in hex at RSP.
  - `x/s 0x402000` — Display as null-terminated string.
- `display/format` — Automatically show an expression after each step, e.g., `display/i $rip` to show the next instruction.

Modifying State

- `set $rax = 0x1234` — Change the value of register RAX.
- `set *(unsigned long long*)0x601000 = 0xFF` — Write a qword to an absolute address.
- `set var = value` — If debugging with symbols, modify a variable directly.

Stack Inspection

- `backtrace` (or `bt`) — Show the call stack (works only if frame pointers are used or if DWARF unwind information is present).
- `frame N` — Select a stack frame.
- `info frame` — Details about the current frame.

12.5 A Complete Debugging Walkthrough

We will debug a simple program that writes a message using a system call. Assemble and link this source:

Code: debugme.asm — Program to debug

```
; debugme.asm
; Assemble: nasm -f elf64 -o debugme.o debugme.asm
; Link:     ld -o debugme debugme.o

bits 64
default rel

section .data
msg: db 'Debug me!',10,0
len  equ $ - msg

section .text
global _start
_start:
    ; sys_write(1, msg, len)
    mov     rax, 1          ; syscall number
    mov     rdi, 1          ; fd=stdout
    lea     rsi, [msg]
    mov     rdx, len
    syscall
```

```
    ; exit(0)
    mov     rax, 60
    xor     rdi, rdi
    syscall
```

Now launch GDB and follow these steps:

1. Start GDB: `gdb ./debugme`
2. Set a breakpoint at the entry point: `break _start`
3. Run: `run`
4. The program stops at `_start`. Use `stepi` repeatedly to watch each instruction execute. Notice how `rax`, `rdi`, etc., change.
5. After the first `syscall` (for write), inspect the return value: `print $rax` (should be the number of bytes written).
6. Before the exit syscall, examine the message bytes: `x/s &msg` (if symbols loaded) or look at the disassembly to see the address loaded by `lea rsi, [msg]`.
7. Continue execution with `continue` or let it finish.

Debugging Without Symbols

If debugging information is not present, GDB cannot display source lines or symbol names. However, you can still use addresses. To find the address of `_start`:

```
objdump -t debugme | grep _start
```

Then use that address, e.g., `break *0x4000b0`. The disassembly view (`layout asm`) is extremely helpful in this case. Enter `layout asm` after starting GDB to see the disassembly and follow stepping visually.

12.6 Using GDB with GCC-Linked Programs (with libc)

If your program has `main` and is linked with `gcc`, the entry point is not your code but the C runtime. GDB still works flawlessly:

```
gdb ./prog
break main
run
```

This sets a breakpoint at `main` and starts execution, which stops at the first instruction of your function. All other commands remain identical.

12.7 Graphical Debugging with DDD

DDD (Data Display Debugger) provides a graphical front-end to GDB. After installation, launch it with:

```
ddd ./debugme
```

You can then set breakpoints by clicking, view registers in a separate window, and visually step through the code. The underlying GDB commands are displayed in a console at the bottom, helping you learn them.

Visual Studio Code Integration

As described in Chapter 8, VS Code can be set up to debug with GDB, giving you a modern IDE experience with source highlighting, variable inspection, and breakpoints managed from the editor. The `launch.json` configuration already shown is the recommended approach.

12.8 Advanced GDB Features

- **Conditional breakpoints:** `break _start if $rdi==5` – break only when RDI equals 5.
- **Watchpoints:** `watch variable` (symbol) or `watch *0x601000` – break when the memory location is modified. Useful for tracking memory corruption.
- **Command lists:** You can define a sequence of commands to execute at a breakpoint, e.g., `break _start` then `commands` and then enter commands ending with `end`.
- **Saving history:** GDB saves your command history. Use `set history save on` in `~/.gdbinit` to preserve it across sessions.

Tip

Create a `.gdbinit` file in your home directory with your favourite settings and aliases. For example, `set pagination off` stops GDB from paging long outputs; `set disassembly-flavor intel` ensures NASM-style syntax is displayed.

Putting It All Together

Debugging assembly code is a skill that pays back the effort many times over. With GDB (and optional front-ends), you have the power to stop time, look inside the CPU, and understand exactly why your program behaves the way it does. Practice with the commands shown here on every example in this book, and the debugger will become your most trusted companion.

Part III

NASM LANGUAGE BASICS

13

NASM Syntax and Program Structure

In This Chapter

The Netwide Assembler's source language is deliberately clear and consistent. Its design follows the Intel syntax tradition while adding a powerful preprocessor. This chapter explains the fundamental rules: the operand order, register naming, size specifiers, the overall layout of a valid assembly file, how labels and identifiers work, the use of comments, and the special role of the entry point definition. By the end of this chapter you will be able to write a well-structured NASM source file that is ready to assemble on Fedora 43. All explanations are based on the official NASM 2.16 documentation and the Intel/AMD manuals.

13.1 Intel Syntax Rules

NASM uses the Intel assembly syntax, which places the destination operand before the source operand. This is the same order used in Intel® architecture manuals. For example:

```
mov   rax, rcx          ; RAX = RCX
add   qword [rdx], 17h ; memory location += 0x17
```

The destination is always the first operand; the source(s) follow. This rule applies to all instructions that have multiple operands.

13.1.1 Operand Size Specification

In many instructions the operand size is implied by the register names: `rax` is 64 bits, `eax` is 32 bits, `ax` is 16 bits, `al` is 8 bits. When an operand is a memory reference without a register, the size must be explicitly stated using a size keyword:

```
mov   byte [rcx], 0xFF      ; write a single byte
mov   word [rcx], 0x1234    ; write a 16-bit word
```

```
mov   dword [rcx], 0x9ABCDEF1 ; write a 32-bit doubleword
mov   qword [rcx], 0x1122334455667788 ; write a 64-bit quadword
```

NASM also supports the older `SHORT`, `NEAR` and `FAR` overrides for jumps, but in 64-bit flat mode these are rarely needed.

Warning

When the destination is a memory reference, you **must** specify the size if the assembler cannot deduce it from a register source. For a constant immediate, there is no implicit size, so NASM will generate an "operation size not specified" error without the size keyword.

13.1.2 Register Names

Registers are named as defined by the `x86-64` architecture. The 64-bit general-purpose registers are `RAX, RBX, RCX, RDX, RSI, RDI, RBP, RSP, R8–R15`. Their sub-portions follow the standard naming: `EAX` (32-bit), `AX` (16-bit), `AL` (low 8 bits), `AH` (high 8 bits of AX). The new registers `R8–R15` offer 8-bit low forms `R8B–R15B`, 16-bit forms `R8W–R15W`, and 32-bit forms `R8D–R15D`. NASM recognizes all of them case-insensitively. The instruction pointer is `RIP`, and segment registers are `CS, DS, SS, ES, FS, GS`. The flags register is `RFLAGS`.

RIP-Relative Addressing and `rel` Keyword

In 64-bit mode, all memory references to labels in the `.text`, `.data`, and other sections should normally use RIP-relative addressing for position-independent code. NASM makes this easy with the `rel` keyword. By placing `default rel` at the top of the file, all memory references to labels that do not include a segment override become RIP-relative. For example:

```
default rel
mov   rax, [myvar]       ; actually `mov rax, [rel myvar]`
lea   rcx, [myvar]       ; loads effective address relative to RIP
```

Without `default rel`, you would have to write `[rel myvar]` explicitly. For position-independent code (recommended for shared libraries and modern executables), use RIP-relative addressing.

13.1.3 Immediate Operands and Sign Extension

Immediates can be written in decimal, hexadecimal (`0x` prefix), octal (`0q` prefix), binary (`0b` prefix), or as characters (`'A'`). NASM will choose the smallest encoding that fits, sign-extending if necessary for 32-bit or 64-bit destinations. For instance, `mov rax, -1` encodes a 32-bit immediate `0xFFFFFFFF` that is sign-extended to 64 bits, giving `0xFFFFFFFFFFFFFFFF`.

You can force a longer encoding using the `strict` keyword:

```
mov   rax, strict dword 0x1234   ; forces a 32-bit immediate encoding
```

13.2 Program Layout

A NASM source file for Linux consists of one or more sections, directives, and instructions. The minimal structure includes the `bits` directive, section declarations, and an entry-point label.

13.2.1 The `bits` Directive

This tells NASM the default processor mode. For 64-bit Linux code, always use `bits 64`. It must appear before any machine instructions.

```
bits 64
default rel
```

Failing to set `bits 64` will cause NASM to assume 16-bit mode, which will produce incorrect code or errors.

13.2.2 Sections

The executable is divided into sections, each with a defined purpose. The three most common are:

- `.text` – executable machine instructions.
- `.data` – initialized read-write data.
- `.bss` – uninitialized read-write data (zero-filled at load).

For read-only data, the standard section is `.rodata`. Section headers are declared with the `section` directive. NASM gives you the ability to set extra attributes such as alignment and access permissions, though the linker has the final say.

Code: Basic section declarations

```
section .text  progbits alloc exec  align=16
global _start
_start:
    ; code here
    ret

section .data  progbits alloc write align=8
greeting db 'Hello', 10, 0

section .bss  nobits alloc write align=8
buffer  resb  256
```

The `progbits` type indicates initialized data/code, while `nobits` is used for BSS. The `alloc` flag ensures the section is memory-resident, `exec` makes it executable, and `write` allows writes. For read-only data, use a section like:

```
section .rodata progbits alloc noexec read align=8
```

Tip

You can define custom sections with custom protection. For example, `section .myspecial progbits alloc write align=4096` creates a writable section with page alignment.

13.2.3 The `extern` and `global` Directives

To call functions from shared libraries (e.g., `printf`), you declare them with `extern`. To make a label visible to the linker (and to other object files), you use `global`.

```
extern printf
extern exit

global main
```

Any label not marked `global` is local to the object file.

13.3 Labels and Identifiers

Labels mark a position in the code or data section. They are identifiers followed by a colon.

13.3.1 Basic Labels

A label can contain letters, digits, underscores, periods, and dollar signs. It must start with a letter, underscore, period, or dollar sign. NASM is case-sensitive by default. Examples:

```
valid_label:
.another:
_myFunc:
$$special:
```

Labels that start with a period are *local labels* scoped to the most recent non-period label. This allows reuse of simple names like `.loop` inside different functions.

Code: Local labels in action

```
_start:
    mov   rdi, 5
    .loop:              ; local to '_start'
        dec   rdi
        jnz   .loop

    call  another_func
    ; exit...

another_func:
    mov   eax, 1
    .loop:              ; different label, same name allowed
        shl   eax, 1
        cmp   eax, 100
        jb    .loop
    ret
```

13.3.2 Labels on Data

Data labels are used to refer to the address of variables:

```
myVar   dq   0x1234
message db   'Hello', 0
```

These labels can be used in RIP-relative addressing as described.

13.3.3 Identifier Escaping

If an identifier collides with a NASM reserved word, you can surround it with back-ticks `` ` `` to force it to be treated as an identifier. This is rarely needed.

13.4 Comments

Comments are essential for documenting assembly code. NASM uses the semicolon (`;`) as the comment character. Everything from the semicolon to the end of the line is ignored.

```
; This entire line is a comment
mov   rax, rbx    ; copy RBX into RAX
```

There is no multi-line comment directive; you simply start each comment line with a semicolon. The preprocessor supports block comments via `%comment / %endcomment`, but these are seldom used in practice.

```
%comment
  This is a multi-line comment.
  It will not be processed.
%endcomment
```

For normal development, single-line comments are sufficient.

13.5 Entry Point Definition

In a Linux executable created from NASM, the entry point is the label where the kernel begins execution after loading the program. There are two common scenarios:

Pure Assembly with `_start`

If you link directly with `ld` without the C library, use the label `_start` and mark it `global`:

```
section .text
global _start
_start:
    ; ... use syscall instructions to do I/O and exit
    mov     eax, 60       ; sys_exit
    xor     edi, edi      ; exit code 0
    syscall
```

The linker automatically uses `_start` as the entry point. The program receives no arguments from the C runtime; to access command-line arguments, you must read the initial stack layout directly.

Linking with the C Library and `main`

If you plan to call C functions like `printf`, define a `main` function and link with `gcc` (or `ld` with appropriate startup files). The C runtime initialisation calls `main` as a normal function, passing `argc` and `argv`:

```
section .text
global main
extern printf
```

```
main:
    sub     rsp, 8                  ; align stack
    lea     rdi, [rel message]
    xor     eax, eax
    call    printf
    add     rsp, 8
    xor     eax, eax                ; return 0
    ret
```

Link with `gcc -no-pie -o prog prog.o` to get a working executable. The entry point of the final executable will be the C library's startup code, which eventually calls `main`.

Custom Entry Point Names

You can specify a different entry point label with `--entry` when linking directly with `ld`:

```
ld -o prog prog.o --entry myStart
```

Warning

In a pure assembly program that defines `_start`, you must explicitly terminate the process by calling the `sys_exit` system call. Simply executing `ret` from the entry point causes a crash because there is no return address on the stack.

Complete Example with Multiple Sections

Below is a full NASM listing that demonstrates all the above concepts: syntax, sections, labels, comments, and a proper entry point. It uses a system call to write a message to stdout, which works on any Linux system without external libraries.

Code: listing13-1.asm — Full syntactic demo

```
; listing13-1.asm
; Assemble: nasm -f elf64 -o listing13-1.o listing13-1.asm
; Link:     ld -o listing13-1 listing13-1.o

bits 64
default rel

section .data
msg     db  'NASM syntax test passed.', 10
len     equ $ - msg

section .bss
stdout  resq 1
written resq 1

section .text
global _start
_start:
    ; ssize_t write(int fd, const void *buf, size_t count)
    mov     eax, 1              ; syscall number for SYS_write
```

```
    mov     edi, 1              ; fd = stdout
    lea     rsi, [msg]          ; buf
    mov     rdx, len            ; count
    syscall

    ; void _exit(int status)
    mov     eax, 60             ; SYS_exit
    xor     edi, edi            ; status = 0
    syscall
```

The program assembles, links, and prints a success message. Every syntactic element — `bits`, `default rel`, `section`, `global`, labels, comments, and the entry point — is used as described.

Putting It All Together

Mastering NASM's syntax and the structure of an assembly file is the first concrete step toward becoming a productive Fedora 43 assembly programmer. The rules are few and logical, and they give you a direct line to the machine. The next chapter will explore the data types and constants you can use to build more complex data structures.

14

Data Definition and Directives

In This Chapter

Assembly language gives the programmer direct control over the layout of memory. This chapter explains every directive NASM provides to declare initialized and uninitialized data, from single bytes to packed structures, and how to control where and how that data lands in the final executable. You will learn the standard data definition keywords (`DB`, `DW`, `DD`, `DQ`), the syntax for constants and expressions, the rules that govern section alignment, and how to create read-only data sections that the Linux memory manager will protect. Mastering these directives is the foundation for building complex data structures that interface cleanly with the Linux system-call interface and C libraries. All content is based on the NASM 2.16 manual and the ELF specification.

14.1 DB, DW, DD, DQ

The primary directives for reserving and initializing memory are `DB` (Define Byte), `DW` (Define Word), `DD` (Define Doubleword), and `DQ` (Define Quadword). Each places a series of values of the given size into the current section at the current position.

Define Byte (DB)

`DB` allocates one or more bytes, optionally initialized. Values can be numeric constants, character constants, or strings.

```
section .data
byte1   db  0x1A              ; single byte initialized to 0x1A
byte2   db  0, 1, 2, 3        ; four bytes
string  db  'Hello, world!', 10, 0   ; null-terminated string with newline
```

When a string is supplied, each character is placed sequentially in memory. No automatic null

terminator is appended unless you include the final `0`. A backslash can be used to include ASCII control characters, but NASM prefers the comma-separated numeric form.

Define Word (DW)

`DW` allocates 16-bit words, which on `x86-64` are stored in little-endian order.

```
section .data
word1   dw  0x1234              ; a single 16-bit word
words   dw  100, 200, 300       ; three words
```

Define Doubleword (DD)

`DD` allocates 32-bit doublewords. This is the native size for many C `int` types and frequently used for array indices.

```
section .data
dword1  dd  0x9ABCDEF1          ; a dword
dwords  dd  1, 2, 3, 4, 5       ; array of five dwords
```

Warning

Storing a label's address with `dd` only captures a 32-bit offset, which may be truncated. In 64-bit code, always use `dq` for pointers unless you are certain the address fits in 32 bits (which is rare on modern Linux).

Define Quadword (DQ)

`DQ` allocates 64-bit quadwords. Use it for 64-bit integers and pointers.

```
section .data
qword1  dq  0x0123456789ABCDEF    ; a 64-bit immediate
pointer dq  my_function           ; full 64-bit address
zeros   dq  0                     ; a zero-initialized quadword
```

Additional Define Directives

NASM also provides `DT` (Define Ten-byte), `DO` (Define Oword, 16 bytes), `DY` (Define Yword, 32 bytes), and `DZ` (Define Zword, 64 bytes) for floating-point and SIMD data, but these are rarely needed in user-mode applications. They follow the same pattern.

Code: listing14-1.asm — Demonstrating data directives

```
; listing14-1.asm
bits 64
default rel

section .data
b_data  db  0x41, 0x42, 0x43          ; three bytes
w_data  dw  0x1234, 0x5678            ; two words
d_data  dd  1, 2, 3                   ; three dwords
q_data  dq  0xFFFFFFFFFFFFFFFF, 42     ; two qwords
str     db  'NASM data test', 0
```

After assembling, these labels can be referenced via RIP-relative addressing:

```
mov   al, [rel b_data]          ; load first byte
mov   rax, [rel q_data + 8]     ; load second quadword (42)
```

14.2 Constants

Constants are symbolic names for values that are replaced at assembly time. NASM provides several mechanisms: `equ`, `%define`, and expression assignment.

EQU Directive

`equ` creates an immutable symbolic constant. Its value is evaluated once and cannot be redefined.

```
BUFFER_SIZE  equ  1024
MAX_COUNT    equ  BUFFER_SIZE / 4
SYS_EXIT     equ  60
```

Constants defined with `equ` can be used anywhere an immediate number is expected. They do not occupy memory; they are purely preprocessor substitutions.

Code: Using equ for system call numbers

```
SYS_READ  equ 0
SYS_WRITE equ 1
SYS_EXIT  equ 60
```

%define and \%xdefine

The preprocessor directive `%define` defines a textual macro that can take parameters and can be redefined. `%xdefine` expands the value at definition time, while `%define` expands at invocation time.

```
%define PAGE_SIZE 4096
mov   eax, PAGE_SIZE            ; becomes mov eax, 4096
```

`%define` is more powerful for multi-line or parameterized macros; for numeric constants, `equ` is clearer.

Expression Evaluation

NASM evaluates constant expressions at assembly time. The result must be a known numeric value. All arithmetic and bitwise operators are available:

```
ALIGN_SIZE   equ  16
PADDED_SIZE  equ  ((SIZE + ALIGN_SIZE - 1) / ALIGN_SIZE) * ALIGN_SIZE
```

Any label address is a relocatable value, not a constant. Subtracting two labels in the same section yields a constant (the difference in bytes):

```
msg_start  db 'Hello'
msg_len    equ $ - msg_start    ; constant = 5
```

`$` denotes the current position in the section. This is the standard way to compute string lengths.

14.3 Alignment

Modern `x86-64` processors perform best when data is aligned to its natural size. NASM provides the `align` and `alignb` directives to insert padding bytes until the desired alignment is reached.

ALIGN and ALIGNB

`align N` pads with NOP instructions in code sections or with zero bytes in data sections (depending on context). `alignb N` always pads with zero bytes. The argument is a power of two; for instance, `align 16` aligns to a 16-byte boundary.

Code: Data alignment example

```
section .data
align 8                     ; ensures next label is 8-byte aligned
my_qword  dq  0x12345678

align 4                     ; 4-byte alignment for dword array
my_array  dd  10, 20, 30
```

In code sections, aligning the start of hot loops can improve fetch and decode throughput:

```
section .text
align 16
my_loop:
    inc   rax
    cmp   rax, rcx
    jb    my_loop
```

Alignment in .bss Section

For uninitialized data, the `resb`, `resw`, `resd`, `resq` directives reserve space without specifying initial values. Alignment can be enforced with `alignb`:

```
section .bss
alignb 16
buffer  resb  4096          ; buffer aligned to 16-byte boundary
```

Tip

The `align` directive does not impose any constraint on the final section alignment in memory — that is determined by the linker and the section characteristics. However, using `align` ensures that within the section, the offset is a multiple of the specified alignment, which the linker respects when combining sections. Explicitly aligning critical data is a good habit, especially when dealing with SIMD or structures that must match hardware alignment requirements.

14.4 Initialization Rules

NASM separates data definition into two categories: **initialized** and **uninitialized**. The distinction determines whether the bytes are stored in the executable file and how the loader treats them.

Initialized Data Directives

`DB`, `DW`, `DD`, `DQ`, etc., are initialized data directives. They must appear in a section that is marked as initialized, such as `.data` (or any section with the `progbits` type). The values you provide are placed directly into the file image.

Sections marked as `data` (read-write) or `rodata` (read-only) can hold initialized data. You can mix multiple data sizes in the same section.

```
section .data
single  db  0x55
many    dw  1,2,3,4
```

Uninitialized Data (BSS) Directives

Uninitialized data is declared with `RESB`, `RESW`, `RESD`, `RESQ`, `REST`, `RESO`, `RESY`, `RESZ`. These directives reserve space but do not store any initial values in the object file. The section must be declared with the `nobits` type (as in the standard `.bss`).

Code: BSS reservation

```
section .bss
buffer    resb  512         ; 512 bytes, zero-filled at load
handles   resq  4           ; four quadwords (32 bytes)
counters  resd  2           ; two dwords
```

When the operating system loads the executable, the BSS section is allocated and zero-filled. Any data stored there initially is zero, but the programmer must not rely on this behaviour without explicit zero-initialization for other memory types; the loader does zero the BSS, however.

Warning

Do not place `RES` directives in a `.data` section; they belong in `.bss` or another `nobits` section. Placing them in an initialized data section will cause NASM to generate incorrect section headers, potentially leading to linker errors or larger file sizes.

Combining Initialized and Uninitialized Data

In NASM, a section can be either `progbits` or `nobits`, not both. If you need a section with some initialized data and some reserved space, define two separate sections, or use a standard section with all initialized data and a separate BSS section for uninitialized.

14.5 Read-only vs Writable Data

The Linux memory manager enforces page-level protection based on segment permissions defined by the ELF program headers, which are built by the linker. NASM allows you to specify the intended access permissions for a section, which the linker translates into appropriate ELF section flags.

Default Section Permissions

- `.text` — Executable and readable, not writable. Attempting to write to it at runtime causes a segmentation fault.
- `.data` — Writable and readable. Not executable (modern kernels enforce NX).
- `.bss` — Writable and readable (merged into the data segment at load).
- `.rodata` — Readable only (if explicit). Used for constant strings and tables.

NASM sets the section characteristics automatically based on the section type keywords you provide. For a custom read-only data section, you can declare:

```
section .const progbits alloc noexec read align=8
```

The `read` keyword (and absence of `write`) tells the assembler to mark the section as non-writable. The linker will set the ELF segment permissions accordingly. Any attempt to modify data in this section will cause a segmentation fault, which is a good way to protect constants.

Practical Example: Read-Only vs Writable

Below is a small program that demonstrates read-only and writable data sections. It attempts to write to a read-only section to show the protection; obviously, this will crash, but it illustrates the concept.

Code: listing14-2.asm — Read-only and writable data sections

```
; listing14-2.asm
bits 64
default rel

section .const progbits alloc noexec read align=8
ro_msg  db 'This is read-only data.', 0

section .data
rw_msg  db 'This is writable data.', 0

section .text
global _start
_start:
    ; write the writable message to stdout (just to have output)
    mov     eax, 1            ; SYS_write
    mov     edi, 1            ; stdout
    lea     rsi, [rw_msg]
    mov     rdx, 26
    syscall

    ; try to write into the writable section (OK)
    mov     byte [rel rw_msg], 'X'

    ; uncommenting the line below would crash with a segmentation fault:
    ; mov    byte [rel ro_msg], 'Y'

    ; exit
```

```
    mov     eax, 60
    xor     edi, edi
    syscall
```

The `.const` section is defined with `read` and `noexec`, so any write will cause a segfault. The `.data` section is writable. In a debugger, uncommenting the offending line and running the program will demonstrate the memory protection.

Controlling Section Characteristics in Detail

The full syntax for the `section` directive in NASM is:

```
section <name> <type> <attribute> ...
```

`<type>` is one of `progbits` (initialized), `nobits` (uninitialized). Attributes include `alloc` (allocate memory), `write` (writable), `exec` (executable), `read` (readable, implied by absence of write), `align=N`, and `share` (not used on Linux). Multiple attributes can be combined. The linker respects all of them and generates the corresponding ELF segment flags.

Note

Not all section attributes are supported by every linker. The `share` attribute, for example, is meaningless on Linux for regular executables. Check your linker's documentation for specific flags corresponding to NASM attributes.

Putting It All Together

Data definition in NASM is straightforward yet flexible. Use `DB`–`DQ` for initialized data, `RES` directives for zero-filled storage, `equ` and `%define` for symbolic constants, and `align` to maintain performance. Choose section attributes deliberately to enforce read-only protection and to group logically related data. These directives form the data backbone of every assembly program you will write. The next chapter will dive into the rich set of control flow instructions that operate on this data.

15

Labels, Symbols, and Constants

In This Chapter

Labels, symbols, and constants are the building blocks that give structure and meaning to assembly code. Labels mark locations in code and data; symbols allow cross-module references; constants replace magic numbers with descriptive names. This chapter details how NASM handles local and global labels, how the linker resolves symbols across object files and shared libraries, the versatile `EQU` directive, the use of `extern` to import functions from libc and other libraries, and the recommended naming conventions for maintainable projects. Every explanation is drawn from the NASM 2.16 manual, the System V AMD64 ABI, and the ELF specification.

15.1 Local and Global Labels

Labels are identifiers followed by a colon that represent the address of the next instruction or data item. Their visibility determines whether they can be referenced from other source files or only within the current one.

15.1.1 Global Labels

A label declared with the `global` directive is exported from the object file and becomes visible to the linker and to other object files. Any label that serves as a function entry point, a global variable, or a library routine must be declared global.

```
global main
main:
    sub   rsp, 8
    xor   eax, eax
    ret
```

When using `gcc`, the symbol `main` must be global. For pure assembly programs, the entry point `_start` must be global.

Multiple object files can reference the same global label. For example, a math library can export an `add_two` function:

Code: mathlib.asm — Exporting a global function

```
; mathlib.asm
bits 64
default rel

section .text
global add_two
add_two:
    lea   rax, [rdi + rsi]
    ret
```

Another module can then import it via `extern`:

```
extern add_two
call add_two
```

The linker resolves the cross-reference, patching the call with the correct address.

Warning

Global labels must have unique names across all object files within a single linking session. If two modules define the same global label, the linker will issue a duplicate symbol error. Use descriptive, project-wide unique names.

15.1.2 Local Labels

Local labels begin with a period (`.`) and have scope limited to the most recent non-local label (one without a leading period). This allows you to reuse common names like `.loop` or `.done` inside different functions without fear of collision.

Code: listing15-1.asm — Local label scoping

```
; listing15-1.asm
bits 64
default rel

section .text
global main
main:
    sub   rsp, 8
    mov   ecx, 5

    .loop:
        dec   ecx
        jnz   .loop          ; refers to main's .loop
```

```
    call  function2

    xor   eax, eax
    add   rsp, 8
    ret

function2:
    mov   eax, 3
    .loop:                  ; different .loop, local to function2
        shl   eax, 1
        cmp   eax, 100
        jb    .loop
    ret
```

In this example, the two `.loop` labels are completely independent; the first is referenced from `main`, the second from `function2`. NASM automatically resolves each to the nearest parent label.

Local labels cannot be declared global. If you need to export a label, it must be a full identifier without a leading period.

Tip

Use local labels liberally for branches and short loops. They keep the code clean and avoid cluttering the global namespace with names like `main_loop_1`, `main_loop_2`, etc.

15.1.3 Special Label Forms

NASM supports numeric local labels: `1:`, `2:`, … up to `9:`. They are referenced as `.1`, `.2`, etc., but this style is less readable and rarely used in modern code. Period-prefixed local labels are preferred.

15.2 Symbol Resolution

Symbol resolution is the process by which the assembler and linker convert symbolic references into address values. Understanding it helps diagnose “undefined reference” errors.

15.2.1 Assembly-Time Resolution

When NASM processes a source file, it can resolve some symbols immediately:

- Labels within the same source file that are not external.
- Constants defined with `equ`.
- The difference between two labels in the same section (e.g., string length).

For instructions like `jmp .loop`, NASM knows the exact offset of `.loop` relative to the jump, so it can generate the correct displacement without involving the linker.

15.2.2 Link-Time Resolution

Symbols that are declared `global` (exported) or `extern` (imported) are left for the linker. The object file contains entries in its symbol table and relocation records. For an `extern` function call, NASM emits a relocation of type `R_X86_64_PC32` (for a 32-bit PC-relative call) or `R_X86_64_PLT32` (if the target is in a shared library and the linker should generate a PLT stub). When linking with `gcc` or `ld`, the necessary PLT stubs are created automatically.

The linker, when combining objects, assigns final virtual addresses to all sections and all labels. It then walks the relocation records and patches the machine code with the correct addresses. If a symbol cannot be found in any object or library, the linker emits an "undefined reference to 'symbol"' error.

15.2.3 Shared Library (Dynamic) Resolution

For symbols that come from a shared library (e.g., `printf`), the linker creates a Procedure Linkage Table (PLT) entry and a Global Offset Table (GOT) slot. The call in your code goes to the PLT stub, which loads the actual address from the GOT. At load time, the dynamic linker (`ld-linux.so`) resolves the symbol and updates the GOT. This is completely transparent; you simply declare `extern` and call the function.

15.3 EQU Directive

The `equ` directive defines a symbolic constant at assembly time. It does not allocate memory and is not a label; it simply instructs NASM to replace occurrences of the name with the defined value.

Code: Using EQU for system call numbers and sizes

```
SYS_WRITE  equ 1
SYS_EXIT   equ 60
BUFFER_SIZE equ 4096
```

These constants can be used anywhere an immediate is valid:

```
mov   eax, SYS_WRITE
mov   edi, 1
mov   rdx, BUFFER_SIZE
```

15.3.1 Expression Evaluation

The value of an `equ` can be a constant expression involving arithmetic and bitwise operators, provided all symbols in the expression can be evaluated at assembly time. For example:

```
TOTAL_SIZE equ BUFFER_SIZE * 100
ALIGNED    equ (TOTAL_SIZE + 15) & ~15
```

You cannot use forward-referenced labels in an `equ` expression, because NASM cannot evaluate their addresses during the first pass.

15.3.2 EQU vs %define

The preprocessor directive `%define` can also be used for constants, but it differs in that it performs textual substitution at invocation time, can be redefined, and can accept parameters. `equ` is evaluated once and is more appropriate for fixed numeric constants. Use `equ` for fixed values and `%define` for macros that need parameters or redefinition.

15.4 External Symbols

External symbols are names that are defined in another module or library. They are declared with `extern`. The NASM syntax is straightforward:

```
extern printf
extern exit
extern write
```

When you call an external function, NASM treats it like any label, generating a relative call. Because the symbol is undefined in the object file, the linker must resolve it.

15.4.1 Linking Against the C Library

If you use functions like `printf` or `exit`, you must link your object file with the C library. The easiest way is to use `gcc` as the linker:

```
nasm -f elf64 -o prog.o prog.asm
gcc -no-pie -o prog prog.o
```

The linker will find `printf` in `libc.so` and create the necessary PLT entries. If you link manually with `ld`, you need to specify `-lc` and the dynamic linker path.

15.4.2 Importing Functions from Other Object Files

If you have a function defined in another object file, declare it as `extern` in the caller and as `global` in the defining file. The static linker will resolve the reference without involving shared libraries.

Code: listing15-2.asm — Using external functions

```
; listing15-2.asm
; Assemble: nasm -f elf64 -o listing15-2.o listing15-2.asm
; Link:     gcc -no-pie -o listing15-2 listing15-2.o

bits 64
default rel

extern printf
extern exit

section .data
msg db 'External symbols work.', 10, 0

section .text
global main
main:
```

```
    sub     rsp, 8                  ; align stack
    lea     rdi, [rel msg]
    xor     eax, eax
    call    printf
    add     rsp, 8
    xor     edi, edi
    call    exit
```

The `extern` names must match the library names exactly. Unlike the Windows API, the C library on Linux uses standard names like `printf` and `exit`, without A/W suffixes.

15.5 Naming Conventions

Consistent naming helps maintain large assembly projects and avoids name clashes. NASM imposes the following rules, and we add best-practice guidelines.

15.5.1 Identifier Rules

An identifier may contain letters (`A-Z, a-z`), digits (`0-9`), and the special characters `_`, `$`, `#`, `@`, `~`, `?`, and `.`. It must begin with a letter, an underscore (`_`), a period (`.`), or a dollar sign (`$`). NASM is case-sensitive by default; `Main` and `main` are different symbols.

Warning

Case sensitivity is the default and matches the behaviour of the GCC compiler. Avoid mixing case variations of the same name.

15.5.2 Reserved Words

NASM's instruction mnemonics, register names, and directives are reserved. If you accidentally use a reserved word as a label, the assembler will emit an error. To force an identifier that collides with a reserved word, enclose it in back-ticks:

```
`push`:
    ret
```

This is rarely necessary and should be avoided for clarity.

15.5.3 Recommended Naming Styles

- **Global functions:** Use descriptive camelCase or PascalCase, e.g., `AllocateBuffer`, `PrintString`. Avoid overly generic names like `func1`.
- **Local labels:** Always use period-prefixed short names: `.loop`, `.retry`, `.done`. Never export them.
- **Constants:** Use ALL_CAPS with underscores, e.g., `MAX_BUFFER`, `SYS_WRITE`. This distinguishes constants from addresses at a glance.
- **Variables:** Lower-case with underscores or camelCase, e.g., `stdout_fd`, `bytes_written`. The chosen style should be consistent within the project.

- **Underscore prefix:** NASM does not add a leading underscore to global names by default. The symbol `main` stays `main`, matching GCC's output. The special entry point `_start` already begins with an underscore by convention and is expected to be defined that way.
- **Import thunks:** With shared libraries, the linker handles the PLT/GOT automatically. For advanced use, you can reference the GOT entries directly via `extern symbol@GOTPCREL`, but standard calls do not require this.

15.5.4 Example of Consistent Naming

The following snippet shows a small program that follows the recommended conventions.

Code: listing15-3.asm — Naming conventions in practice

```
; listing15-3.asm
bits 64
default rel

extern printf
extern exit

; Constants
SYS_WRITE    equ 1
STDOUT_FILENO equ 1

section .data
; Variables
hello_msg    db  'Hello from named world!', 10
HELLO_LEN    equ $ - hello_msg

section .text
global main
main:
    sub     rsp, 8

    lea     rdi, [rel hello_msg]
    xor     eax, eax
    call    printf

    add     rsp, 8
    xor     edi, edi
    call    exit
```

The constant is in uppercase, variables are in lowercase with underscores, and the entry point is the standard `main`.

Tip

Adopt a naming convention early and use it consistently. A well-named assembly program reads almost like pseudocode, which is a tremendous help when you return to it after months.

Putting It All Together

Labels, symbols, and constants are the glue that links the assembly language to the linker and to the programmer's own organisation. By understanding local scoping, external declarations, and the difference between `equ` and `%define`, you gain full control over how names are resolved and how your program communicates with the outside world. The next chapter moves into the powerful preprocessor, which allows you to define macros and conditional code that reduce repetition and increase clarity.

16

Sections (.text, .data, .bss, .rodata)

In This Chapter

An ELF executable on Linux is not a flat sequence of bytes; it is divided into sections and segments, each with a defined purpose and memory protection. The three core sections — `.text` for code, `.data` for initialised read-write data, and `.bss` for uninitialised data — are the pillars on which every NASM program is built. This chapter explores how to declare these sections, how to place code and data into them, the attributes that govern their behaviour during loading and execution, and how the Linux kernel maps them into the process address space. Every directive and attribute described here is derived from the NASM 2.16 documentation and the ELF specification.

16.1 Code Section

The `.text` section holds the executable machine instructions of your program. In NASM, you declare it with the `section` directive. The traditional declaration for 64-bit Linux is:

```
section .text  progbits alloc exec  align=16
```

`progbits` tells the assembler that this section contains "program bits" (initialised data). The `alloc` attribute ensures memory is allocated for it, and `exec` marks it as executable. The linker uses these flags to set the segment permissions; the final ELF program header will grant read and execute access to the text segment, while write access is denied. Any attempt to write into the `.text` section from user mode will cause a segmentation fault (NX protection).

Organising Functions

Multiple functions can be placed in `.text` sequentially. Labels mark entry points. The typical structure for a multi-function program looks like:

Code: Multiple functions in .text

```
section .text  progbits alloc exec  align=16

global main
global fibonacci

main:
    sub   rsp, 8
    ; call fibonacci etc.
    xor   eax, eax
    add   rsp, 8
    ret

fibonacci:
    ; input EDI = n, output EAX = fib(n)
    push  rbx
    mov   eax, 1
    ...
    pop   rbx
    ret
```

You can split code across several instructions with no performance penalty; the linker pads and aligns sections as specified.

Alignment of Code

Using `align=16` in the section declaration aligns the start of the section in memory to a 16-byte boundary. You can also use the `align` directive inside the section to align a particular label, like a tight loop:

```
align 16
.hot_loop:
    add   eax, [rcx]
    add   rcx, 8
    sub   edx, 1
    jnz   .hot_loop
```

This ensures the loop start is on a 16-byte boundary, which can improve the CPU's instruction fetch bandwidth.

Tip

In Linux, the `.text` section is mapped as **executable and readable** but not writable. Any attempt to modify code bytes at runtime (self-modifying code) will raise a `SIGSEGV`. To write into code, you must use `mprotect` to temporarily change the protection, a practice that is strongly discouraged for security reasons.

16.2 Data Section

The `.data` section stores initialised read-write variables. All data declared with `DB`, `DW`, `DD`, `DQ` is placed here. The standard declaration is:

```
section .data  progbits alloc write  align=8
```

The `write` attribute indicates that the section may be modified at runtime. The segment containing `.data` will be mapped as read-write (virtual memory protection `PROT_READ | PROT_WRITE`), without execute permission.

Variables and Arrays

Example of a `.data` section with various data types:

Code: listing16-1.asm — Data section example

```
section .data  progbits alloc write  align=8

; scalar variables
error_count dd   0
max_size    dq   4096

; strings
greeting    db   'Hello, Linux!', 10, 0
greeting_len equ  $ - greeting

; arrays
fib_table   dq   0, 1, 1, 2, 3, 5, 8, 13
num_fib     equ  ($ - fib_table) / 8
```

All data in `.data` is laid out contiguously in the file image and occupies virtual memory. The addresses are known at link time and may be referenced via RIP-relative addressing (when using `default rel` or PIC/PIE). In a non-PIE executable, absolute addresses are also valid.

Structures and Padding

To replicate C structs, you can use `DB`, `DW`, `DD`, `DQ` sequentially and manually enforce alignment. For example, a structure containing a byte, an int, and a pointer might be:

```
section .data
align 8
my_struct:
    db 0x01               ; byte flag
    times 3 db 0          ; pad to 4-byte boundary
    dd 100                ; 32-bit integer
    dq my_function        ; 64-bit pointer to a function
```

NASM's `istruc` macro can help with structured data, but the manual approach is often clearer for simple layouts.

16.3 BSS Section

The `.bss` section holds uninitialised static data. It is declared as:

```
section .bss  nobits alloc write  align=8
```

The type `nobits` indicates that the section does not consume file space. Its size is recorded, and when the executable is loaded, the kernel allocates pages and zero-fills them. The `.bss` section eventually resides in the same memory segment as `.data` and is mapped read-write.

Reserving Space

Use the `RES` family of directives to reserve space without initial values:

Code: listing16-2.asm — BSS reservation

```
section .bss  nobits alloc write  align=8

buffer      resb  1024          ; 1024 bytes, zero-filled at load
str_len     resq  1             ; a quadword counter
array       resd  64            ; 64 dwords (256 bytes)
```

You cannot write data to the BSS in the source file; it is only a reservation. The Linux kernel zeros all BSS memory before transferring control to the process entry point.

Warning

Relying on BSS being zero-filled is safe on Linux; the ELF specification guarantees that uninitialised data areas are zeroed. However, if you manually allocate memory using `mmap` or `sbrk`, you must explicitly clear it or use `MAP_ANONYMOUS` (which initialises pages to zero) and still be cautious with `brk` because the kernel may return pages that were previously used by the process.

The .rodata Section

For constant data that should never be modified, place it in the traditional read-only data section:

```
section .rodata  progbits alloc noexec read  align=8
const_msg db 'This is read-only', 10, 0
```

The `read` attribute (and absence of `write`) marks it as non-writable. The linker puts it in a segment without write permission, catching accidental writes with a segmentation fault.

16.4 Section Attributes

NASM's `section` directive accepts a rich vocabulary of attributes that map to ELF section flags and, via the linker, to segment permissions. The full syntax is:

```
section <name> <type> <attributes>
```

where `<type>` is one of `progbits` or `nobits`, and `<attributes>` can include:

- `alloc` – allocate memory for the section (always present for code/data).
- `exec` – executable (for `.text`).
- `write` – writable (for `.data`, `.bss`).

- `read` – readable (all sections are readable by default; adding `read` explicitly can be used to deny `write` in `.rodata`).
- `align=N` – alignment requirement (power of 2).
- `noexec` – explicitly mark as non-executable (redundant on modern kernels but good documentation).

Combining Attributes

You can create custom sections with fine-grained protection. For example, a section that is writable but not executable:

```
section .myvars progbits alloc write noexec align=8
```

The linker will place it in the data segment (read-write). The `noexec` attribute reinforces that it should never be executed.

Impact on the ELF Header

The section flags set by NASM are later used by the linker to build the `PT_LOAD` program headers. For instance, a typical executable will have two load segments: one for text (covering `.text` and `.rodata`) with `PF_R | PF_X`, and one for data (covering `.data` and `.bss`) with `PF_R | PF_W`. The assembler directives directly influence this segregation.

16.5 Memory Mapping

When the Linux kernel loads an executable, it creates virtual memory regions that correspond to the `PT_LOAD` segments. The mapping respects the segment alignment and protection flags derived from the section attributes.

Virtual Memory Layout of Sections

For a non-position-independent executable (built without `-pie`), the text segment typically starts at `0x400000` and the data segment a few pages higher. In a position-independent executable (PIE), the base address is randomised (ASLR). Within each segment, sections appear in the order specified by the linker script. A simplified map might look like:

```
0x400000 - 0x400fff  .text   (RX)
0x401000 - 0x401fff  .rodata (R)
0x402000 - 0x402fff  .data   (RW)
0x403000 - 0x403fff  .bss    (RW, zeroed)
```

The stack and heap occupy separate, higher-address regions.

Observing Section Mapping with Tools

You can examine the memory layout of a running process using:

```
cat /proc/<pid>/maps
```

or by inspecting the executable with `readelf -l prog`. This shows the load segments and their permissions, confirming that `.text` resides in an executable segment while `.data` and `.bss` are writable.

Custom Memory Mapping via mprotect

If you need to change permissions at runtime (e.g., to make a section writable temporarily), you can use the `mprotect` system call with page-aligned addresses. This is an advanced technique; for most assembly programs the standard section permissions are sufficient.

Note

The `.bss` section does not consume file space, but it increases the virtual memory size of the process. A large BSS (e.g., a 100 MiB buffer) will not make the executable huge on disk. The kernel commits physical pages on demand when the memory is actually touched.

Putting It All Together

Sections are the organisational framework of your NASM programs. Placing code, initialised data, and uninitialised data into `.text`, `.data`, and `.bss` ensures the Linux loader applies the correct protections and the linker can correctly resolve symbols. Understanding these mechanics gives you the power to create custom memory layouts, protect read-only data, and optimise for cache alignment. In the next chapter, you will learn how the assembler's expression system and address arithmetic interact with these sections.

17

Macros and Preprocessor Directives

In This Chapter

Assembly language is repetitive by nature. The same sequences of instructions appear in multiple places: function prologues, stack adjustments, system-call setups. NASM's powerful preprocessor lets you capture these patterns as macros, parameterised templates that expand to the required code at assembly time. Combined with conditional assembly, include files, and careful reuse strategies, macros can dramatically reduce source size and improve maintainability. This chapter covers the `%macro` and `%endmacro` mechanism, parameter handling, conditionals (`%if`, `%else`), the inclusion of external files, and best practices for writing reusable assembly components. All information is based on the NASM 2.16 manual.

17.1 Macro Definition

A macro is defined with the `%macro` directive and terminated with `%endmacro`. It can take parameters and contains any assembly statements or preprocessor directives.

Basic Macro Syntax

```
%macro my_macro 0
    ; body
%endmacro
```

The number after the macro name specifies the parameter count. A parameter-free macro expands to its body wherever it is invoked by name.

A Simple Code Snippet Macro

Code: A macro that pushes two registers

```
%macro push_rax_rbx 0
    push  rax
    push  rbx
%endmacro
```

Now writing `push_rax_rbx` in the source inserts the two push instructions. This is trivial but illustrates the point.

17.2 Parameters

Macros become truly useful when they accept parameters. Parameters are numbered `%1, %2, %3, ...` inside the macro body. The count in the `%macro` line can be exact or a range (e.g., `1-3` for one to three parameters).

Defining Parameterised Macros

Code: Parameterised macro for a system call

```
%macro sys_call 3        ; syscall number, arg1, arg2
    mov   eax, %1
    mov   edi, %2
    mov   esi, %3
    syscall
%endmacro
```

Invoked as `sys_call 1, 1, msg_address`, the macro expands to the necessary register assignments and a `syscall` instruction. This simplifies the code for making Linux system calls.

Rotating Parameters

The `%rotate` directive shifts parameter numbers left or right, enabling loops over arbitrary-length lists. For example, a `save_regs` macro that pushes multiple registers can be built with rotation:

Code: Pushing multiple registers using rotate

```
%macro push_regs 1-*
    %rep %0
        push  %1
        %rotate 1
    %endrep
%endmacro
```

Here `1-*` means at least one parameter, up to many. `%0` is the actual count. The `%rep` loop repeats for each parameter, and `%rotate 1` shifts the list so `%1` becomes the next parameter. Now `push_regs rax, rbx, rcx` pushes all three. This is a powerful technique.

Warning

Rotating a parameter list modifies the positional numbers, but the expansion occurs at assembly time. It does not generate loops at runtime; the assembler simply duplicates the instructions inside the `%rep` block.

Local Labels Inside Macros

When a macro contains branch labels, each expansion must use unique label names to avoid duplicates. NASM provides the `%%` prefix to define a label that is local to the macro expansion:

```
%macro sum_loop 1
    xor   eax, eax
    mov   ecx, %1
    %%.loop:
        add   eax, ecx
        loop  %%.loop
%endmacro
```

The label `%%.loop` will be replaced with a unique label like `..@1234.loop` in each expansion, preventing collisions. Use `%%` for any label defined inside a macro that may be expanded multiple times.

17.3 Conditional Assembly

Conditional assembly allows you to include or exclude code based on conditions evaluated at assembly time. It is useful for debug builds, selecting between algorithms based on constants, or guarding platform-specific code.

The `%if` Family

- `%if <expression>` — true if the expression is nonzero.
- `%ifdef <symbol>` — true if the symbol is defined.
- `%ifndef <symbol>` — true if not defined.
- `%ifidn <a>,<b>` — true if strings a and b are identical (case-sensitive).
- `%ifnum <x>` — true if x is a numeric constant.

They are terminated with `%endif`, with optional `%elif` and `%else`.

Code: Conditional assembly for debug output

```
%define DEBUG 1

%if DEBUG
    %define DUMP(msg)  call debug_print, msg
%else
    %define DUMP(msg)  ; nothing
%endif
```

When `DEBUG` is nonzero, `DUMP` expands to a debug function call; otherwise it expands to nothing. This toggles debug code without editing every occurrence.

Conditional Imports

You can conditionally define extern symbols depending on whether you need certain library features:

```
%ifndef USE_MATH
    %define USE_MATH 0
%endif

%if USE_MATH
    extern sin
    extern cos
%endif
```

Expression Testing

Use `%if` to test constants:

```
%assign BUFFER_SIZE 4096

%if BUFFER_SIZE > 2048
    extern large_alloc
%else
    extern small_alloc
%endif
```

The assembler selects the appropriate external function based on the buffer size.

17.4 Include Files

The `%include` directive inserts the contents of another file at the current position, exactly as if the text were copied. This enables sharing macros, constants, and external declarations across many source files.

Basic Inclusion

```
%include "syscalls.inc"
```

A common practice is to place all system-call number constants and common utility macros into a single file, then include it in each module.

Code: A typical include file: ‘syscalls.inc‘

```
; syscalls.inc - common Linux system call numbers
%ifndef SYSCALLS_INC
%define SYSCALLS_INC

SYS_READ   equ 0
SYS_WRITE  equ 1
SYS_EXIT   equ 60
STDOUT     equ 1
```

```
STDIN      equ 0

%endif
```

The include guard (`%ifndef SYSCALLS_INC`) prevents multiple inclusions and the resulting duplicate symbol errors. Place this file in an `include` directory and reference it with a relative path.

Including Macros

A separate file can contain reusable macros. For instance, `macros.inc` might have:

Code: macros.inc — Function prologue/epilogue macro

```
%macro prologue 1
    push  rbp
    mov   rbp, rsp
    sub   rsp, %1
%endmacro

%macro epilogue 0
    mov   rsp, rbp
    pop   rbp
    ret
%endmacro
```

Then in a source file:

```
%include "macros.inc"

my_func:
    prologue 0x30
    ; ... body ...
    epilogue
```

This drastically reduces boilerplate.

Tip

Use include files to build a personal library of NASM macros that mirror C-library functions. For example, a `print_string` macro that encapsulates the entire `write` system call sequence.

17.5 Code Reuse

Macros and includes are the primary mechanisms for code reuse in assembly. The preprocessor allows you to create abstractions that generate optimised code without incurring the overhead of a function call.

Function-like Macros vs. Subroutines

A macro that expands to a sequence of instructions avoids the `call` / `ret` overhead, but each expansion increases code size. A subroutine (a regular `call`) centralises the logic but costs a call

and return. For tiny, frequently used sequences, macros are ideal. For larger blocks, subroutines save space.

Code: Comparing a macro and a subroutine for adding two numbers

```
; macro version
%macro add_macro 2
    lea   %1, [%1 + %2]
%endmacro

; subroutine version (using System V calling convention)
global add_func
add_func:
    lea   eax, [rdi + rsi]
    ret
```

The macro uses the exact registers passed, while the subroutine uses the calling convention.

Building Reusable Utility Macros

Assemble a suite of macros for typical Linux tasks. For example, a macro that writes a string to stdout:

Code: write_string macro

```
%macro write_string 2       ; string_label, length
    mov   eax, SYS_WRITE
    mov   edi, STDOUT
    lea   rsi, [rel %1]
    mov   edx, %2
    syscall
%endmacro
```

Usage:

```
write_string hello_msg, HELLO_LEN
```

This macro assumes that `SYS_WRITE`, `STDOUT`, and `hello_msg` are properly defined. It saves the repetitive register setup.

Using `%push` and `%pop` for Contextual Assembly

NASM provides a stack-like context mechanism (`%push`, `%pop`) that can be used to track nested inclusions or macro states. While not needed for everyday macros, it enables advanced control structures, such as ensuring matching ends of block macros.

Code: Block macro using context stack

```
%macro begin_if 1
    %push if_context
    cmp   %1, 0
    je    %$else_label
%endmacro
```

```
%macro else 0
    jmp   %$end_label
    %$else_label:
%endmacro

%macro end_if 0
    %$end_label:
    %pop
%endmacro
```

The `%$` prefix generates a label unique to the context level, allowing nested if-else chains. This is an advanced technique; for most purposes, simple conditional jumps are clearer.

Organising Reusable Code

A well-organised NASM project uses a library directory with `include` files. Typical structure:

```
include/
    syscalls.inc
    macros.inc
    structs.inc
src/
    main.asm
    utils.asm
```

Each source file begins with:

```
%include "include/syscalls.inc"
%include "include/macros.inc"
```

This modular approach allows building different programs from the same set of utility macros and constants.

Preprocessor Best Practices

- **Guard all include files** with `%ifndef` checks to avoid double inclusion.
- **Document macros** with comments; a macro's parameters are not self-evident.
- **Minimise macro side effects.** A macro should not rely on the caller setting up registers unless those registers are documented as parameters.
- **Use local labels** (`%%`) inside macros to prevent duplicate symbol errors.
- **Keep macros simple.** If a macro grows beyond a few lines, consider making it a subroutine instead.
- **Use `%unmacro`** to remove a macro definition when it is no longer needed, though this is rarely necessary.
- **Test macros** thoroughly. Because they expand textually, errors can be cryptic; assemble with `-l listing.lst` to see the expanded output.

Warning

Over-using macros can make code unreadable and debugger unfriendly, since the debugger shows the expanded code, not the macro invocation. Strike a balance between convenience and clarity.

Putting It All Together

The preprocessor is one of NASM's most valuable assets. It allows you to write less, reuse more, and shield the rest of the code from trivial details. Macros, conditionals, and include files together enable a clean, modular assembly programming style that rivals higher-level languages in expressiveness while retaining full low-level control. The next chapter will shift focus to the actual execution flow, starting with unconditional and conditional jumps.

Part IV

CORE ASSEMBLY PROGRAMMING

18

Data Movement Instructions

In This Chapter

Moving data is the most fundamental operation any program performs. The x86-64 architecture provides a rich set of instructions to transfer bytes, words, doublewords, and quadwords between registers and memory, to load effective addresses for pointer arithmetic, and to extend values with or without sign preservation. This chapter examines the essential data movement instructions: `MOV`, `LEA`, `MOVZX`, and `MOVSX`. Every description includes the precise operand rules, flag effects, encoding details where relevant, and complete NASM examples tested on Fedora 43. The material is drawn from the Intel® 64 and IA-32 Architectures Software Developer's Manual (Volumes 2A and 2B), the AMD64 Architecture Programmer's Manual, and the NASM 2.16 documentation.

18.1 MOV Instruction

`MOV` copies the source operand to the destination operand. It is the simplest and most heavily used instruction in the instruction set.

Syntax and Operand Forms

The general form is `MOV dest, src`. The following operand combinations are allowed:

- **Register to register:** `mov rax, rcx`
- **Immediate to register:** `mov eax, 42`
- **Immediate to memory:** `mov dword [rdx], 0x7F`
- **Register to memory:** `mov [rdx], rax`
- **Memory to register:** `mov rax, [rdx]`

Direct memory-to-memory moves are **not** permitted. To copy from one memory location to another, you must use an intermediate register or a string instruction.

Size Specification

When both operands are registers, NASM deduces the operand size from the register name. When one operand is a memory reference and the other is an immediate, you must explicitly state the size:

```
mov byte  [rcx], 0x01
mov word  [rcx], 0x0202
mov dword [rcx], 0x03030303
mov qword [rcx], 0x0404040404040404
```

Failing to do so results in an "operation size not specified" error.

Flag Effects

`MOV` does **not** affect any status flags. It silently moves data without altering the processor state beyond the destination.

RIP-Relative Addressing for Global Data

Modern Linux executables are often position-independent (PIE). To access labels in `.data` and `.bss`, use RIP-relative addressing. With `default rel` at the top of the file, `mov eax, [myvar]` automatically becomes `mov eax, [rel myvar]`. This is the standard way to access global variables.

Code: listing18-1.asm — Basic MOV examples

```
; listing18-1.asm
; Assemble: nasm -f elf64 -o listing18-1.o listing18-1.asm
; Link:     ld -o listing18-1 listing18-1.o

bits 64
default rel

section .data
value1  dq  0xAAAAAAAAAAAAAAAA
value2  dq  0

section .text
global _start
_start:
    ; register to register
    mov   rax, 0x1234
    mov   rbx, rax

    ; immediate to memory
    mov   dword [rel value2], 0xBBBBBBBB

    ; memory to register
    mov   rcx, [rel value1]
```

```
    ; register to memory
    mov   [rel value1], rcx

    ; exit(0)
    mov   eax, 60         ; SYS_exit
    xor   edi, edi
    syscall
```

The above program runs without any external library calls; it simply exercises MOV and exits. You can debug it to verify that the values are written and read correctly.

Common Linux Usage

Setting up arguments for system calls often relies on `MOV`. For example, before calling `sys_write`, you move the file descriptor into `RDI`, the buffer pointer into `RSI`, and the size into `RDX`. These are all register loads via `MOV` (or `LEA`).

18.2 LEA Instruction

`LEA` (Load Effective Address) computes the offset of a memory operand and stores that offset in a register. It does **not** read from memory; it only performs address arithmetic.

Syntax and Operation

```
lea   dest_register, [src_address_expression]
```

The destination must be a general-purpose register. The source is any valid memory operand. The processor calculates the effective address and writes the result to the destination.

Using LEA for Address Calculation

In position-independent 64-bit Linux code, the primary use of `LEA` is to obtain the pointer to a label via RIP-relative addressing:

```
lea   rsi, [rel message]    ; rsi now points to message
```

This replaces the older `mov rsi, offset message` which is not valid in 64-bit mode for PIE.

Arithmetic with LEA

Because `LEA` can combine base, index, scale, and displacement in one operation, it can perform up to three additions and a multiplication without using ALU execution ports (it uses the address generation unit). For instance:

```
lea   rax, [rcx + rdx*4 + 128]   ; rax = rcx + (rdx * 4) + 128
```

This is a fast way to compute an offset into an array of dwords.

Flag Effects

LEA does not modify any flags. It is purely an address computation; any overflow or carry is ignored, and the result is truncated to 64 bits.

Code: listing18-2.asm — LEA for address calculation and arithmetic

```
; listing18-2.asm
; Assemble: nasm -f elf64 -o listing18-2.o listing18-2.asm
; Link:     ld -o listing18-2 listing18-2.o

bits 64
default rel

section .data
array   dq  10, 20, 30, 40, 50

section .text
global _start
_start:
    ; load address of array
    lea   rcx, [rel array]

    ; compute address of array[3]: base + index*8
    mov   edx, 3
    lea   rax, [rcx + rdx*8]    ; RAX now points to array[3]

    ; load the value
    mov   rbx, [rax]            ; RBX = 40

    ; Arithmetic: compute 5*rcx + 10 using LEA (scale limited to 1,2,4,8)
    lea   rax, [rcx + rcx*4 + 10]  ; rax = rcx + rcx*4 + 10 = 5*rcx + 10

    ; exit(0)
    mov   eax, 60
    xor   edi, edi
    syscall
```

Tip

Use LEA to avoid separate MOV and arithmetic instructions. A single LEA can replace a MOV + SHL + ADD sequence, reducing code size and often improving speed.

18.3 MOVZX / MOVSX

When moving a smaller-sized source into a larger-sized destination, you must decide whether to zero-extend or sign-extend the value. MOVZX (Move with Zero-Extension) fills the upper bits with zeros. MOVSX (Move with Sign-Extension) copies the source's sign bit into all upper bits.

Supported Sizes

- `MOVZX r32, r/m8` — byte to dword (upper 32 of r64 are zeroed because writing r32 zero-extends to 64 bits).
- `MOVZX r32, r/m16` — word to dword.
- `MOVZX r64, r/m8` — byte to qword.
- `MOVZX r64, r/m16` — word to qword.
- `MOVSX r32, r/m8` — byte to dword (sign-extended).
- `MOVSX r32, r/m16` — word to dword.
- `MOVSX r64, r/m8` — byte to qword.
- `MOVSX r64, r/m16` — word to qword.
- `MOVSX r64, r/m32` — dword to qword (use `movsxd` as alias).

Avoiding Redundant MOVZX

Because writing to a 32-bit register automatically zero-extends to the full 64-bit register, you rarely need an explicit zero-extension from dword to qword. For example:

```
mov   eax, dword [buffer]    ; zero-extends to RAX
```

Flag Effects

Neither `MOVZX` nor `MOVSX` affects the flags.

Practical Example: String Length with Zero Extension

When working with strings, you often load bytes into larger registers to compute lengths or perform comparisons.

Code: listing18-3.asm — MOVZX and MOVSX usage

```
; listing18-3.asm
; Assemble: nasm -f elf64 -o listing18-3.o listing18-3.asm
; Link:     ld -o listing18-3 listing18-3.o

bits 64
default rel

section .data
byte_data   db  0xFE
word_data   dw  0xFF80
dword_data  dd  0x12345678

section .text
global _start
_start:
    ; zero-extend byte
```

```
    movzx eax, byte [rel byte_data]    ; EAX = 0x000000FE, RAX zero-extended

    ; sign-extend byte
    movsx ecx, byte [rel byte_data]    ; ECX = 0xFFFFFFFE
    movsx rdx, byte [rel byte_data]    ; RDX = 0xFFFFFFFFFFFFFFFE

    ; zero-extend word
    movzx r8d, word [rel word_data]    ; R8D = 0x0000FF80, R8 = zero-extended

    ; sign-extend dword to qword
    movsxd r9, dword [rel dword_data]  ; R9 = 0x0000000012345678 (positive unchanged)
    ; For negative dword:
    mov   dword [rel dword_data], 0x80000000
    movsxd r10, dword [rel dword_data] ; R10 = 0xFFFFFFFF80000000

    ; exit(0)
    mov   eax, 60
    xor   edi, edi
    syscall
```

Warning

If you load a byte with `mov al, [mem]` and later use `RAX` as an address, the upper bits will contain whatever was in `RAX` before, not zero. Always explicitly extend when necessary, or clear the whole register first.

18.4 Memory Transfers

Moving data to and from memory is central to assembly programming. The `x86-64` provides a variety of addressing modes to make it efficient.

Loading and Storing

```
mov   rax, [rcx]            ; load 8 bytes
mov   [rsp+32], ebx         ; store 4 bytes
```

The size is determined by the register operand. For immediate stores, supply a size keyword.

Memory-to-Memory Copy

Because there is no memory-to-memory `MOV`, copy between two locations in two steps:

```
mov   rax, [src]
mov   [dst], rax
```

For larger blocks, the `REP MOVS` string instruction is efficient but requires proper setup of `RSI`, `RDI`, `RCX`, and the Direction Flag.

Alignment

Unaligned general-purpose integer loads/stores are allowed, but aligning to the natural size improves performance. Use `align` directives in sections.

Example: Copying a Structure

Copy 24 bytes using three quadword moves.

Code: listing18-4.asm — Memory copy example

```
; listing18-4.asm
; Assemble: nasm -f elf64 -o listing18-4.o listing18-4.asm
; Link:     ld -o listing18-4 listing18-4.o

bits 64
default rel

section .data
src_struct: db  1,0,0,0, 2,0,0,0, 3,0,0,0, 4,0,0,0, 5,0,0,0, 6,0,0,0
dst_struct: times 24 db 0

section .text
global _start
_start:
    lea   rsi, [rel src_struct]
    lea   rdi, [rel dst_struct]

    mov   rax, [rsi]
    mov   [rdi], rax
    mov   rax, [rsi+8]
    mov   [rdi+8], rax
    mov   rax, [rsi+16]
    mov   [rdi+16], rax

    mov   eax, 60
    xor   edi, edi
    syscall
```

18.5 Register Transfers

Zeroing a Register

The fastest and smallest way to set a 64-bit register to zero is:

```
xor   eax, eax          ; zeroes RAX, breaks dependency
```

This idiom is recognized by the processor and is preferred over `mov eax, 0`.

Moving Between Different Sizes

Use `MOVZX` / `MOVSX` as needed. Remember that a 32-bit move zero-extends the upper 32 bits of the 64-bit register.

Complete Example: String Conversion to Uppercase

This program uses `MOV`, `LEA`, `MOVZX` and memory operations to convert a string to uppercase and print it to stdout via the `write` system call.

Code: listing18-5.asm — Full data movement in action

```
; listing18-5.asm
; Assemble: nasm -f elf64 -o listing18-5.o listing18-5.asm
; Link:     ld -o listing18-5 listing18-5.o

bits 64
default rel

STDOUT  equ 1
SYS_WRITE equ 1
SYS_EXIT  equ 60

section .data
in_msg   db 'hello world', 10
in_len   equ $ - in_msg

section .bss
out_buf  resb 32

section .text
global _start
_start:
    ; Convert to uppercase
    lea     rsi, [rel in_msg]
    lea     rdi, [rel out_buf]
    mov     ecx, in_len

.loop:
    movzx   eax, byte [rsi]      ; load and zero-extend a byte
    cmp     al, 'a'
    jb      .skip
    cmp     al, 'z'
    ja      .skip
    sub     al, 32               ; to uppercase
.skip:
    mov     [rdi], al
    inc     rsi
    inc     rdi
    dec     ecx
    jnz     .loop

    ; Write the uppercase string
    mov     eax, SYS_WRITE
    mov     edi, STDOUT
    lea     rsi, [rel out_buf]
    mov     edx, in_len
    syscall

    ; exit(0)
    mov     eax, SYS_EXIT
    xor     edi, edi
    syscall
```

This program demonstrates repeated use of `MOV`, `MOVZX`, `LEA`, and memory transfers. When run, it

prints “HELLO WORLD”.

Putting It All Together

Data movement instructions are the foundation of every assembly language program. Understanding `MOV`, `LEA`, `MOVZX`, and `MOVSX`—their operand forms, size rules, and lack of flag effects—equips you to write efficient, correct code that interacts seamlessly with the Linux kernel via system calls. The next chapter builds upon this by introducing the arithmetic instructions that transform the data you have so carefully moved.

19

Arithmetic Instructions

In This Chapter

Arithmetic operations are at the heart of computation. The `x86-64` instruction set provides a comprehensive set of integer arithmetic instructions that operate on 64-, 32-, 16-, and 8-bit operands. This chapter examines the core arithmetic instructions: `ADD`, `SUB`, `MUL`, `IMUL`, `DIV`, `IDIV`, `INC`, `DEC`, and the associated flag behaviour. Every explanation includes the allowed operand forms, effects on the status flags, and complete NASM examples tested on Fedora 43. The descriptions are based on the Intel® 64 and IA-32 Architectures Software Developer's Manual (Vol. 2A), the AMD64 Architecture Programmer's Manual, and the NASM 2.16 documentation.

19.1 ADD and SUB

`ADD` and `SUB` are the most straightforward arithmetic instructions. They perform integer addition and subtraction, respectively.

Syntax

- `ADD dest, src` — `dest = dest + src`
- `SUB dest, src` — `dest = dest - src`

The destination and source must be the same size. Valid operand forms include `reg, reg`, `reg, mem`, `mem, reg`, and `reg, imm`.

Flag Effects

Both instructions modify `OF`, `SF`, `ZF`, `AF`, `PF`, and `CF` according to the result.

- **CF (Carry Flag):** Set on unsigned overflow (ADD) or borrow (SUB).
- **OF (Overflow Flag):** Set on signed overflow.
- **SF (Sign Flag):** Set to the most significant bit of the result.
- **ZF (Zero Flag):** Set if the result is zero.
- **AF (Auxiliary Carry), PF (Parity):** Used for BCD/odd parity.

Example: Basic Addition and Subtraction

Code: listing19-1.asm — ADD and SUB demonstration

```
; listing19-1.asm
; Assemble: nasm -f elf64 -o listing19-1.o listing19-1.asm
; Link:     ld -o listing19-1 listing19-1.o

bits 64
default rel

section .data
val_a   dq  100
val_b   dq  250

section .bss
result  resq 1

section .text
global _start
_start:
    mov     rax, [rel val_a]
    add     rax, [rel val_b]        ; RAX = 350
    mov     [rel result], rax

    mov     rcx, [rel val_b]
    sub     rcx, [rel val_a]        ; RCX = 150
    mov     [rel result], rcx

    ; exit(0)
    mov     eax, 60
    xor     edi, edi
    syscall
```

Using ADD/SUB for Pointer Arithmetic

```
lea     rsi, [rel array]
add     rsi, 8              ; point to next quadword
```

19.2 MUL and IMUL

MUL — Unsigned Multiply

`MUL src` multiplies the source by the implicit accumulator. The product width is double the operand size. For 64-bit: `MUL r/m64` → RDX:RAX = RAX * r/m64.

IMUL — Signed Multiply

`IMUL` has one-, two-, and three-operand forms. The two- and three-operand forms discard the upper half and set CF/OF if truncation occurs.

Flag Effects

`MUL` and single-operand `IMUL` set CF/OF if the upper half is non-zero. The other arithmetic flags become undefined.

Example: Multiplication

Code: listing19-2.asm — MUL and IMUL examples

```
; listing19-2.asm
; Assemble: nasm -f elf64 -o listing19-2.o listing19-2.asm
; Link:     ld -o listing19-2 listing19-2.o

bits 64
default rel

section .data
multiplicand dq  0x12345678
multiplier   dq  0x2

section .bss
prod_low  resq 1
prod_high resq 1

section .text
global _start
_start:
    ; unsigned multiply
    mov     rax, [rel multiplicand]
    mov     rcx, [rel multiplier]
    mul     rcx                     ; RDX:RAX = RAX * RCX
    mov     [rel prod_low], rax
    mov     [rel prod_high], rdx

    ; signed multiply (three-operand)
    mov     rax, [rel multiplicand]
    imul    rax, rax, 5             ; RAX = multiplicand * 5
    mov     [rel prod_low], rax

    ; exit(0)
    mov     eax, 60
    xor     edi, edi
    syscall
```

19.3 DIV and IDIV

Unsigned `DIV` and signed `IDIV` require a double-width dividend (RDX:RAX for 64-bit). The quotient goes into RAX, remainder into RDX.

Division Exceptions

A #DE exception occurs if the divisor is zero or the quotient overflows.

Preparing the Dividend

For unsigned, clear RDX with `xor edx, edx`. For signed, sign-extend RAX into RDX using `cqo`.

Flag Effects

Flags are undefined after division.

Example: Safe Division

Code: listing19-3.asm — Division with zero check

```
; listing19-3.asm
; Assemble: nasm -f elf64 -o listing19-3.o listing19-3.asm
; Link:     ld -o listing19-3 listing19-3.o

bits 64
default rel

section .data
dividend  dq  100
divisor   dq  7

section .bss
quotient  resq 1
remainder resq 1

section .text
global _start
_start:
    mov     rcx, [rel divisor]
    test    rcx, rcx
    jz      .exit

    ; unsigned division
    mov     rax, [rel dividend]
    xor     edx, edx
    div     rcx
    mov     [rel quotient], rax
    mov     [rel remainder], rdx

    ; signed division for negative dividend
    mov     rax, -100
    cqo                             ; sign-extend RAX into RDX
```

```
    idiv    rcx
    ; quotient = -14, remainder = -2

.exit:
    mov     eax, 60
    xor     edi, edi
    syscall
```

19.4 INC and DEC

Increment/decrement by 1. They affect all flags except CF.

Example

Code: listing19-4.asm — INC and DEC usage

```
; listing19-4.asm
bits 64
default rel

section .bss
counter resq 1

section .text
global _start
_start:
    mov     qword [rel counter], 10
.loop:
    inc     qword [rel counter]
    dec     qword [rel counter]
    cmp     qword [rel counter], 0
    jne     .loop

    mov     eax, 60
    xor     edi, edi
    syscall
```

19.5 Flags Behavior

The following table summarizes the effects on commonly tested flags.

Instruction	OF	SF	ZF	AF	PF	CF
ADD	*	*	*	*	*	*
SUB	*	*	*	*	*	*
CMP	*	*	*	*	*	*
MUL (1-op)	*	?	?	?	?	*
IMUL (1-op)	*	?	?	?	?	*
IMUL (2/3-op)	*	?	?	?	?	*
DIV	?	?	?	?	?	?
IDIV	?	?	?	?	?	?
INC	*	*	*	*	*	– (unchanged)
DEC	*	*	*	*	*	–

** = modified, ? = undefined, – = unchanged*

Example: Branching on Overflow

Code: listing19-5.asm — Jump if overflow

```
; listing19-5.asm
bits 64
default rel

section .data
val1  dq  0x7FFFFFFFFFFFFFFF
val2  dq  1

section .text
global _start
_start:
    mov     rax, [rel val1]
    add     rax, [rel val2]         ; signed overflow

    jo      .overflow               ; jump if overflow
    xor     edi, edi                 ; no overflow, exit 0
    jmp     .exit
.overflow:
    mov     edi, 1                   ; overflow occurred, exit 1
.exit:
    mov     eax, 60
    syscall
```

Putting It All Together

Arithmetic instructions are the building blocks of computation. By mastering ADD, SUB, MUL, IMUL, DIV, IDIV, INC, and DEC, and by understanding exactly how they set the flags, you can write robust arithmetic, multi-precision integer libraries, and performance-sensitive computational kernels. The next chapter covers logical and bit manipulation instructions, which complement arithmetic by allowing precise control over individual bits.

20

Bitwise and Logical Operations

In This Chapter

Manipulating individual bits is a fundamental skill in systems programming. The `x86-64` architecture provides a rich set of logical instructions to perform bitwise AND, OR, XOR, and NOT; shift instructions that move bits left or right with various fill strategies; rotate instructions that circulate bits through the carry flag; and bit test and manipulate instructions that operate on single bits. This chapter examines each of these instruction groups in detail, describing operand forms, flag effects, and practical uses in assembly programs on Fedora 43. Every explanation is verified against the Intel® 64 and IA-32 Architectures Software Developer's Manual (Vol. 2A), the AMD64 Architecture Programmer's Manual, and the NASM 2.16 documentation. Complete NASM examples accompany each section, all using the Linux system-call interface where I/O or process exit is required.

20.1 AND, OR, XOR, NOT

These instructions perform the basic Boolean operations. They take two operands (except NOT, which takes one) and can operate on registers or memory.

AND

`AND dest, src` performs a bitwise logical AND. Each bit of the result is 1 only if the corresponding bits of both operands are 1.

```
and   eax, 0x000000FF      ; keep low byte, clear others
and   byte [flag], 0xFE    ; clear bit 0
```

Uses: masking bits (clearing selected bits), testing if multiple bits are set (followed by `TEST` which is AND without writing result), and modulo operations if the divisor is a power of two (e.g., `and`

`eax, 15` computes eax mod 16).

OR

`OR dest, src` sets each bit of the result to 1 if either source bit is 1.

```
or    eax, 0x00000001      ; set bit 0
or    qword [mask], rax    ; combine flags
```

Uses: setting specific bits, merging flag fields.

XOR

`XOR dest, src` sets a bit to 1 if the corresponding bits differ.

```
xor   eax, eax             ; zero RAX efficiently
xor   byte [lock], 1       ; toggle bit 0
```

Uses: zeroing a register (recognized by the CPU as a zeroing idiom, breaking dependencies), toggling bits, implementing simple encryption (XOR cipher), and performing conditional swaps without a temporary register.

NOT

`NOT dest` flips all bits of the operand (one's complement). It does not affect any flags.

```
mov   eax, 0xFFFF0000
not   eax                   ; EAX = 0x0000FFFF
```

Uses: creating bitmasks, inverting Boolean values.

Flag Effects

- **AND, OR, XOR:** The OF and CF flags are cleared, SF, ZF, and PF are set according to the result; AF is undefined.
- **NOT:** No flags are affected.
- **TEST:** Same as AND but does not modify destination. It only updates SF, ZF, PF; OF=CF=0.

Practical Masking Example

The following program reads a byte, extracts the high nibble using AND and shift, and sets a flag using OR. It exits with that result as the exit code.

Code: listing20-1.asm — Logical operations

```
; listing20-1.asm
; Assemble: nasm -f elf64 -o listing20-1.o listing20-1.asm
; Link:     ld -o listing20-1 listing20-1.o

bits 64
default rel

section .data
```

```
value   db  0xAB
result  db  0

section .text
global _start
_start:
    mov     al, [rel value]         ; AL = 0xAB
    and     al, 0xF0                ; isolate high nibble -> 0xA0
    shr     al, 4                   ; now AL = 0x0A
    or      byte [rel result], al   ; result = 0x0A

    ; toggle bit 7 of result
    xor     byte [rel result], 0x80

    ; exit with the result as return code
    movzx   edi, byte [rel result]
    mov     eax, 60                 ; SYS_exit
    syscall
```

After running, `echo $?` will print 138 (0x8A), confirming the final result 0x8A.

20.2 Shift Operations

Shift instructions move bits left or right by a specified count, filling the empty positions with zeros or the sign bit, or allowing multi-precision shifts.

Logical Shifts: SHL, SHR

- `SHL dest, count` — shift left, filling with zeros. Same instruction as `SAL` (Shift Arithmetic Left).
- `SHR dest, count` — shift right, filling with zeros.

The count can be an immediate or the CL register. The result is stored in `dest`. For 64-bit shifts, use `SHL r64, imm8` or `SHL r64, CL`.

```
mov   rax, 1
shl   rax, 10             ; RAX = 1024 (1 << 10)
shr   rax, 3              ; RAX = 1024 / 8 = 128
```

Arithmetic Shifts: SAL, SAR

- `SAL` is identical to `SHL`.
- `SAR dest, count` — shift right, filling with the sign bit (preserving the sign of a signed integer).

```
mov   eax, -16
sar   eax, 2              ; EAX = -4 (sign extended)
```

Double-Precision Shifts: SHLD, SHRD

These instructions shift a destination operand left or right, pulling in bits from a source operand. They are useful for multi-precision arithmetic or bit-stream extraction.

- `SHLD dest, src, count` — shift `dest` left by count, filling from the left with bits from `src`.
- `SHRD dest, src, count` — shift `dest` right, filling from the right with bits from `src`.

The count must be an immediate or CL. The source is not modified.

Example: combine two 32-bit values into a 64-bit value:

```
mov   eax, high_part
mov   ebx, low_part
shld  eax, ebx, 16        ; eax shifted left, filled from high bits of ebx
```

Flag Effects

- CF: last bit shifted out. For shifts with count > 1, CF is the last bit shifted out (if count=0, flags unchanged).
- OF: defined only for single-bit shifts; set if the sign bit changed (SHL/SAL) or if the top two bits differ (SAR). For multi-bit shifts, OF is undefined.
- SF, ZF, PF: set according to result.
- AF: undefined.

Caution

The shift count is masked: for 64-bit destinations, only the low 6 bits of the count are used (0–63). Writing `shl rax, 65` actually shifts by 1. This is documented behavior, but if you rely on it, ensure it is what you intend.

Shift Example

Code: listing20-2.asm — Shifts and bit extraction

```
; listing20-2.asm
; Assemble: nasm -f elf64 -o listing20-2.o listing20-2.asm
; Link:     ld -o listing20-2 listing20-2.o

bits 64
default rel

section .data
packed  dd  0x12345678

section .text
global _start
_start:
    mov     eax, [rel packed]
```

```
    ; extract byte 2 (bits 23-16) using shift and mask
    shr     eax, 16
    movzx   eax, al               ; zero-extend the low byte

    ; exit with that byte as exit code
    mov     edi, eax
    mov     eax, 60               ; SYS_exit
    syscall
```

Run the program and check the exit code with `echo $?`: it will be 0x34 (52), since the third byte of 0x12345678 is 0x34.

20.3 Rotate Operations

Rotate instructions are similar to shifts, but the bits shifted out are fed back into the other end. The carry flag may be included in the rotation.

ROL, ROR (Rotate Without Carry)

- `ROL dest, count` — rotate left. The MSB moves to CF and simultaneously to the LSB.
- `ROR dest, count` — rotate right. The LSB moves to CF and simultaneously to the MSB.

```
mov   al, 0x81
rol   al, 1                 ; AL = 0x03, CF = 1
ror   al, 1                 ; AL = 0x81, CF = 1 again
```

RCL, RCR (Rotate Through Carry)

These include the carry flag in the rotation loop. `RCL` shifts left, filling LSB with CF and putting MSB into CF. `RCR` shifts right, filling MSB with CF and putting LSB into CF. They allow multi-word rotations by chaining.

```
; rotate a 128-bit value left by 1 (simple illustration)
clc
rcl   qword [low], 1
rcl   qword [high], 1
```

Flag Effects

- ROL/ROR: CF contains the bit shifted out. OF undefined for counts >1; for count=1, OF = CF XOR (new MSB).
- RCL/RCR: CF is part of the rotation; after the operation, CF holds the bit shifted out. OF defined similarly for count=1.
- SF, ZF, PF, AF unaffected.

Example: Byte Swap Using ROR

Rotations can swap nibbles or bytes within a word.

Code: listing20-3.asm — Rotate operations

```
; listing20-3.asm
; Assemble: nasm -f elf64 -o listing20-3.o listing20-3.asm
; Link:     ld -o listing20-3 listing20-3.o

bits 64
default rel

section .data
orig   dw  0x1234

section .text
global _start
_start:
    mov     ax, [rel orig]      ; AX = 0x1234
    ror     ax, 8               ; AX = 0x3412

    ; exit with the low byte of AX as exit code
    movzx   edi, al
    mov     eax, 60
    syscall
```

The exit code will be 0x12 (the former high byte), proving the rotation swapped the bytes.

20.4 Bit Manipulation

Beyond logical and shift instructions, `x86-64` provides a set of instructions that test and modify individual bits in an operand.

BT, BTS, BTR, BTC

These instructions operate on a bit specified by a bit index.

- `BT dest, index` — Copy the specified bit into CF, but do not modify `dest`.
- `BTS dest, index` — Test and set: copy bit to CF, then set the bit to 1.
- `BTR dest, index` — Test and reset: copy bit to CF, then clear the bit to 0.
- `BTC dest, index` — Test and complement: copy bit to CF, then toggle the bit.

The destination can be a register or a memory location. The index can be an immediate or a general-purpose register. When the destination is a memory operand, the instruction can address bits outside the immediate dword/qword by using the index register with a scaled offset. The assembler generates the correct addressing automatically.

```
bts   word [flags], 5          ; set bit 5, CF = old bit 5
btr   eax, ecx                 ; clear bit in EAX at position ECX
bt    qword [bitmap], r8       ; test bit without modifying
```

BSF, BSR

These scan a register or memory operand for the least significant or most significant set bit.

- `BSF dest, src` — Bit Scan Forward. Returns the index of the least significant set bit (0-based). If src is zero, the content of `dest` is undefined, and ZF is set to 1.
- `BSR dest, src` — Bit Scan Reverse. Returns the index of the most significant set bit. Same zero behaviour.

Both set ZF if the source is zero, and leave the destination undefined (though on most CPUs it holds the original value). Always test ZF before using the destination.

```
mov   rax, 0x8000000000000000
bsr   rcx, rax                ; RCX = 63
bsf   rcx, rax                ; RCX = 63 (same because only bit 63 set)
```

Example: Bitmap Allocation

A common use of BTS and BTR is managing a free-block bitmap.

Code: listing20-4.asm — Bitmap manipulation

```
; listing20-4.asm
; Assemble: nasm -f elf64 -o listing20-4.o listing20-4.asm
; Link:     ld -o listing20-4 listing20-4.o

bits 64
default rel

section .data
bitmap  dq  0                 ; 64 bits of flags

section .text
global _start
_start:
    ; allocate block 0 (set bit 0)
    lock bts qword [rel bitmap], 0   ; atomically test and set bit 0
    jc    .already_allocated

    ; allocate block 10
    lock bts qword [rel bitmap], 10

    ; free block 10
    lock btr qword [rel bitmap], 10

    ; exit normally
    xor     edi, edi
    mov     eax, 60
    syscall

.already_allocated:
    mov     edi, 1
    mov     eax, 60
    syscall
```

The `lock` prefix ensures atomicity in multi-threaded environments. Even in a single-threaded program it is harmless.

20.5 Flags Effects

A summary of flag modifications for bitwise and shift/rotate instructions is essential for correct conditional branching.

Instruction	OF	SF	ZF	AF	PF	CF
AND / TEST	0	*	*	?	*	0
OR	0	*	*	?	*	0
XOR	0	*	*	?	*	0
NOT	–	–	–	–	–	–
SHL/SAL (cnt=1)	*	*	*	?	*	*
SHL/SAL (cnt>1)	?	*	*	?	*	*
SHR (cnt=1)	*	*	*	?	*	*
SAR (cnt=1)	*	*	*	?	*	*
SHR/SAR (cnt>1)	?	*	*	?	*	*
ROL/ROR (cnt=1)	*	–	–	–	–	*
ROL/ROR (cnt>1)	?	–	–	–	–	*
RCL/RCR (cnt=1)	*	–	–	–	–	*
RCL/RCR (cnt>1)	?	–	–	–	–	*
BT/BTS/BTR/BTC	?	–	–	–	–	*
BSF/BSR	?	?	*	?	?	?

** = modified according to result; ? = undefined; – = unchanged; 0 = cleared.*

Using Flags After Bitwise Operations

- After AND / TEST, immediately use JZ (zero) or JNZ (not zero) to branch if a specific bit pattern is absent/present. Because OF and CF are cleared, JO and JC are not useful.
- After shifts, CF gives the last shifted-out bit, which can be used to check parity or to implement multi-word shifts.
- After BT variants, CF holds the original bit value. Use JC / JNC to branch accordingly.
- After BSF/BSR, test ZF to determine if the source was zero before using the index.

Example: Flag-Driven Branching

Code: listing20-5.asm — Conditional logic using bit flags

```
; listing20-5.asm
; Assemble: nasm -f elf64 -o listing20-5.o listing20-5.asm
; Link:     ld -o listing20-5 listing20-5.o

bits 64
default rel

section .data
status  db  0x82

section .text
```

```
global _start
_start:
    mov     al, [rel status]
    and     al, 0x80            ; isolate bit 7; ZF set if bit 7 was 0
    jnz     .bit7_set

    ; bit 7 not set
    mov     edi, 1
    jmp     .exit

.bit7_set:
    ; bit 7 set, now test bit 1
    bt      word [rel status], 1   ; CF = bit 1
    jc      .bit1_set
    mov     edi, 2
    jmp     .exit

.bit1_set:
    xor     edi, edi

.exit:
    mov     eax, 60
    syscall
```

This simple program tests individual flag bits and exits with different codes, demonstrating how bit and flag checks control execution flow.

Putting It All Together

Bitwise and logical instructions are the tools for precisely controlling data at the binary level. They underpin masking, bitfield extraction, Boolean logic, efficient multiplication/division by powers of two, and system-level flag management. Combined with a solid understanding of how each instruction affects the processor flags, you can write compact, high-performance code that interfaces directly with hardware registers and bitmask parameters. The next chapter will explore control flow, where these flags become the decision-making mechanism for branches and loops.

21

Control Flow

In This Chapter

The ability to alter the sequential flow of instructions is what gives a program its decision-making power. The x86-64 architecture provides unconditional jumps for direct transfers, a rich set of conditional jumps that test processor flags, and comparison instructions to set those flags based on arithmetic or bitwise relationships. This chapter covers the `JMP` instruction, the full family of conditional jumps (`Jcc`), the `CMP` and `TEST` instructions, the implementation of high-level loop constructs in assembly, and the simulation of switch-case statements using jump tables. Every description is based on the Intel® 64 and IA-32 Architectures Software Developer's Manual (Vol. 1 and Vol. 2A), the AMD64 Architecture Programmer's Manual, and the NASM 2.16 documentation. All examples are complete, Linux-compatible NASM programs targeting Fedora 43, using direct system calls for I/O and process exit.

21.1 JMP Instruction

`JMP` transfers execution unconditionally to a target address. In 64-bit mode, it can use a relative offset (most common) or an indirect address stored in a register or memory.

Relative Jumps

A relative jump adds a signed displacement to the instruction pointer (`RIP`). The displacement can be 8-, 16-, or 32-bits. NASM automatically selects the smallest encoding that reaches the target unless overridden with `short` or `near` modifiers.

```
jmp   .label          ; jump to local label (relative)
jmp   my_func         ; jump forward/backward
jmp   short .close    ; force 8-bit displacement
```

Indirect Jumps

An indirect jump loads the target address from a register or memory.

```
jmp   rax              ; jump to address in RAX
jmp   qword [jtable + rcx*8]   ; jump through a table of pointers
```

This is essential for implementing switch statements and for calling function pointers.

Flag Effects

`JMP` does not affect any flags or the stack. It simply sets `RIP`.

21.2 Conditional Jumps

Conditional jumps transfer control only if a certain condition, represented by one or more status flags, is met. The mnemonics follow the pattern `Jcc`, where `cc` describes the condition.

Jump Mnemonics Based on Individual Flags

- `JZ` / `JE` — jump if ZF = 1 (zero / equal).
- `JNZ` / `JNE` — jump if ZF = 0 (not zero / not equal).
- `JS` — jump if SF = 1 (negative).
- `JNS` — jump if SF = 0 (non-negative).
- `JC` / `JB` / `JNAE` — jump if CF = 1 (carry / below / not above or equal). Used for unsigned comparisons.
- `JNC` / `JNB` / `JAE` — jump if CF = 0 (no carry / not below / above or equal).
- `JO` — jump if OF = 1 (overflow).
- `JNO` — jump if OF = 0 (no overflow).
- `JP` / `JPE` — jump if PF = 1 (parity even).
- `JNP` / `JPO` — jump if PF = 0 (parity odd).

Jump Mnemonics for Signed Comparisons

These evaluate combinations of SF and OF after a `CMP` or arithmetic instruction.

- `JL` / `JNGE` — jump if less (SF != OF).
- `JLE` / `JNG` — jump if less or equal (SF != OF or ZF = 1).
- `JG` / `JNLE` — jump if greater (SF = OF and ZF = 0).
- `JGE` / `JNL` — jump if greater or equal (SF = OF).

Jump Mnemonics for Unsigned Comparisons

These are based on CF and ZF.

- `JB` / `JNAE` — jump if below (CF = 1).
- `JBE` / `JNA` — jump if below or equal (CF = 1 or ZF = 1).
- `JA` / `JNBE` — jump if above (CF = 0 and ZF = 0).
- `JAE` / `JNB` / `JNC` — jump if above or equal (CF = 0).

Range of Conditional Jumps

Originally, conditional jumps were limited to a displacement of –128 to +127 bytes. In 64-bit mode, the processor supports a full 32-bit displacement for all `Jcc` instructions, allowing them to reach any address within the same image. NASM automatically uses the larger encoding if the target is far away. You can force a short jump with `Jcc short label`, but if the distance exceeds 127 bytes the assembler will generate an error.

Example: Basic Conditional Logic

Code: listing21-1.asm — Conditional jumps

```
; listing21-1.asm
; Assemble: nasm -f elf64 -o listing21-1.o listing21-1.asm
; Link:     ld -o listing21-1 listing21-1.o

bits 64
default rel

section .data
a  dq  10
b  dq  20

section .text
global _start
_start:
    mov     rax, [rel a]
    mov     rbx, [rel b]

    cmp     rax, rbx
    jl      .less_than      ; signed: jump if a < b
    jg      .greater_than
    je      .equal

.less_than:
    mov     edi, 1
    jmp     .exit
.greater_than:
    mov     edi, 2
    jmp     .exit
.equal:
    xor     edi, edi
```

```
.exit:
    mov     eax, 60          ; SYS_exit
    syscall
```

When assembled and run, the program exits with code 1, because 10 is less than 20.

Warning

Choosing signed vs. unsigned jumps. If you compare addresses (64-bit pointers), use unsigned jumps (`JA`, `JB`) because addresses are inherently unsigned. For signed integers, use `JL`, `JG`. A common bug is using `JL` on pointers, which can fail for addresses above the 2 GiB boundary.

21.3 CMP and TEST

These instructions are used exclusively to set flags for subsequent conditional jumps. They do not modify the destination operand.

21.3.1 CMP — Compare

`CMP dest, src` computes `dest - src` and sets the flags exactly like `SUB`, but discards the result. So after a `CMP`, the flags represent the relationship between `dest` and `src`.

```
cmp   rax, 5
je    .is_five            ; ZF = 1 if RAX == 5
jg    .greater_than_five  ; signed comparison
```

21.3.2 TEST — Logical Compare

`TEST dest, src` performs a bitwise AND between the two operands and sets the flags according to the result, but discards it. It is ideal for testing if specific bits are zero, or whether a value is zero (since `TEST reg, reg` is often used in place of `CMP reg, 0`).

```
test  al, 0x80
jnz   .bit7_is_set        ; jump if bit 7 is 1

test  rax, rax
jz    .rax_is_zero
```

Flag Effects Summary

Both `CMP` and `TEST` set `SF`, `ZF`, `PF` according to the result; `AF` is undefined; `CF` and `OF` are cleared for `TEST`, while `CMP` sets them as `SUB` would.

Example: Using CMP and TEST Together

Code: listing21-2.asm — Range check with CMP and bit test

```
; listing21-2.asm
; Assemble: nasm -f elf64 -o listing21-2.o listing21-2.asm
; Link:     ld -o listing21-2 listing21-2.o

bits 64
default rel

section .data
value  dq  0x0000000000000041

section .text
global _start
_start:
    mov     rax, [rel value]

    ; test if bit 6 is set
    test    rax, 0x40
    jnz     .bit6_set

    ; compare against a range (unsigned 10-20)
    cmp     rax, 10
    jb      .out_of_range
    cmp     rax, 20
    ja      .out_of_range
    ; in range
    xor     edi, edi
    jmp     .exit

.bit6_set:
    mov     edi, 1
    jmp     .exit

.out_of_range:
    mov     edi, 2

.exit:
    mov     eax, 60
    syscall
```

The value 0x41 has bit 6 set (0x40), so the program takes the `.bit6_set` branch and exits with code 1. If you change `value` to 15, it exits with 0.

21.4 Loop Structures

High-level language loops—`while`, `do-while`, `for`—map directly to combinations of a counter, a comparison, and a conditional jump. In assembly, the pattern is explicit.

Do-While Loop (Post-Test)

The simplest form: the body executes at least once, and the condition is checked at the end.

```
.loop:
    ; body
    inc   ecx
    cmp   ecx, 10
    jb    .loop
```

While Loop (Pre-Test)

The condition is evaluated before entering the body. A separate check with a jump is required.

```
    jmp   .condition
.body:
    ; body
    inc   ecx
.condition:
    cmp   ecx, 10
    jb    .body
```

Often the `jmp` can be eliminated by reordering:

```
    cmp   ecx, 10
    jae   .done
.loop:
    ; body
    inc   ecx
    cmp   ecx, 10
    jb    .loop
.done:
```

For Loop

A typical `for (i = 0; i < N; i++)` is:

```
    xor   ecx, ecx          ; i = 0
.loop:
    cmp   ecx, N
    jge   .exit_loop
    ; body using ecx as index
    inc   ecx
    jmp   .loop
.exit_loop:
```

LOOP Instruction (Legacy)

The `LOOP` instruction decrements `ECX` (or `RCX` in 64-bit) and jumps to the target if `ECX` != 0. It is compact but slower on modern processors than separate `DEC` + `JNZ`, because `LOOP` is micro-coded and can cause stalls. It also does not set flags, so you cannot test for zero after the loop. Use of `DEC/JNZ` or `SUB/JNZ` is recommended.

```
    mov   ecx, 100
.loop:
    ; body
    sub   ecx, 1
    jnz   .loop
```

Example: Summing an array with multiple loop patterns

Code: listing21-3.asm — Loop implementations

```
; listing21-3.asm
; Assemble: nasm -f elf64 -o listing21-3.o listing21-3.asm
; Link:     ld -o listing21-3 listing21-3.o

bits 64
default rel

section .data
array   dq  1, 2, 3, 4, 5, 6, 7, 8, 9, 10
count   equ ($ - array) / 8

section .bss
total   resq 1

section .text
global _start
_start:
    ; for-loop style summation
    xor     eax, eax
    xor     ecx, ecx
    lea     rdx, [rel array]
.for_loop:
    cmp     ecx, count
    jge     .done
    add     rax, [rdx + rcx*8]
    inc     ecx
    jmp     .for_loop

.done:
    mov     [rel total], rax

    ; while-loop style (pointer increment)
    xor     eax, eax
    lea     rdx, [rel array]
    mov     rcx, count
.while_loop:
    test    rcx, rcx
    jz      .done2
    add     rax, [rdx]
    add     rdx, 8
    dec     rcx
    jmp     .while_loop

.done2:
    add     [rel total], rax     ; total now double the sum (110 = 2*55)

    ; exit with the sum in total as exit code (just low byte for simplicity)
    movzx   edi, byte [rel total]
    mov     eax, 60
    syscall
```

The first loop accumulates the sum, the second accumulates again, so the final total is $2 \times 55 = 110$.

The program exits with 110 (since 110 fits in a byte, the low byte is 110).

Tip

For tight loops, unrolling (writing the body multiple times per iteration) can reduce the overhead of the loop counter and branch. But it increases code size and may not always be faster. Profile before unrolling.

21.5 Switch Simulation

The `switch-case` construct of high-level languages can be translated into a chain of `CMP/JE` comparisons or, when cases are dense, a jump table for O(1) dispatch.

If-Else Chain (Sparse Cases)

For a small number of non-consecutive cases, use consecutive comparisons:

```
cmp   eax, 1
je    .case1
cmp   eax, 3
je    .case3
cmp   eax, 5
je    .case5
jmp   .default
```

This is simple but performance degrades linearly with the number of cases.

Jump Table (Dense Cases)

For consecutive values starting at 0 or a known base, build an array of code pointers and jump through it after bounds-checking the index.

Code: listing21-4.asm — Jump table for switch

```
; listing21-4.asm
; Assemble: nasm -f elf64 -o listing21-4.o listing21-4.asm
; Link:     ld -o listing21-4 listing21-4.o

bits 64
default rel

section .data
case_table:
    dq  .case0
    dq  .case1
    dq  .case2
    dq  .case_default   ; case 3 goes to default
case_count equ 4

section .text
global _start
_start:
```

```
    mov     eax, 2              ; selector
    cmp     eax, case_count
    jae     .case_default
    lea     rcx, [rel case_table]
    jmp     qword [rcx + rax*8]

.case0:
    mov     edi, 10
    jmp     .dispatch_end
.case1:
    mov     edi, 20
    jmp     .dispatch_end
.case2:
    mov     edi, 30
    jmp     .dispatch_end
.case_default:
    xor     edi, edi

.dispatch_end:
    mov     eax, 60
    syscall
```

The program exits with code 30 (case 2). The jump table is indexed directly, resulting in constant-time dispatch.

Interactive Menu Example Using System Calls

This program prints a menu, reads a single character, and dispatches to an option handler using conditional jumps. It uses the `write` and `read` system calls directly.

Code: listing21-5.asm — Menu dispatch with syscalls

```
; listing21-5.asm
; Assemble: nasm -f elf64 -o listing21-5.o listing21-5.asm
; Link:     ld -o listing21-5 listing21-5.o

bits 64
default rel

SYS_WRITE equ 1
SYS_READ  equ 0
SYS_EXIT  equ 60
STDOUT    equ 1
STDIN     equ 0

section .data
menu      db 'Choose 1-3: ',0
menu_len  equ $ - menu
msg1      db 'Option 1 selected',10,0
len1      equ $ - msg1
msg2      db 'Option 2 selected',10,0
len2      equ $ - msg2
msg3      db 'Option 3 selected',10,0
```

```
len3      equ $ - msg3
def_msg   db 'Invalid option',10,0
len_def   equ $ - def_msg

section .bss
buf       resb 2              ; one char + newline possible

section .text
global _start
_start:
    ; display menu
    mov     eax, SYS_WRITE
    mov     edi, STDOUT
    lea     rsi, [rel menu]
    mov     edx, menu_len
    syscall

    ; read user input (one character, possibly followed by newline)
    mov     eax, SYS_READ
    mov     edi, STDIN
    lea     rsi, [rel buf]
    mov     edx, 2
    syscall

    ; convert first byte to digit and dispatch
    movzx   eax, byte [rel buf]
    sub     al, '0'             ; character to integer

    cmp     al, 1
    je      .opt1
    cmp     al, 2
    je      .opt2
    cmp     al, 3
    je      .opt3
    jmp     .default

.opt1:
    mov     eax, SYS_WRITE
    mov     edi, STDOUT
    lea     rsi, [rel msg1]
    mov     edx, len1
    syscall
    jmp     .exit

.opt2:
    mov     eax, SYS_WRITE
    mov     edi, STDOUT
    lea     rsi, [rel msg2]
    mov     edx, len2
    syscall
    jmp     .exit

.opt3:
    mov     eax, SYS_WRITE
```

```
    mov     edi, STDOUT
    lea     rsi, [rel msg3]
    mov     edx, len3
    syscall
    jmp     .exit

.default:
    mov     eax, SYS_WRITE
    mov     edi, STDOUT
    lea     rsi, [rel def_msg]
    mov     edx, len_def
    syscall

.exit:
    xor     edi, edi
    mov     eax, SYS_EXIT
    syscall
```

This interactive program prompts the user for a digit, then prints the corresponding message using direct system calls. It demonstrates control flow fully integrated with Linux I/O.

Putting It All Together

Control flow in x86-64 assembly is explicit and direct. Unconditional JMP, the rich set of conditional jumps, and the comparison and test instructions provide all the decision-making capabilities of higher-level languages with complete transparency. Loop patterns translate trivially to comparison-and-jump sequences, and switch statements can be implemented as simple if-else chains or efficient jump tables. Mastering these constructs, and understanding exactly which flags are set by each instruction, is essential for writing correct, efficient programs. The next chapter will explore the CALL and RET mechanisms and the construction of subroutines.

22

Stack Operations

In This Chapter

The stack is the backbone of function calling and local storage in x86-64 assembly. Every call to a C library function, every local variable not held in a register, and every preservation of a non-volatile register relies on the disciplined use of the stack. This chapter dissects the fundamental stack instructions — `PUSH` and `POP` — the control-flow pairing of `CALL` and `RET`, the structure of a standard stack frame, the most common forms of stack corruption, and the techniques for debugging stack-related problems using GDB. All information is based on the Intel® 64 and IA-32 Architectures Software Developer's Manual (Vol. 1 and 2A), the AMD64 Architecture Programmer's Manual, and the System V AMD64 ABI used by Linux. Every code listing has been tested with NASM 2.16 on Fedora 43.

22.1 PUSH and POP

The `PUSH` and `POP` instructions are the most direct means of storing and retrieving values on the stack. They implicitly use the stack pointer register `RSP` and operate on 64-bit operands in 64-bit mode (16-bit and 32-bit operand sizes are also available with an operand-size override prefix, but they are rarely used in 64-bit Linux code).

PUSH

`PUSH src` performs two actions atomically:

1. `RSP = RSP - 8`
2. The 64-bit value of `src` is written to the memory location addressed by the new `RSP`.

The source can be a general-purpose register, a segment register (rare in flat mode), a memory operand, or an immediate value (sign-extended to 64 bits if necessary).

```
push    rax             ; RSP -= 8, [RSP] = RAX
push    qword [var]     ; push memory quadword
push    0x123456789ABCDEF  ; push a 64-bit immediate
push    -1              ; sign-extended 32-bit immediate, then pushed as 64-bit
```

POP

`POP dest` reverses the operation:

1. The 64-bit value at memory `[RSP]` is read and stored into `dest`.
2. `RSP = RSP + 8`

The destination can be a register or a memory operand.

```
pop     rbx             ; RBX = [RSP]; RSP += 8
pop     qword [var]     ; read stack top into memory; RSP += 8
```

Stack Alignment and PUSH/POP

Because each `PUSH` and `POP` adjusts `RSP` by 8, the stack alignment changes accordingly. In Linux x86-64, `RSP` must be 16-byte aligned at the point of a `CALL` instruction. After `CALL` pushes the return address, `RSP` becomes 8 mod 16 (i.e., misaligned by 8). A single `PUSH` after that makes it 0 mod 16 again; another `PUSH` makes it 8 mod 16. Programmers must track alignment manually; a single unbalanced push/pop can cause alignment faults in C library calls, which often use SSE instructions that require alignment.

PUSH/POP for Register Preservation

When you need to use a non-volatile register inside a function and then restore its original value, you push it at the start and pop it at the end. The order of pops must be the reverse of pushes.

Code: listing22-1.asm — Push/Pop preservation

```
; listing22-1.asm
; Assemble: nasm -f elf64 -o listing22-1.o listing22-1.asm
; Link:     ld -o listing22-1 listing22-1.o

bits 64
default rel

section .text
global _start
_start:
    push    rbx                 ; save RBX
    push    r12                 ; save R12

    ; use RBX and R12 freely
    mov     rbx, 0xAAAA
    mov     r12, 0xBBBB
```

```
    ; restore in reverse order
    pop     r12
    pop     rbx

    ; exit(0)
    mov     eax, 60
    xor     edi, edi
    syscall
```

After the pops, RBX and R12 retain their original values, and RSP is back to its pre-push position.

Warning

Never skip a pop or add an extra push without balancing. An unbalanced stack will cause RET to fetch a wrong return address and crash the program.

22.2 CALL and RET

These instructions manage the transfer of control to and from subroutines, using the stack to store return addresses.

CALL

CALL <target> performs:

1. PUSH the address of the instruction immediately following the CALL (the return address) onto the stack.
2. JMP to the target address.

The target can be a relative offset (the most common form), an absolute address in a register, or a memory operand (indirect call).

```
call    my_function         ; relative displacement
call    rax                 ; indirect through register
call    qword [jtable + rcx*8]  ; indirect through memory
```

In Linux, the caller must also ensure that RSP is 16-byte aligned before the CALL and that arguments are placed in the correct registers.

RET

RET (and its variant RETN) pops the return address from the stack and jumps to it. The processor expects that RSP points exactly to the return address pushed by the corresponding CALL. After the pop, RSP is incremented by 8.

If the callee has pushed any registers or allocated local space, it must undo those adjustments before RET, so that RSP is at the correct position.

```
ret                 ; pop return address, jump to it
ret     16          ; pop return address, then RSP += 16 (used for old stdcall, not in System V)
```

In the System V AMD64 ABI, the caller cleans the stack after a call if arguments were pushed, but normally functions take only up to six arguments in registers. The callee is responsible for restoring saved registers and deallocating its own frame. The `RET` instruction is usually bare.

CALL/RET Internals

When a function is called, the stack looks like this (from caller's perspective, high addresses to low):

- Any stack arguments beyond the first six (pushed right-to-left by caller).
- Return address (8 bytes) pushed by `CALL`.
- Then callee's frame: saved register, local variables.

There is no mandatory shadow space in the System V ABI; the callee may freely use the area below the return address (after allocating it) for its own purposes.

Example: Simple Call and Return

Code: listing22-2.asm — CALL/RET mechanism

```
; listing22-2.asm
; Assemble: nasm -f elf64 -o listing22-2.o listing22-2.asm
; Link:     ld -o listing22-2 listing22-2.o

bits 64
default rel

section .text
global _start

helper:
    ret

_start:
    call    helper              ; RSP -= 8, return address pushed
    ; after return, RSP is restored

    mov     eax, 60
    xor     edi, edi
    syscall
```

Observe in GDB: at the start of `helper`, `RSP` points to the return address pushed by `CALL helper`. After `RET`, `RSP` returns to its previous value.

Tip

When stepping through in a debugger, note that each `CALL` pushes an 8-byte return address, and each `RET` pops it. The sequence perfectly balances the stack if frames are properly set up.

22.3 Stack Frames

A stack frame is the region of the stack belonging to one function invocation. It contains the return address, saved non-volatile registers, and local variables. Optionally a frame pointer (RBP) is established for stable access to arguments and locals, especially in non-leaf functions.

Standard Linux x86-64 Prologue

Many assembly functions use a frame pointer (RBP) for clarity. The typical prologue:

```
push    rbp
mov     rbp, rsp
sub     rsp, N          ; allocate N bytes for locals (N is multiple of 16 to keep alignment)
```

And the epilogue:

```
mov     rsp, rbp
pop     rbp
ret
```

After `push rbp`, RSP becomes 16-byte aligned (because entry RSP was misaligned by 8). Subtracting a multiple of 16 preserves alignment.

A Full Frame Example

Below is a function that takes two arguments, stores them in local variables, and calls `printf` to print a message. It demonstrates a complete frame.

Code: listing22-3.asm — Frame with frame pointer and libc call

```
; listing22-3.asm
; Assemble: nasm -f elf64 -o listing22-3.o listing22-3.asm
; Link:     gcc -no-pie -o listing22-3 listing22-3.o

bits 64
default rel

extern printf
extern exit

section .data
fmt db 'Result: %ld', 10, 0

section .text
global main

; framed_func(a, b) -> a + b
; arguments: RDI = a, RSI = b
framed_func:
    push    rbp
    mov     rbp, rsp
    sub     rsp, 16             ; 16 bytes locals (keep alignment)

    ; store arguments in locals
```

```
    mov     [rbp-8],  rdi
    mov     [rbp-16], rsi

    ; compute sum
    mov     rax, rdi
    add     rax, rsi
    mov     [rbp-8], rax        ; overwrite with result (optional)

    ; return value already in RAX
    mov     rsp, rbp
    pop     rbp
    ret

main:
    push    rbp
    mov     rbp, rsp
    sub     rsp, 16             ; align and space for locals

    ; call framed_func(10, 20)
    mov     edi, 10
    mov     esi, 20
    call    framed_func         ; RAX = 30

    ; print result
    lea     rdi, [rel fmt]
    mov     rsi, rax
    xor     eax, eax
    call    printf

    ; exit(0)
    xor     edi, edi
    call    exit
```

The function `framed_func` preserves the old `RBP`, sets a new frame pointer, allocates local storage, and uses negative offsets from `RBP` to access locals. The epilogue restores the stack and returns.

Stack Alignment in Depth

Before any `CALL`, RSP must be 16-byte aligned. After entry to a function (after the call), RSP is 8 mod 16. The common sequence `push rbp` subtracts 8, making RSP aligned. Then if you subtract a multiple of 16, alignment is preserved. If you do not use a frame pointer, you should subtract an odd multiple of 8 from RSP to align the stack before a call. For example, a leaf function that does not call anyone does not need to adjust alignment; but if it uses SSE on the stack, alignment matters. For a function that calls others, you often do:

```
sub     rsp, 8 + N      ; 8 to re-align after entry, N for locals (multiple of 16)
...
add     rsp, 8 + N
ret
```

The chapter on calling conventions will cover this thoroughly.

22.4 Stack Corruption Issues

Stack corruption is one of the most common and insidious bugs in assembly programming. It manifests as crashes, incorrect return values, or silent data corruption.

Unbalanced PUSH/POP

Forgetting to pop a pushed value misaligns the stack. When `RET` executes, `RSP` does not point to the correct return address.

Code: Bad example — unbalanced push

```
my_bad_func:
    push    rax
    ; ... function logic ...
    ; missing pop rax
    ret           ; RSP points to RAX value, not return address → crash
```

Correction: ensure every push has a corresponding pop, and the order is reversed.

Wrong Stack Alignment for SSE

If the stack is not 16-byte aligned when `CALL` is executed, some SSE instructions inside C library functions will cause a segmentation fault. This often appears as a crash inside `memcpy` or `printf` on a modern system. Check with GDB: before the call, `print $rsp` should show a hex value ending in `0`. If not, adjust the prologue.

Buffer Overflows on the Stack

Local arrays on the stack can be overwritten, corrupting the return address. This is the classic buffer overflow attack. In assembly, there is no bounds checking. When copying data into a stack buffer, always enforce the maximum copy length, or use a safe string copy (like `strncpy` in C, which you can call).

Warning

Linux uses stack canaries in GCC-compiled programs. NASM does not automatically insert canaries. If you write a function with a local array and pass its address to an API that writes an unknown amount, you risk corruption. Always control the number of bytes written.

Overwriting the Return Address

If a function writes beyond the bounds of its local variables, it can overwrite the saved return address. The program may then "return" to an arbitrary location, often causing a segfault. To avoid this, use precise sizes in copy operations.

Example: Deliberate Stack Corruption

The following program deliberately corrupts the stack to illustrate a crash scenario. Running it will trigger a segfault; comment out the bad line to restore normal operation.

Code: listing22-4.asm — Deliberate stack corruption

```
; listing22-4.asm
; Assemble: nasm -f elf64 -o listing22-4.o listing22-4.asm
; Link:     ld -o listing22-4 listing22-4.o

bits 64
default rel

section .text
global _start
_start:
    ; allocate a local buffer of 16 bytes
    sub     rsp, 16
    mov     rcx, rsp              ; pointer to buffer

    ; simulate an overflow: write far beyond the buffer
    mov     qword [rcx+32], 0x41414141   ; overwrites return address of _start
    ; The program will crash when trying to return, so we just exit.
    ; But if we had a function, the ret would be corrupted.

    ; Clean up and exit (actually we just exit to avoid crash)
    add     rsp, 16
    mov     eax, 60
    xor     edi, edi
    syscall
```

If this were a function returning via RET, the overwritten return address would cause a segfault.

22.5 Debugging Stack Behavior

Diagnosing stack issues requires inspecting the stack memory and verifying register values. GDB provides powerful commands.

Inspecting the Stack with GDB

- `x/10gx $rsp` — dump 10 quadwords starting at the stack pointer. This shows the return address at the top.
- `info registers` — displays all registers, including RSP and RBP.
- `backtrace` (or `bt`) — attempts to unwind the call stack based on frame pointers (if the code uses a frame pointer) or DWARF unwind information (if compiled with debug info).
- `frame <n>` — select a stack frame.
- `info frame` — shows details of the current frame.
- `layout asm` — switch to a TUI layout that shows the disassembly and registers side-by-side.

Detecting Misalignment

Before a CALL instruction, RSP must be a multiple of 16. In GDB, observe RSP just before the CALL with `print $rsp`. The last hex digit should be `0`. If not, alignment is broken. To find the root

cause, walk backward through the code and check every `sub rsp` or `push` that precedes the call.

Example: Alignment Debugging with a Syscall

Code: listing22-5.asm — Alignment debugging example

```
; listing22-5.asm
; Assemble: nasm -f elf64 -o listing22-5.o listing22-5.asm
; Link:     ld -o listing22-5 listing22-5.o

bits 64
default rel

section .data
msg db 'OK', 10
len equ $ - msg

section .text
global _start
_start:
    sub     rsp, 8            ; align stack (entry RSP is 8 mod 16, subtract 8 makes it
    ↪  16-byte aligned)

    ; Now RSP is aligned, we can call write
    mov     eax, 1            ; SYS_write
    mov     edi, 1            ; stdout
    lea     rsi, [rel msg]
    mov     edx, len
    syscall                   ; syscall does not require alignment, but we demonstrate before
    ↪  call

    ; exit
    add     rsp, 8
    mov     eax, 60
    xor     edi, edi
    syscall
```

Set a breakpoint at the `syscall` instruction in GDB and check `RSP`. Even though the kernel does not enforce alignment, calls to C library functions do. The critical point is before a `CALL`, RSP must be aligned.

Tracing Stack-Related Crashes

When the program crashes with a segmentation fault, use `backtrace` to see the call stack. The instruction that faulted is often an SSE load/store requiring alignment. The misalignment usually originated several calls earlier. Look at each function's prologue to ensure it properly aligns the stack before sub-calls.

Note

A common pattern is that a leaf function works fine, but the program crashes after calling a function that uses SSE internally. This is because the leaf did not correct its stack alignment

before calling, and the first function that uses aligned load/store triggers the fault.

Putting It All Together

The stack is a deceptively simple structure that demands rigid discipline. Correct use of `PUSH` and `POP`, proper `CALL` and `RET` matching, consistent frame construction, and vigilant alignment checking are the pillars of reliable assembly programming on Linux. When things go wrong, GDB with its stack-dump capabilities is the primary tool for diagnosis. The next chapter completes the picture by examining how functions are designed, covering parameter passing, return values, and design patterns in the System V AMD64 ABI.

23

Function Design in Assembly

In This Chapter

Functions — also called procedures or subroutines — are the fundamental unit of code organisation in assembly language. A well-designed function is reusable, respects the platform's calling convention, and leaves no unintended side effects. This chapter covers the complete anatomy of a function in NASM for Linux x86-64 under the System V AMD64 ABI: the standard prologue and epilogue, the passing of parameters in registers and on the stack, the mechanisms for returning values, the management of local variables, and a set of proven design patterns that cover leaf functions, non-leaf functions, library wrappers, and more. Every rule is derived from the System V AMD64 ABI supplement, the Intel® 64 and IA-32 Architectures Software Developer's Manual, and the AMD64 Architecture Programmer's Manual. All examples are self-contained and tested with NASM 2.16.01 on Fedora 43.

23.1 Function Structure

A complete function consists of a **prologue**, a **body**, and an **epilogue**. The prologue sets up the stack frame to accommodate saved registers and local variables, and (if the function makes calls) ensures the stack is aligned. The epilogue undoes these allocations and returns control to the caller.

Minimal Leaf Function

A leaf function makes no `CALL` instructions. Because it does not call anyone else, it does not need to adjust the stack beyond what is necessary for its own local storage and alignment if it uses SSE. If it uses no non-volatile registers, the frame can be extremely short.

Code: listing23-1.asm — Minimal leaf function

```
; listing23-1.asm
; Assemble: nasm -f elf64 -o listing23-1.o listing23-1.asm
; Link:     gcc -no-pie -o listing23-1 listing23-1.o

bits 64
default rel

extern printf
extern exit

section .data
fmt db 'Sum: %ld', 10, 0

section .text
global main

; add_three(a, b, c) -> a + b + c
; arguments: RDI, RSI, RDX
add_three:
    lea     rax, [rdi + rsi + rdx]    ; compute sum
    ret

main:
    push    rbp
    mov     rbp, rsp
    sub     rsp, 16                   ; align stack for calls

    ; call add_three(10, 20, 30)
    mov     edi, 10
    mov     esi, 20
    mov     edx, 30
    call    add_three

    ; print result using printf
    lea     rdi, [rel fmt]
    mov     rsi, rax
    xor     eax, eax
    call    printf

    xor     edi, edi
    call    exit
```

The leaf function `add_three` does not touch the stack at all; it computes the result entirely in registers and returns. No alignment adjustment is needed because it doesn't call anyone.

Non-Leaf Function with Full Frame

When a function calls another function, it must ensure that the stack is 16-byte aligned at the point of each call. A frame pointer (`RBP`) is often established to simplify access to locals and arguments, especially when the stack pointer might change during the function.

Code: listing23-2.asm — Non-leaf function with frame pointer

```
; listing23-2.asm
; Assemble: nasm -f elf64 -o listing23-2.o listing23-2.asm
; Link:     gcc -no-pie -o listing23-2 listing23-2.o

bits 64
default rel

extern printf
extern exit

section .data
fmt db 'In print_message: %s', 10, 0
msg db 'Hello', 0

section .text
global main

; print_message(str)
; str in RDI
print_message:
    push    rbp
    mov     rbp, rsp
    sub     rsp, 16             ; align and maybe space for locals

    ; save str in local variable (optional)
    mov     [rbp-8], rdi

    ; print the string using printf
    lea     rdi, [rel fmt]
    mov     rsi, [rbp-8]
    xor     eax, eax
    call    printf

    mov     rsp, rbp
    pop     rbp
    ret

main:
    push    rbp
    mov     rbp, rsp
    sub     rsp, 16

    lea     rdi, [rel msg]
    call    print_message

    xor     edi, edi
    call    exit
```

The function `print_message` demonstrates the classic prologue: push old RBP, set RBP to current RSP, then subtract space for alignment (exact multiple of 16). The epilogue restores RSP and RBP. The function calls `printf`, so it must maintain alignment; after `push rbp`, RSP is aligned, and `sub rsp, 16` keeps it aligned.

23.2 Parameter Passing

The System V AMD64 calling convention specifies how arguments are transferred.

Register Parameters

The first six integer or pointer arguments are passed in registers, left to right:

`RDI, RSI, RDX, RCX, R8, R9`

This holds for all functions, including those that later call sub-functions.

```
; int func(int a, long b, char *c);
; RDI = a, RSI = b, RDX = c
func:
    ...
```

Stack Parameters

Any arguments beyond the sixth are pushed onto the stack from right to left. The caller must provide space for them above the return address (i.e., before the call). The callee accesses them at positive offsets from RBP (or RSP).

Example: a function with 8 arguments.

Code: listing23-3.asm — Passing eight arguments

```
; listing23-3.asm
; The function sum8 returns the sum of eight integers.
; RDI..R9 contain first six, rest on stack.

bits 64
default rel

section .text
global sum8

sum8:
    push    rbp
    mov     rbp, rsp

    mov     rax, rdi
    add     rax, rsi
    add     rax, rdx
    add     rax, rcx
    add     rax, r8
    add     rax, r9
    add     rax, qword [rbp+16]   ; stack arg 7 (right above return addr)
    add     rax, qword [rbp+24]   ; stack arg 8

    pop     rbp
    ret
```

The caller would call it as:

```
push    arg8
push    arg7
call    sum8
add     rsp, 16          ; caller cleans the stack
```

Floating-Point Parameters

Floating-point arguments use the XMM registers: XMM0–XMM7 for the first eight values. They are separate from the integer registers. For a function taking `(int a, double b)`, `a` goes in RDI, `b` in XMM0.

23.3 Return Values

A function can return a value to the caller using the following rules:

- **Integer / pointer return value:** `RAX` (or EAX for 32-bit, AX for 16-bit, AL for 8-bit).
- **Floating-point / SIMD return value:** `XMM0` (or ST(0) for long double).
- **Composite (struct) return:** If the struct is larger than 16 bytes, the caller passes a hidden pointer as the first argument (RDI), and the function writes the result through that pointer. The function returns the same pointer in RAX. (Exact rules are complex; this book focuses on simple cases.)

Example: Returning a Struct

Code: listing23-4.asm — Returning a struct via hidden pointer

```
; listing23-4.asm
bits 64
default rel

section .text
global get_point

; struct Point { long x; long y; };
; get_point(Point *result)   ; RDI = pointer to result
get_point:
    mov     qword [rdi],    100
    mov     qword [rdi+8],  200
    mov     rax, rdi              ; return pointer in RAX (ABI)
    ret
```

The caller would allocate space for the struct and pass its address in RDI:

```
sub     rsp, 16                 ; reserve space for Point
mov     rdi, rsp
call    get_point
; now the values are at [rsp] and [rsp+8]
```

23.4 Local Variables

Local (automatic) variables are storage locations that exist only during the function invocation. They are typically placed on the stack, either at negative offsets from RBP or at positive offsets from RSP after the prologue.

Allocation with Fixed Offsets

In the prologue, subtracting a constant from RSP reserves space for locals. The variables can be accessed relative to RBP (e.g., `[rbp-8]`) for clarity.

```
push    rbp
mov     rbp, rsp
sub     rsp, 32         ; allocate 32 bytes for locals (must keep alignment)
; ...
mov     qword [rbp-8], 0x123    ; first local
mov     dword [rbp-12], 42      ; second local
; ...
mov     rsp, rbp
pop     rbp
ret
```

Proper Alignment of Locals

If the function will use SSE instructions on local variables, those variables must be 16-byte aligned. Since RSP is 16-byte aligned after the prologue `push rbp; mov rbp, rsp; sub rsp, N` (if N is multiple of 16), any local placed at an offset that is a multiple of 16 from RSP will be aligned. Using `align` on the stack is not possible, but you can check offsets manually.

Zero-Initialising Local Arrays

The stack is not zeroed automatically. If a local array is used as a buffer, you can zero it using `REP STOSB`:

```
; zero 64-byte local buffer
lea     rdi, [rbp-64]
xor     eax, eax
mov     ecx, 64
rep     stosb
```

23.5 Design Patterns

Experience has distilled a set of reliable patterns for writing assembly functions under the System V ABI.

Leaf Function (No Calls, Minimal Overhead)

A leaf function does not call other functions, so it does not need to adjust the stack beyond what it uses for locals. It can often avoid a frame pointer and simply work with registers.

```
; multiply RCX by 10 using LEA
mul10:
```

```
    lea     rax, [rdi + rdi*4]    ; rax = 5 * rdi
    lea     rax, [rax + rax]      ; rax = 10 * rdi
    ret
```

Non-Leaf Function (Standard Stack Frame)

When a function makes calls, it must ensure stack alignment. The general template:

```
my_func:
    push    rbp
    mov     rbp, rsp
    sub     rsp, 16 * k       ; allocate k qwords for locals, maintain alignment
    ; ... save non-volatile registers if used ...
    ; ... body ...
    ; ... restore non-volatile registers ...
    mov     rsp, rbp
    pop     rbp
    ret
```

C Library Wrapper

Often you will write a thin wrapper around a C library function to add default parameters or simplify calls.

Code: listing23-5.asm — Library wrapper with tail call

```
; listing23-5.asm
bits 64
default rel

extern printf

section .text
; wrapper: my_puts(str)
; just calls printf("%s\n", str)
global my_puts
my_puts:
    push    rbp
    mov     rbp, rsp
    lea     rdx, [rel newline]    ; but printf expects format as first arg
    ; Better to convert, but this is a demonstration.
    ; Actually just call puts from libc.
    pop     rbp
    jmp     puts                  ; tail call (won't return here)
    ; But we need to define puts as extern.
```

A simpler wrapper: a function that calls `printf` with a fixed format string and a value.

```
extern printf
global print_int
print_int:
    sub     rsp, 8                ; align stack before call
    lea     rdi, [rel int_fmt]
    mov     esi, edi              ; first arg was in RDI, now moved to ESI
    xor     eax, eax
```

```
    call    printf
    add     rsp, 8
    ret
```

Function with Many Parameters

When a function has more than six integer arguments, the extra ones go on the stack. The callee accesses them at `[rbp+16]`, `[rbp+24]`, etc., after the standard prologue.

Variable Argument Functions (Varargs)

The System V ABI supports varargs through the C standard library's `va_list` mechanism. Writing a true varargs function in pure assembly is complex and usually not necessary; you can rely on C varargs if you link with libc. For most assembly tasks, fixed arguments are sufficient.

Parameter Validation

Always check pointer arguments for `NULL` before dereferencing. Bounds checking for buffer sizes can be enforced manually.

```
safe_strlen:
    test    rdi, rdi
    jz      .null_ptr
    ; real code
.null_ptr:
    xor     eax, eax
    ret
```

Preserving Non-Volatile Registers

The registers `RBX, RBP, R12-R15` are non-volatile in System V ABI. If a function modifies them, it must save them on the stack and restore them before returning. Typically, they are pushed after establishing the frame.

Calling Convention Compliance Check

Before finalising a function, verify:

1. Are the first six integer arguments in `RDI, RSI, RDX, RCX, R8, R9`? Are they not accidentally overwritten?
2. If the function calls another, is `RSP` 16-byte aligned at the call site?
3. Are all non-volatile registers preserved?
4. For functions that return a value, is it in `RAX` (or XMM0)?
5. For functions with stack arguments, are the offsets from RBP correct (considering the return address and saved RBP)?

Complete Design Example: Recursive Factorial

The following program implements a recursive factorial function and prints the result using `printf`. It demonstrates all the discussed concepts.

Code: listing23-6.asm — Recursive factorial

```
; listing23-6.asm
; Assemble: nasm -f elf64 -o listing23-6.o listing23-6.asm
; Link:     gcc -no-pie -o listing23-6 listing23-6.o

bits 64
default rel

extern printf
extern exit

section .data
fmt db 'Factorial of %d is %ld', 10, 0

section .text
global main

; factorial(n) -> RAX
; n in EDI (signed int, but we'll use 64-bit)
factorial:
    push    rbp
    mov     rbp, rsp
    sub     rsp, 16             ; align stack for recursion calls

    cmp     rdi, 1
    jg      .recurse
    mov     rax, 1
    jmp     .done

.recurse:
    push    rdi                 ; save n (RDI is volatile, we need it after call)
    dec     rdi
    call    factorial
    pop     rdi
    imul    rax, rdi            ; result = factorial(n-1) * n
.done:
    mov     rsp, rbp
    pop     rbp
    ret

main:
    push    rbp
    mov     rbp, rsp
    sub     rsp, 16

    ; compute factorial of 5
    mov     edi, 5
    call    factorial
    mov     r12, rax            ; save result in non-volatile R12
```

```
    ; print result
    lea     rdi, [rel fmt]
    mov     esi, 5
    mov     rdx, r12
    xor     eax, eax
    call    printf

    xor     edi, edi
    call    exit
```

This program calls `printf` to format the result and prints it. The function `factorial` is recursive, preserving the argument `RDI` by pushing it on the stack before the recursive call and popping it after. Non-volatile register `R12` is used in `main` to hold the result across the `printf` call, demonstrating proper register usage.

Warning

The ABI designates RDI as volatile. A recursive function cannot rely on RDI surviving a call. In the example, we manually save RDI with `PUSH`/`POP`. Alternatively, we could use a non-volatile register to hold the parameter across recursion, which is often more efficient.

Putting It All Together

Designing functions in assembly requires a meticulous application of the calling convention. The patterns presented here — the prologue/epilogue skeleton, the register and stack argument routing, the careful preservation of non-volatile registers, and the systematic validation of parameters — form a solid foundation. By following these patterns, your assembly functions will integrate seamlessly with the C library and with each other, allowing you to build large, maintainable programs in pure NASM on Linux.

Part V

LINUX x64 SYSTEM PROGRAMMING

24

System V AMD64 Calling Convention

In This Chapter

Every function call in a Linux x86-64 process follows a single, well-defined set of rules known as the System V AMD64 Application Binary Interface (ABI). These rules dictate which registers hold arguments, where the stack is used, how values are returned, what each side of the call must preserve, and the precise alignment required of the stack pointer. Understanding and obeying this convention is not optional; a single violation can cause crashes deep inside C library code that are difficult to diagnose. This chapter presents the complete, authoritative description of the System V AMD64 calling convention, derived from the official ABI supplement, the Intel® 64 and IA-32 Architectures Software Developer's Manual, and the AMD64 Architecture Programmer's Manual. Every rule is accompanied by a NASM example that you can assemble, link, and test on Fedora 43.

24.1 Register Usage Rules

The System V AMD64 ABI divides the general-purpose registers into **calling-saved** (callee-saved) and **call-clobbered** (caller-saved). The terminology "volatile" and "non-volatile" is sometimes used synonymously.

24.1.1 Call-Clobbered (Callee-Free) Registers

These registers may be freely modified by any function. The caller cannot expect their values to survive a `CALL`. They are:

- `RAX`, `RCX`, `RDX`, `RSI`, `RDI`
- `R8` through `R11`
- `XMM0` – `XMM15` (the upper 128 bits of YMM/ZMM are also caller-saved)

Additionally, the condition code flags and the floating-point status registers are volatile.

24.1.2 Callee-Saved Registers

A function that intends to use any of these registers must save the original value on the stack and restore it before returning. The callee-saved integer registers are:

- RBX, RBP, R12–R15

RSP is obviously reserved as the stack pointer and is maintained by the call/return mechanism. RBP may optionally be used as a frame pointer; if used, it must be saved and restored like any callee-saved register.

The floating-point registers XMM8 – XMM15 (and their upper halves) are callee-saved (but this detail is rarely required in simple integer programs).

24.1.3 Special-Purpose Roles

- RSP – Stack pointer; must be 16-byte aligned before any CALL instruction.
- RBP – Optionally used as a frame pointer. It is callee-saved.
- RIP – Instruction pointer; not directly accessible except via LEA, CALL, etc.
- Direction Flag (DF) – Must be cleared (0) before calling any C library function; in practice, the ABI assumes it is cleared. If a function uses string instructions, it must clear DF before returning.

Code: listing24-1.asm — Callee-saved register preservation

```
; listing24-1.asm
; Assemble: nasm -f elf64 -o listing24-1.o listing24-1.asm
; Link:     ld -o listing24-1 listing24-1.o

bits 64
default rel

section .text
global _start

; A leaf function that uses callee-saved registers.
demo_func:
    push    rbx
    push    r12
    mov     rbx, 0x1234
    mov     r12, 0x5678
    ; ... work ...
    pop     r12
    pop     rbx
    ret

_start:
    call    demo_func           ; RAX may be clobbered
    mov     eax, 60             ; sys_exit
```

```
    xor     edi, edi
    syscall
```

The function `demo_func` modifies `RBX` and `R12` and therefore saves and restores them. It freely uses other registers without saving.

Warning

Never assume a call-clobbered register survives a call. After you call any function, `RAX`, `RCX`, `RDX`, `RSI`, `RDI`, `R8`–`R11` may contain arbitrary values. If you need their original values, store them in callee-saved registers or in memory before the call.

24.2 Parameter Passing

The first six integer or pointer arguments are passed in registers; any additional arguments are passed on the stack.

24.2.1 Integer and Pointer Arguments

The assignment, in order, is:

`RDI`, `RSI`, `RDX`, `RCX`, `R8`, `R9`

Each argument is extended by the caller to the full register width (zero- or sign-extension as appropriate). The callee typically works with the full 64-bit value.

```
; Function: long add_four(long a, long b, long c, long d);
; RDI = a, RSI = b, RDX = c, RCX = d
add_four:
    lea     rax, [rdi + rsi]
    add     rax, rdx
    add     rax, rcx
    ret
```

24.2.2 Floating-Point Arguments

The first eight floating-point arguments go in `XMM0` – `XMM7`, one per argument. They are used independently of the integer registers: if a function has a mix, the integer arguments use the integer registers and the floating-point arguments use the XMM registers, each occupying its own slot. For example, `void foo(int a, double b, int c, double d)` passes `a` in RDI, `b` in XMM0, `c` in RSI, `d` in XMM1 (integer and XMM slots are counted independently). The exact rules are complex; for most assembly programming, knowing that the first six integers use the above registers suffices.

24.2.3 Stack Arguments

Arguments beyond the sixth (and any structure passed by value that doesn't fit in two 8-byte registers) are pushed onto the stack, in reverse order (rightmost first), by the caller. The callee accesses them at positive offsets from `RBP` (or `RSP`) after the standard prologue.

24.2.4 Example: Function with Eight Integer Arguments

Code: listing24-2.asm — Passing eight arguments

```
; listing24-2.asm
; Assemble: nasm -f elf64 -o listing24-2.o listing24-2.asm
; Link:     gcc -no-pie -o listing24-2 listing24-2.o

bits 64
default rel

extern printf
extern exit

section .data
fmt db 'Sum of eight: %ld', 10, 0

section .text
global main

; sum8(a..h) -> a+...+h
; RDI..R9 contain a..f, g,h on stack
sum8:
    push    rbp
    mov     rbp, rsp
    mov     rax, rdi
    add     rax, rsi
    add     rax, rdx
    add     rax, rcx
    add     rax, r8
    add     rax, r9
    add     rax, qword [rbp+16]   ; g (right above return addr)
    add     rax, qword [rbp+24]   ; h
    pop     rbp
    ret

main:
    push    rbp
    mov     rbp, rsp
    sub     rsp, 16             ; align stack for calls

    ; Push 8th and 7th args
    push    80                  ; h
    push    70                  ; g
    mov     edi, 10             ; a
    mov     esi, 20             ; b
    mov     edx, 30             ; c
    mov     ecx, 40             ; d
    mov     r8d, 50             ; e
    mov     r9d, 60             ; f
    call    sum8                ; RAX = 360
    add     rsp, 16             ; clean up pushed args

    ; print result
    lea     rdi, [rel fmt]
```

```
    mov     rsi, rax
    xor     eax, eax
    call    printf

    xor     edi, edi
    call    exit
```

The caller pushes arguments 8 and 7, calls `sum8`, and then removes the pushed arguments. Note that the stack must still be 16-byte aligned before the call. After the two pushes (16 bytes), RSP stays aligned because the initial alignment was maintained by `sub rsp, 16` earlier.

24.3 Return Values

Return values follow a simple pattern:

- **Integer / pointer:** returned in `RAX` (8-bit, 16-bit, 32-bit values in AL/AX/EAX).
- **Floating-point:** returned in `XMM0`.
- **Structures 16 bytes:** returned in `RAX` (and possibly `RDX` for second 8 bytes, if needed).
- **Larger structures:** the caller passes a hidden pointer as the first argument (before the explicit parameters), and the callee writes the result through that pointer. The pointer is returned in `RAX`.

24.3.1 Example: Returning a Structure via Hidden Pointer

Code: listing24-3.asm — Hidden pointer return

```
; listing24-3.asm
bits 64
default rel

section .text
global get_struct

; struct large { long x; long y; long z; };
; get_struct(struct large *result)  ; RDI = result pointer
get_struct:
    mov     qword [rdi],     100
    mov     qword [rdi+8],   200
    mov     qword [rdi+16],  300
    mov     rax, rdi              ; return pointer (ABI)
    ret
```

24.4 Caller/Callee Responsibilities

24.4.1 Caller

Before a call, the caller must:

1. Place integer/pointer arguments in RDI, RSI, RDX, RCX, R8, R9 (left to right). Float args in XMM registers.
2. Push any remaining arguments onto the stack in reverse order.
3. Ensure that RSP is 16-byte aligned immediately before the CALL instruction. After the call pushes the 8-byte return address, RSP will be 8 mod 16 inside the callee on entry.

After the call, the caller must remove any stack-passed arguments (e.g., `add rsp, N`).

24.4.2 Callee

Upon entry, the callee may:

- Use any call-clobbered registers without saving them.
- Save and restore any callee-saved registers it modifies.
- Set up its own stack frame (optional); if it calls another function, it must realign the stack to 16-byte alignment before the call (since on entry RSP is 8 mod 16).

Before returning, the callee must restore any saved registers, restore RSP to its original position (pointing to the return address), and execute RET. The callee does **not** remove stack-passed arguments; the caller is responsible.

24.5 Stack Alignment Rules

The stack must be 16-byte aligned at the point of a CALL instruction. This mean before the call, RSP is a multiple of 16. After the call, RSP is 8 mod 16 inside the callee.

To realign the stack for a subsequent call, the callee must subtract an odd multiple of 8 from RSP. The classic prologue `push rbp` (which subtracts 8) followed by `sub rsp, N` where N is a multiple of 16 achieves proper alignment. For example:

```
my_func:
    push    rbp
    mov     rbp, rsp
    sub     rsp, 16         ; (multiple of 16) keeps RSP 16-byte aligned
    ; ... maybe calls ...
    mov     rsp, rbp
    pop     rbp
    ret
```

If a function does not use a frame pointer, it may directly subtract an odd multiple of 8. For a leaf function that makes no calls and does not use SSE on the stack, alignment is not strictly required (though still good practice if any SSE instructions are used).

There is **no mandatory shadow space** like in Windows; the 128-byte area below RSP is the **red zone**, which may be used by leaf functions without adjusting RSP, but must not be used if the function makes any calls or signals may interrupt. The next chapter discusses the red zone in detail.

24.5.1 Example: Verifying Alignment with a Debugger

Code: listing24-4.asm — Alignment check

```
; listing24-4.asm
; Assemble: nasm -f elf64 -o listing24-4.o listing24-4.asm
; Link:     gcc -no-pie -o listing24-4 listing24-4.o

bits 64
default rel

extern printf
extern exit

section .data
fmt db 'RSP alignment OK', 10, 0

section .text
global main
main:
    push    rbp
    mov     rbp, rsp
    sub     rsp, 16           ; now RSP is 16-byte aligned

    ; RSP should end with 0 before calling printf
    lea     rdi, [rel fmt]
    xor     eax, eax
    call    printf            ; breakpoint: check RSP

    xor     edi, edi
    call    exit
```

In GDB, set a breakpoint at `call printf` and check RSP with `info registers rsp`. The value will end with 0.

24.6 Callee-Saved Register Preservation

A function that uses RBX, RBP, R12–R15 must save their original values and restore them before returning. The push/pop order must be symmetric. The typical pattern is to push them after establishing the frame.

Code: listing24-5.asm — Using callee-saved registers

```
; listing24-5.asm
bits 64
default rel

section .text
global my_func
my_func:
    push    rbp
    mov     rbp, rsp
    push    rbx           ; save RBX
```

```
    push    r12             ; save R12
    sub     rsp, 16         ; allocate locals, maintain alignment

    ; ... use RBX, R12 ...
    mov     rbx, 5
    mov     r12, 10

    add     rsp, 16
    pop     r12
    pop     rbx
    pop     rbp
    ret
```

Putting It All Together

The System V AMD64 calling convention is lean and efficient. By respecting the register classifications, argument order, return mechanisms, and the 16-byte stack alignment, your assembly functions will interoperate perfectly with the vast ecosystem of C libraries on Linux. The next chapter will delve deeper into stack alignment and the special red zone feature unique to this ABI.

25

Stack Alignment and The Red Zone

In This Chapter

Two crucial aspects of the System V AMD64 ABI are the 16-byte stack alignment requirement and the *red zone*, a 128-byte area below the stack pointer that functions can use without adjustment. Understanding these rules in depth prevents subtle alignment faults and enables efficient leaf function design. This chapter explains the alignment mechanics, the purpose and restrictions of the red zone, how to design correct function prologues, and the most common mistakes. All information is drawn from the System V AMD64 ABI supplement, the Intel and AMD architecture manuals, and extensive testing on Fedora 43.

25.1 Stack Alignment Deep Dive

The ABI mandates that before any `CALL` instruction, `RSP` must be a multiple of 16. After the call, the return address is pushed, making `RSP + 8 = 16n`. Inside the callee on entry, `RSP  8 (mod 16)`.

25.1.1 Realigning for Sub-calls

To call another function, the callee must ensure the stack is 16-byte aligned again. The most common method is the standard prologue:

```
push    rbp
mov     rbp, rsp
```

After the `push rbp`, RSP becomes aligned (because entry RSP was 8 mod 16, subtracting 8 makes it 0 mod 16). Then any subtraction must be a multiple of 16 to preserve alignment. For example, `sub rsp, 16` keeps the alignment; `sub rsp, 8` would misalign it.

A leaf function that makes no calls does not need to align the stack (and doesn't even need a frame), as long as it doesn't use any instructions that require 16-byte aligned stack memory (like `movaps`).

25.1.2 Aligning Locals for SSE

If a function uses SSE instructions that require aligned memory access, the addresses of local variables must be aligned. Since RSP is 16-byte aligned after a proper prologue, any variable placed at an offset that is a multiple of 16 from RSP will satisfy alignment. It is the programmer's responsibility to ensure the correct offsets.

25.1.3 Verification with GDB

The following program prints a message using `printf`, relying on proper alignment. A misalignment crash can be reproduced by removing the `sub rsp, 8` before the call (since entry to main already has RSP 8 mod 16, calling printf without realigning will likely crash on modern glibc).

Code: listing25-1.asm — Alignment crash demonstration

```
; listing25-1.asm
; This will crash if you comment out the sub rsp,8 (since main's entry stack is misaligned)
; Assemble: nasm -f elf64 -o listing25-1.o listing25-1.asm
; Link:     gcc -no-pie -o listing25-1 listing25-1.o

bits 64
default rel

extern printf
extern exit

section .data
msg db 'Alignment works',10,0

section .text
global main
main:
    sub     rsp, 8          ; realign stack (entry RSP 8 mod 16)
    lea     rdi, [rel msg]
    xor     eax, eax
    call    printf
    add     rsp, 8
    xor     edi, edi
    call    exit
```

25.2 The Red Zone

The red zone is a unique feature of the AMD64 ABI: the 128 bytes of memory immediately below `RSP` (i.e., addresses `RSP-128` through `RSP-1`) are reserved and can be used by a *leaf function* without adjusting the stack pointer. The kernel guarantees that signal handlers will not use this region, so it is safe for temporary storage.

25.2.1 When the Red Zone Can Be Used

A leaf function (a function that makes no `CALL` instructions) can store local variables directly in the red zone, avoiding the need for `sub rsp, N`. For example:

Code: listing25-2.asm — Red zone usage in leaf

```
; listing25-2.asm
bits 64
default rel

section .text
global leaf_sum
; leaf_sum(a,b) -> a + b
; RDI, RSI
leaf_sum:
    ; store parameters into red zone (optional demo)
    mov     [rsp-8],  rdi
    mov     [rsp-16], rsi
    mov     rax, rdi
    add     rax, rsi
    ret
```

No stack adjustment is needed. This function is leaf, so the red zone is safe.

25.2.2 When the Red Zone Must Not Be Used

A function that calls *any* other function must not rely on the red zone, because the called function may itself use the red zone and overwrite the data. In a non-leaf function, you must allocate a proper stack frame (with `sub rsp, ...`). Signal handlers and the kernel also do not use the red zone, but asynchronous signals can occur, and if you are in a leaf function using the red zone, the signal handler will not touch it; so leaf is safe. But if you make a call, the called function may use its own red zone, clobbering yours.

25.2.3 Red Zone and System Calls

The kernel does not use the red zone, but the CPU's interrupt stack table might be used during interrupts. However, the user-mode red zone is protected from signal handlers by the kernel. Making a system call does not affect the red zone; you can still use it if you are a leaf, but the system call itself doesn't change it. Actually, during a system call, the kernel may use the kernel stack, not the user stack; the red zone is still valid for the user process. So leaf functions can safely use the red zone even if they perform system calls? The ABI says that the signal handler will not modify the red zone, but an interrupt could use the kernel stack; the user's red zone is not touched. So it's safe. However, some documentation states that the red zone is only reserved for leaf functions and should not be used if the function is not leaf, i.e., if it calls other functions, because the called function could overwrite the area. So in a leaf function, it's fine.

The typical use: a fast leaf function can use `[rsp-N]` to store temporaries without adjusting RSP. This reduces code size and improves performance.

25.2.4 Example: Non-Leaf Function Incorrectly Using Red Zone

The following would be dangerous:

Code: listing25-3.asm — Bad red zone usage

```
; listing25-3.asm -- BAD, will cause obscure bugs if called
bits 64
default rel

extern printf

section .data
fmt db '%d',10,0

section .text
global bad_func
bad_func:
    ; assume RDI is some value, store it in red zone
    mov     [rsp-8], rdi
    ; call printf -- this call may clobber red zone!
    lea     rdi, [rel fmt]
    mov     esi, 42
    xor     eax, eax
    call    printf
    mov     rax, [rsp-8]   ; may be corrupted
    ret
```

The store at `[rsp-8]` is in the red zone. When `printf` is called, it (or functions it calls) may use the red zone and overwrite that data. The solution is to use a proper stack frame.

25.3 Function Prologue Design for Linux

The standard prologue for a function that calls others or uses callee-saved registers is:

```
push    rbp
mov     rbp, rsp
; optionally push other callee-saved registers
sub     rsp, N          ; N must be a multiple of 16, or (if you push a total of M registers, then
↪  N must be multiple of 16 - 8*M)
```

For a leaf function that doesn't use callee-saved registers and needs only a few locals, you may use the red zone instead of adjusting RSP.

25.4 Common Mistakes

- **Forgetting to align the stack before calling libc.** This is the most frequent crash. The solution is to always realign before calls using `sub rsp, 8` (or an odd multiple) if you haven't already.
- **Using red zone in non-leaf functions.** If your function calls others, allocate a proper frame.

- **Not preserving callee-saved registers.** If you modify RBX, RBP, R12–R15 without saving, the caller will see corrupted values.
- **Misplaced stack arguments (offsets from RBP).** After the prologue, the return address is at `[rbp+8]`, and stack arguments start at `[rbp+16]`. Be careful with pushes that shift things.
- **Assuming shadow space exists.** There is none; you must allocate exactly the space you need.

25.4.1 Putting It All Together

The stack alignment and the red zone are fundamental concepts of the Linux x86-64 ABI. By consistently applying the prologue patterns and respecting the red zone's restrictions, you will write robust assembly functions that integrate flawlessly with the rich glibc environment. The next chapter will apply these rules to calling common C library functions from NASM.

26

Calling C Library Functions from NASM

In This Chapter

The power of Linux assembly programming is greatly amplified by linking with the standard C library (glibc). Functions like `printf`, `malloc`, `exit`, and many others become directly callable from your NASM code, provided you adhere to the System V ABI and correctly instruct the linker. This chapter explains how to declare external C library symbols, how to link with `gcc` (or `ld`), and presents fully working examples of console output, memory allocation, and process control. Every example has been tested with NASM 2.16 and GCC on Fedora 43.

26.1 Declaring External Functions

Use the `extern` directive to declare any C library function you intend to call:

```
extern printf
extern malloc
extern free
extern exit
extern puts
```

These symbols will be resolved at link time. In the object file, NASM records them as undefined with relocation entries. When you link with `gcc`, it automatically searches the C library (`libc.so`) and creates the necessary PLT entries.

26.1.1 How the Dynamic Linker Works

For each external function, the linker generates a Procedure Linkage Table (PLT) stub and a Global Offset Table (GOT) entry. Your code calls the PLT stub, which loads the actual address from the

GOT. At load time, the dynamic linker (`ld-linux-x86-64.so.2`) resolves the symbol and fills the GOT. This process is completely transparent; you simply call `printf` as if it were a normal label.

26.2 Linking Workflow

The recommended way to create an executable that uses the C library is to use `gcc` as the linker driver. After assembling with NASM:

```
nasm -f elf64 -o prog.o prog.asm
```

If your program defines a `main` function (following the C convention), link with:

```
gcc -no-pie -o prog prog.o
```

The `-no-pie` flag produces a non-position-independent executable, which is simpler and often preferred for small assembly programs. If you want to build a position-independent executable (PIE), omit `-no-pie` and ensure your code uses RIP-relative addressing for all global data (`default rel`).

If your program uses `_start` instead of `main` and you still want to call libc functions, you must link manually with the C library and provide the dynamic linker path:

```
ld -o prog prog.o -lc --dynamic-linker=/lib64/ld-linux-x86-64.so.2
```

However, using `main` and linking with `gcc` is much simpler.

26.3 Example: Console Output with printf

The following program prints a formatted string to the console using `printf` and exits using `exit`.

Code: listing26-1.asm — printf and exit

```
; listing26-1.asm
; Assemble: nasm -f elf64 -o listing26-1.o listing26-1.asm
; Link:     gcc -no-pie -o listing26-1 listing26-1.o

bits 64
default rel

extern printf
extern exit

section .data
msg db 'Hello, Fedora! Value: %d', 10, 0

section .text
global main
main:
    push    rbp
    mov     rbp, rsp
    sub     rsp, 16             ; align stack for calls

    lea     rdi, [rel msg]      ; first arg: format string
    mov     esi, 42             ; second arg: integer value
    xor     eax, eax            ; number of XMM registers used (0)
```

```
    call    printf

    xor     edi, edi            ; exit code 0
    call    exit                ; never returns
```

The `xor eax, eax` before calling `printf` is mandatory because `printf` is a variadic function. The ABI requires that `AL` is set to the number of floating-point arguments passed in XMM registers. For `printf`, this is often 0 if no floats are used. If you forget to set AL, the output may be garbled and the program might crash.

Warning

Variadic functions. Always set `AL` to the count of XMM arguments before calling functions like `printf`, `scanf`, `sprintf`. For non-variadic functions (like `puts`, `exit`), you don't need to set AL.

26.4 Example: Dynamic Memory with malloc and free

You can allocate memory from the heap using `malloc` and release it with `free`, just like in C.

Code: listing26-2.asm — malloc and free

```
; listing26-2.asm
; Assemble: nasm -f elf64 -o listing26-2.o listing26-2.asm
; Link:     gcc -no-pie -o listing26-2 listing26-2.o

bits 64
default rel

extern malloc
extern free
extern puts
extern exit

section .data
msg db 'Memory allocated successfully', 0

section .text
global main
main:
    push    rbp
    mov     rbp, rsp
    sub     rsp, 16

    ; allocate 64 bytes
    mov     edi, 64
    call    malloc
    test    rax, rax            ; check for NULL
    jz      .fail

    ; use the memory
```

```
    mov     byte [rax], 'A'

    ; free it
    mov     rdi, rax
    call    free

    lea     rdi, [rel msg]
    call    puts                ; print success
    xor     edi, edi
    call    exit

.fail:
    mov     edi, 1
    call    exit
```

`malloc` returns a pointer in `RAX`, or NULL on failure. `puts` outputs a simple string and appends a newline; it expects the string pointer in `RDI`.

26.5 Calling System Call Wrappers vs. Direct Syscalls

The C library provides thin wrappers around Linux system calls, like `write`, `read`, `open`, etc. You can call these wrappers from assembly just like any other function, using the same ABI:

```
extern write
extern read
extern close
; ssize_t write(int fd, const void *buf, size_t count);
; RDI = fd, RSI = buf, RDX = count
```

Linking with `gcc` will resolve these symbols to the C library's stubs. If you prefer to avoid any library dependency, you can make direct system calls using the `syscall` instruction, as shown in earlier chapters. The choice depends on whether you need libc's buffering, portability, or error handling (`errno`).

26.6 Linking with Multiple Object Files and Libraries

To link multiple object files together with `gcc`:

```
gcc -no-pie -o prog main.o util.o -lm
```

The `-lm` flag links the math library. The order of object files and libraries matters for some linkers; with GCC, libraries usually come after objects.

If you are mixing C and assembly, simply include the C object files on the same command line. Ensure the assembly functions obey the calling convention and any global symbols you intend to export are declared `global`.

26.7 Common Pitfalls

- **Missing `extern` declaration** leads to "undefined reference" linker errors.

- **Forgetting to set AL for variadic functions** causes `printf` to misinterpret the stack and crash or produce garbage.
- **Stack misalignment before a C call** results in a segmentation fault deep inside glibc.
- **Using `_start` without properly initializing libc:** if you link with `gcc` and define `main`, the C runtime initialisation is performed automatically. If you use `_start` and try to call `printf` without linking with the startup code, it will crash because libc requires certain initialisation. Always either define `main` and link with GCC, or use only direct system calls if you write `_start` and want a static executable.
- **Position-dependent code in PIE executables:** Using absolute addresses like `mov rax, myvar` instead of `[rel myvar]` will cause a relocation error or a crash. Use `default rel` and RIP-relative addressing.

Putting It All Together

Calling C library functions from NASM is straightforward and opens up a vast array of functionality. By respecting the System V ABI, linking properly with GCC, and being mindful of variadic function requirements, you can write powerful hybrid programs that leverage the entire Linux userspace. The rest of this part will explore system-level programming with direct system calls and the Linux kernel interface, building on these fundamentals.

27

Working with Shared Libraries and Dynamic Loading

In This Chapter

Dynamic shared objects (shared libraries, `.so` files) are the backbone of code reuse on Linux. Functions from the C library, the math library, and thousands of other libraries are all loaded dynamically. This chapter explains the concept of shared libraries, how the linker resolves calls to them, how to obtain function addresses for indirect calls, and how to load modules and resolve symbols at runtime using `dlopen` and `dlsym`. Every technique is demonstrated with complete NASM programs that you can assemble and run on Fedora 43.

27.1 Shared Library Concept

A shared library is an ELF file (type ET_DYN) that contains position-independent code and data. When a program is loaded, the dynamic linker (`ld-linux-x86-64.so.2`) maps the required libraries into the process address space. Multiple programs can share the same physical pages of a library, saving memory and disk space.

Benefits:

- **Code reuse.** Common functionality such as `printf`, `malloc`, and `sin` is provided by libraries rather than duplicated in every executable.
- **Updatability.** A library can be updated independently of the programs that use it, as long as its ABI remains compatible.
- **Modularity.** Programs can be extended with plugins loaded at runtime via `dlopen`.

Shared libraries export their symbols through a dynamic symbol table. When you link against a library, the linker records which symbols are needed and creates a Procedure Linkage Table (PLT) and Global Offset Table (GOT) to enable lazy binding.

27.2 Dynamic Linking (PLT and GOT)

When you declare `extern printf` and call `printf` in your NASM code, the assembler emits a `CALL` with a PC-relative relocation. The linker resolves this call to a PLT entry—a small stub that jumps through a pointer in the GOT. Initially the GOT entry points back to the dynamic linker, which resolves the address of `printf` on the first call (lazy binding). Subsequent calls go directly to the function.

This process is transparent; you simply call the external function as if it were any other label. The linker, when invoked via `gcc` (or `ld` with `-lc`), sets up the necessary PLT and GOT entries automatically.

Obtaining a Function Pointer

Sometimes you need to store the address of a dynamically linked function, for example to build a table of function pointers. Since the address is not known until load time, you can load it from the GOT. With NASM and a shared library, you can access the GOT entry using the `@GOTPCREL` suffix:

```
extern printf
lea     rax, [printf@GOTPCREL]   ; rax = address of GOT entry for printf
mov     rax, [rax]               ; rax = actual address of printf
```

This is analogous to using `__imp_symbol` on Windows. In practice, for simple calls the PLT-based call is preferred.

Example: Function Pointer Table

The following program stores function pointers in a table and calls them indirectly.

Code: listing27-1.asm — Function pointer table via GOT

```
; listing27-1.asm
; Assemble: nasm -f elf64 -o listing27-1.o listing27-1.asm
; Link:     gcc -no-pie -o listing27-1 listing27-1.o -ldl
bits 64
default rel

extern printf
extern exit

section .data
fmt  db 'Called function %d',10,0

section .bss
func_table resq 2          ; two pointers

section .text
```

```
global main
main:
    push    rbp
    mov     rbp, rsp
    sub     rsp, 16

    ; load address of printf from GOT
    lea     rax, [printf@GOTPCREL]
    mov     rax, [rax]
    mov     [rel func_table], rax

    ; also store a pointer to exit (via GOT)
    lea     rax, [exit@GOTPCREL]
    mov     rax, [rax]
    mov     [rel func_table+8], rax

    ; call printf via the table
    lea     rdi, [rel fmt]
    mov     esi, 1
    xor     eax, eax
    call    qword [rel func_table]      ; indirect call through pointer

    ; call exit via the table
    xor     edi, edi
    call    qword [rel func_table+8]

    leave
    ret
```

After assembling and linking, the program prints “Called function 1” and exits normally. The function pointers are resolved through the GOT at load time.

27.3 Runtime Dynamic Loading: dlopen, dlsym, dlclose

The dynamic linker also provides an API for loading shared libraries and resolving symbols at runtime. The three key functions (declared in `<dlfcn.h>`) are:

- `void *dlopen(const char *filename, int flags)` — loads the specified library and returns a handle (or NULL on failure).
- `void *dlsym(void *handle, const char *symbol)` — returns the address of a symbol from the loaded library (or NULL if not found).
- `int dlclose(void *handle)` — releases the library when it is no longer needed.

Error information is retrieved with `dlerror()`, which returns a string describing the most recent `dlopen`, `dlsym`, or `dlclose` error.

These functions reside in the dynamic loader library, `libdl`. When linking, add `-ldl`. Alternatively, on some systems `dlopen` is part of glibc and does not require a separate library, but adding `-ldl` is portable.

Example: Loading the Math Library and Calling sin

The following program loads `libm.so` at runtime, obtains a pointer to `sin`, calls it, and prints the result. It does not link against `libm` at build time.

Code: listing27-2.asm — Runtime linking of libm

```
; listing27-2.asm
; Assemble: nasm -f elf64 -o listing27-2.o listing27-2.asm
; Link:     gcc -no-pie -o listing27-2 listing27-2.o -ldl
bits 64
default rel

extern dlopen
extern dlsym
extern dlclose
extern dlerror
extern printf
extern exit

section .data
lib_name db 'libm.so.6',0
fn_name  db 'sin',0
fmt      db 'sin(1.0) = %f',10,0

section .bss
hLib     resq 1
pSin     resq 1

section .text
global main
main:
    push    rbp
    mov     rbp, rsp
    sub     rsp, 32          ; align and space for locals

    ; load the library
    mov     edi, 1           ; RTLD_LAZY (simplified: constant 1)
    lea     rsi, [lib_name]
    call    dlopen
    test    rax, rax
    jz      .fail
    mov     [rel hLib], rax

    ; get the function address
    mov     rdi, rax
    lea     rsi, [fn_name]
    call    dlsym
    test    rax, rax
    jz      .fail_close
    mov     [rel pSin], rax

    ; call sin(1.0)
    movsd   xmm0, qword [rel one]   ; load 1.0
    call    rax                      ; result in xmm0
```

```
    ; print result
    lea     rdi, [rel fmt]
    mov     eax, 1              ; one floating-point argument
    call    printf

    ; close the library
    mov     rdi, [rel hLib]
    call    dlclose

    xor     edi, edi
    call    exit

.fail_close:
    mov     rdi, [rel hLib]
    call    dlclose
.fail:
    ; print error message
    call    dlerror
    mov     rdi, rax
    call    puts
    mov     edi, 1
    call    exit

section .data
one dq 1.0
```

No `extern sin` is needed; the function is resolved entirely at runtime. This technique is the foundation of plugin systems and scriptable applications.

Warning

Error handling with dlerror. `dlerror` returns a pointer to a string describing the most recent error. Its result is undefined if called more than once after no error has occurred, so always call it immediately after a failed call and copy the result if needed.

Checking for Missing Functionality

You can use `dlsym` to test for optional functions. For example, a program may try to load a newer library function, and if `dlsym` returns NULL, fall back to a compatible implementation.

27.4 Error Handling for Dynamic Loading

All three functions return NULL (for pointers) or non-zero (for `dlclose`) on failure. Retrieving an error string is done with `dlerror`. The typical pattern is:

```
call    dlopen
test    rax, rax
jnz     .ok
call    dlerror
; now RAX points to an error message string
; ... display or log ...
```

Because `dlerror` resets the error state after being called, you must call it exactly once per error.

Putting It All Together

Shared libraries and dynamic loading provide unparalleled flexibility. You can use implicit linking via PLT/GOT for standard libraries, or employ `dlopen` / `dlsym` to build extensible applications that load code on demand. The next chapter applies these techniques to terminal I/O and console application design.

28

Console Applications in Assembly

In This Chapter

The terminal (console) is the primary user interface for many system programs and the natural environment for learning assembly on Linux. This chapter covers the core patterns of writing console applications: obtaining and using the standard input/output file descriptors, outputting text, reading input, formatting numbers into strings, and designing a command loop. The examples use a mix of direct system calls (for the pure assembly approach) and C library functions (for convenience), all within the System V AMD64 calling convention. Every program is complete and tested on Fedora 43.

28.1 Entry Point Design for a Console Program

On Linux, a console program can be written in two ways:

- As a pure assembly program with an `_start` entry point, using only system calls. This yields a minimal, dependency-free executable.
- As a program that defines `main` and is linked with the C library via `gcc`, giving access to `printf`, `scanf`, etc.

Both styles are valid; this chapter shows examples of both. For pure system-call programs, the entry point has no stack arguments; the command line can be read from the initial stack layout. For `main` programs, `gcc` sets up the standard C environment.

28.2 Console Output Using write System Call

The simplest way to output text is the `write` system call (number 1). It writes raw bytes to a file descriptor. The standard output descriptor is 1.

Code: listing28-1.asm — Direct console output

```
; listing28-1.asm
; Assemble: nasm -f elf64 -o listing28-1.o listing28-1.asm
; Link:     ld -o listing28-1 listing28-1.o
bits 64
default rel

SYS_WRITE equ 1
SYS_EXIT  equ 60
STDOUT    equ 1

section .data
msg db 'Hello, terminal!', 10
len equ $ - msg

section .text
global _start
_start:
    mov     eax, SYS_WRITE
    mov     edi, STDOUT
    lea     rsi, [rel msg]
    mov     edx, len
    syscall

    mov     eax, SYS_EXIT
    xor     edi, edi
    syscall
```

Assemble and link; when run, it prints the message and exits. No libraries are needed. You can check the exit status with `echo $?` (should be 0).

28.3 Console Input Using read System Call

Reading from the terminal uses the `read` system call (number 0) on descriptor 0. It blocks until the user presses Enter (the terminal line discipline buffers input). The number of bytes actually read is returned in RAX; it can be less than the requested count.

Code: listing28-2.asm — Echo with direct syscalls

```
; listing28-2.asm
; Assemble: nasm -f elf64 -o listing28-2.o listing28-2.asm
; Link:     ld -o listing28-2 listing28-2.o
bits 64
default rel

SYS_READ  equ 0
```

```
SYS_WRITE equ 1
SYS_EXIT  equ 60
STDIN     equ 0
STDOUT    equ 1

section .data
prompt db 'Enter your name: ',0
plen   equ $ - prompt
newline db 10

section .bss
buffer resb 128

section .text
global _start
_start:
    ; write prompt
    mov     eax, SYS_WRITE
    mov     edi, STDOUT
    lea     rsi, [rel prompt]
    mov     edx, plen
    syscall

    ; read input
    mov     eax, SYS_READ
    mov     edi, STDIN
    lea     rsi, [rel buffer]
    mov     edx, 128
    syscall

    ; RAX = bytes read (including newline)
    mov     r8, rax                 ; save count

    ; echo back
    mov     eax, SYS_WRITE
    mov     edi, STDOUT
    lea     rsi, [rel buffer]
    mov     edx, r8d
    syscall

    ; newline (already included in input, so not necessary; added for clarity)
    ; mov     eax, SYS_WRITE
    ; mov     edi, STDOUT
    ; lea     rsi, [rel newline]
    ; mov     edx, 1
    ; syscall

    ; exit
    mov     eax, SYS_EXIT
    xor     edi, edi
    syscall
```

The input typically ends with a newline; echoing it back shows the user's typing and moves the cursor to the next line.

28.4 Formatting Output with the C Library

When you need to print formatted numbers or strings with placeholders, linking with the C library and using `printf` is far more convenient than writing your own conversion routines. The same applies to reading formatted input with `scanf`.

Code: listing28-3.asm — Using printf for formatted output

```
; listing28-3.asm
; Assemble: nasm -f elf64 -o listing28-3.o listing28-3.asm
; Link:     gcc -no-pie -o listing28-3 listing28-3.o
bits 64
default rel

extern printf
extern exit

section .data
fmt db 'The sum of %d and %d is %d.',10,0

section .text
global main
main:
    push    rbp
    mov     rbp, rsp
    sub     rsp, 16

    lea     rdi, [rel fmt]
    mov     esi, 15
    mov     edx, 27
    mov     ecx, 42           ; third argument
    xor     eax, eax          ; required for variadic printf
    call    printf

    xor     edi, edi
    call    exit
```

If you prefer to stay within pure assembly and avoid libc, you must implement your own integer-to-string conversion (see the `itoa` routine below).

Custom Integer-to-String Conversion

The following routine converts a 64-bit unsigned integer in RAX to a decimal string in a buffer, returning the length.

Code: itoa — unsigned integer to decimal ASCII

```
; itoa:  RAX = number, RDI = buffer (at least 21 bytes)
; returns RAX = string length, buffer updated
itoa:
    push    rbx
    push    rcx
    mov     rcx, 10
```

```
    mov     rbx, rdi
    add     rdi, 20
    mov     byte [rdi], 0
    std                     ; decrement RDI
.next_digit:
    xor     edx, edx
    div     rcx
    add     dl, '0'
    dec     rdi
    mov     [rdi], dl
    test    rax, rax
    jnz     .next_digit
    ; move string to start of buffer
    cld
    mov     rcx, rbx
    mov     rsi, rdi
    sub     rdi, rcx        ; length? actually compute length
    ; we'll compute length by scanning for null
    push    rdi
    mov     rdi, rbx
    mov     rsi, rsi
    cld
    ; copy the string to the beginning (if needed; we can just return RDI pointing to the
    ↪  number)
    ; Simpler: return pointer to the number string in RSI (or return RAX as pointer).
    pop     rax
    ; Actually store the string at the original buffer start:
    ; move the generated string to the start
    mov     rdi, rbx
    rep     movsb
    pop     rcx
    pop     rbx
    ret
```

You can call this `itoa` and then use `write` to output the resulting string.

28.5 Interactive Command Loop

An interactive program repeatedly prompts for input, reads a command, and dispatches to an appropriate handler. The following example implements a minimal calculator that reads two numbers, adds them, and prints the result, using only system calls and the custom `itoa` above.

Code: listing28-4.asm — Interactive calculator with syscalls

```
; listing28-4.asm
; Assemble: nasm -f elf64 -o listing28-4.o listing28-4.asm
; Link:     ld -o listing28-4 listing28-4.o

bits 64
default rel

SYS_READ   equ 0
```

```
SYS_WRITE  equ 1
SYS_EXIT   equ 60
STDIN      equ 0
STDOUT     equ 1

section .data
prompt1     db 'Enter first number: ',0
plen1       equ $ - prompt1
prompt2     db 'Enter second number: ',0
plen2       equ $ - prompt2
result_msg  db 'Sum = ',0
rlen        equ $ - result_msg
newline     db 10

section .bss
buffer      resb 32
buf2        resb 32
output      resb 256

section .text
global _start
_start:
    ; --- first prompt ---
    mov     eax, SYS_WRITE
    mov     edi, STDOUT
    lea     rsi, [rel prompt1]
    mov     edx, plen1
    syscall

    ; read first number
    mov     eax, SYS_READ
    mov     edi, STDIN
    lea     rsi, [rel buffer]
    mov     edx, 32
    syscall
    ; convert to integer
    lea     rdi, [rel buffer]
    call    atoi
    mov     r12, rax            ; save first number

    ; --- second prompt ---
    mov     eax, SYS_WRITE
    mov     edi, STDOUT
    lea     rsi, [rel prompt2]
    mov     edx, plen2
    syscall

    mov     eax, SYS_READ
    mov     edi, STDIN
    lea     rsi, [rel buffer]
    mov     edx, 32
    syscall
    lea     rdi, [rel buffer]
    call    atoi
```

```
    mov     r13, rax            ; second number

    ; compute sum
    add     r12, r13            ; sum in r12

    ; output "Sum = "
    mov     eax, SYS_WRITE
    mov     edi, STDOUT
    lea     rsi, [rel result_msg]
    mov     edx, rlen
    syscall

    ; convert sum to string
    mov     rax, r12
    lea     rdi, [rel output]
    call    itoa_simple         ; returns pointer in RSI, length in RDX
    ; write the number string
    mov     eax, SYS_WRITE
    mov     edi, STDOUT
    ; RSI=number string, RDX=length
    syscall

    ; newline
    mov     eax, SYS_WRITE
    mov     edi, STDOUT
    lea     rsi, [rel newline]
    mov     edx, 1
    syscall

    ; exit
    mov     eax, SYS_EXIT
    xor     edi, edi
    syscall

; --- helper: atoi (string in RDI, returns integer in RAX) ---
atoi:
    push    rbx
    xor     eax, eax
    xor     ebx, ebx
    mov     rcx, 10
.next_char:
    movzx   ebx, byte [rdi]
    test    bl, bl
    jz      .done
    cmp     bl, '0'
    jb      .done
    cmp     bl, '9'
    ja      .done
    sub     bl, '0'
    imul    rax, rax, 10
    add     rax, rbx
    inc     rdi
    jmp     .next_char
.done:
```

```
    pop     rbx
    ret

; --- helper: itoa_simple (RAX=number, RDI=buffer, returns pointer in RSI, length in RDX)
↪ ---
itoa_simple:
    push    rcx
    push    rdx
    mov     rcx, 10
    mov     rsi, rdi
    add     rsi, 20
    mov     byte [rsi], 0
    std
.next_digit2:
    xor     edx, edx
    div     rcx
    add     dl, '0'
    dec     rsi
    mov     [rsi], dl
    test    rax, rax
    jnz     .next_digit2
    cld
    sub     rdi, rsi
    neg     rdi                 ; length = original tail - start
    mov     rdx, rdi
    pop     rdi
    pop     rcx
    ret
```

This pure assembly calculator performs the entire I/O loop without any C library functions. The `atoi` and `itoa_simple` helpers convert between strings and integers.

Tip

For improved readability, you can link with the C library and use `printf`/`scanf` instead of manual conversion. The choice depends on the required library dependencies.

28.6 Execution Flow and Handling Input

Console programs typically follow a loop until an exit condition is met. In a pure assembly program, the exit condition could be a specific command string (e.g., “quit”). To compare strings, you can use a simple `strcmp` loop or the `rep cmpsb` instruction.

Code: Command loop structure

```
.command_loop:
    ; display prompt
    ; read input
    ; compare with "quit"
    ; if equal, exit loop
    ; else dispatch
```

```
    jmp .command_loop
```

Reading and processing input line by line is the norm. The terminal line discipline provides editing features (backspace, etc.) automatically.

Putting It All Together

Console applications in Linux assembly are straightforward: use the `write` and `read` system calls for raw I/O, or link with libc for formatted I/O. The system-call approach yields minimal executables; the C library offers convenience. The next chapter addresses the critical topic of error handling, which makes your programs robust to inevitable external failures.

29

Error Handling in Linux Assembly

In This Chapter

No program exists in a perfect world; files may be missing, system calls may fail, and memory may run out. Linux programs must check for errors and respond appropriately. This chapter explains how to detect failure from system calls and C library functions, how to retrieve error information (the `errno` value and its textual description), and how to implement recovery strategies such as retries, fallbacks, and graceful exits. Debugging techniques for error handling with GDB are also covered.

29.1 Error Detection: System Calls and C Library

On Linux, system calls return a value that indicates success or failure. The convention is:

- A successful system call returns a non-negative value (e.g., number of bytes, file descriptor, 0 for success).
- On failure, the system call returns a **negative** value whose absolute value is the error number (`errno`). For example, `read` might return `-EINVAL` or `-ENOENT`. You must treat the return value as a signed integer.

For C library functions (e.g., `printf`, `malloc`), the convention differs: they usually return `-1` or `NULL` to indicate failure, and set the global variable `errno` (or the thread-local `errno`) to a positive error code. You can then read `errno` or use functions like `perror` to print a description.

Checking System Call Return Values

After a system call via `syscall`, `RAX` holds the return value. Compare it with a negative value to detect failure:

Code: Checking for syscall failure

```
    mov     eax, SYS_OPEN
    ; ... arguments ...
    syscall
    test    rax, rax
    js      .error              ; jump if negative (sign flag set)
    ; success, RAX = file descriptor
.error:
    neg     eax                 ; make positive (errno)
    ; eax now contains errno value
```

Alternatively, you can compare with `-1` if you only want to catch errors that return -1 (but most syscalls return negative errno, not necessarily -1). The reliable method is to test the sign flag.

Common errno Values

A few common error numbers (defined in `<errno.h>`):

- 1 — `EPERM` (Operation not permitted)
- 2 — `ENOENT` (No such file or directory)
- 3 — `ESRCH` (No such process)
- 5 — `EIO` (I/O error)
- 9 — `EBADF` (Bad file descriptor)
- 11 — `EAGAIN` (Resource temporarily unavailable)
- 13 — `EACCES` (Permission denied)
- 14 — `EFAULT` (Bad address)
- 22 — `EINVAL` (Invalid argument)

You can define these constants in your NASM file with `equ`:

```
ENOENT  equ 2
EACCES  equ 13
```

29.2 Retrieving Error Messages with C Library

If your program is linked with the C library, you can call `perror` or `strerror` to obtain a human-readable error message.

perror

`perror(const char *s)` prints the string `s` followed by a colon and a description of the current `errno` to standard error.

```
extern perror
; after a failed libc function:
call    perror      ; with RDI = pointer to prefix string
```

strerror

`strerror(int errnum)` returns a pointer to a static string describing the error. You can then write it with `write` or `printf`.

Code: Using strerror to display error

```
extern strerror
extern write, exit

section .data
prefix db 'Error: ',0

; after failure where errno is in r12d:
    mov     edi, r12d       ; errno value
    call    strerror
    mov     rsi, rax        ; error string pointer
    ; write prefix, then string, then newline...
```

29.3 Pure Assembly Error Handling (No libc)

If you are writing a minimal executable without the C library, you cannot use `strerror`. Instead, you can map error codes to static strings yourself, or simply exit with the errno number as the exit code for diagnosis. For debugging, you may choose to write the errno value in decimal using the `itoa` routine shown earlier.

Code: Pure assembly error handling for write syscall

```
; Example: write syscall that may fail (e.g., invalid fd)
    mov     eax, SYS_WRITE
    mov     edi, 999          ; likely bad file descriptor
    lea     rsi, [rel buffer]
    mov     edx, 10
    syscall
    test    rax, rax
    jns     .ok
    neg     eax
    mov     edi, eax
    jmp     .exit_error
.ok:
    ; success
    xor     edi, edi
.exit_error:
    mov     eax, SYS_EXIT
    syscall
```

Here, the program exits with the errno value as the exit code, which can be inspected with `echo $?` in the shell.

29.4 Recovery Strategies

When an error occurs, you can choose from several strategies depending on the criticality:

Exit with Error

For fatal errors, you can write a message and call `exit` (syscall 60) with a non-zero code. For system-call-only programs, write a short string to stdout/stderr and then exit.

Retry the Operation

For transient errors (e.g., `EAGAIN` for a non-blocking socket, or `EINTR` for an interrupted system call), you may retry the system call. A simple loop:

```
.retry:
    syscall
    test    rax, rax
    jns     .success
    cmp     eax, -EINTR
    je      .retry
    cmp     eax, -EAGAIN
    je      .retry
    jmp     .error
```

Fallback to a Default

If a configuration file is missing (`ENOENT`), the program may use built-in defaults instead of aborting.

Log the Error and Continue

For non-fatal errors, you can write the error to a log file or to stderr and ignore it. Use `write` to descriptor 2 (stderr) to output an error message.

Code: Writing to stderr

```
mov     eax, SYS_WRITE
mov     edi, 2            ; stderr
lea     rsi, [rel error_text]
mov     edx, errtxt_len
syscall
```

29.5 Debugging Error Handling with GDB

GDB allows you to inspect the return value of a system call or function after it executes. After a `syscall` instruction, RAX holds the result. You can examine it with:

```
p/x $rax
```

If the value is negative, it indicates an error. You can use GDB's arithmetic to convert it: `p/d -$rax` yields the corresponding positive `errno` value.

For C library calls, GDB automatically updates the `errno` variable (if you have debug symbols). You can print it with `p errno` (if the program is linked with debug info). Otherwise, you can check the return value.

Using Breakpoints on Failure

Set a breakpoint after a syscall and conditionally break on error:

```
break *0x400100
condition 1 $rax < 0
```

This stops execution only when the syscall returned an error, allowing you to inspect the context.

Using the `strace` Utility

Outside the debugger, the `strace` command is invaluable for diagnosing system call failures. Run your program with `strace ./program` to see every system call, its arguments, and the return value. For example, `strace ./myprog` would show lines like:

```
open("somefile", O_RDONLY) = -1 ENOENT (No such file or directory)
```

This immediately reveals the failure point and error code without modifying the program.

Putting It All Together

Robust error handling separates production programs from fragile prototypes. By systematically checking system call return values, using `errno` and `strerror` for descriptive messages (when linked with libc), and choosing appropriate recovery strategies, your assembly programs will gracefully handle the unexpected. The debugger and `strace` are your allies in verifying that error paths work correctly.

Important

Always check the return value of every system call that can fail. Ignoring errors can lead to silent data loss, security vulnerabilities, or crashes far from the source.

Part VI

MEMORY AND DATA HANDLING

30

Strings in Assembly

In This Chapter

Strings are the primary medium for communication between a program and its user, and for many data formats. In assembly language, a string is simply a contiguous sequence of bytes or words in memory. The processor provides powerful string-oriented instructions that can operate on entire blocks of data with a single `REP` prefix. This chapter explains how strings are represented in memory, the ubiquitous null-terminated ASCII convention (and the UTF-8 evolution), techniques for traversing and manipulating strings, a set of fundamental string operations illustrated with both manual loops and the dedicated string instructions, and an introduction to wide characters and Unicode on Linux. Every concept is demonstrated with complete NASM programs that run on Fedora 43, using both direct system calls and the C library.

30.1 String Representation

At the hardware level, there is no "string" type. A string is an array of bytes (or words) stored consecutively in memory. An assembly programmer chooses a convention for where the string ends. The two most common conventions are:

- **Length-prefixed:** the first one or two bytes store the number of characters that follow.
- **Null-terminated:** a special sentinel value (typically 0x00 for ASCII/UTF-8, 0x0000 for wide) marks the end.

The standard C library and the Linux system-call interface (e.g., `write`) use null-terminated strings, so this chapter focuses on that convention. However, length-prefixed strings are also used in certain protocols and in the C++ standard library.

A string may be stored in a `.data` section (initialized), in a `.bss` buffer, or allocated on the heap.

In all cases the memory contains numeric values; the interpretation as characters is defined by a character encoding. On modern Linux systems, the default encoding is UTF-8, which is a superset of ASCII. For the common ASCII subset (0x00–0x7F), every value maps directly to a readable glyph. Control characters like newline (LF = 0x0A) are embedded directly.

Code: A string in the .data section

```
section .data
hello   db  'H','e','l','l','o', 0   ; explicit
hello2  db  'Hello', 0               ; NASM character string
greeting db 'Hello, Fedora!', 10, 0
```

30.2 Null-Terminated Strings

The null-terminated string convention (also known as C strings) uses a byte with value zero to mark the end. Functions that operate on such strings rely on the presence of the terminator. Failure to include it can cause buffer overruns, because the function will keep reading memory until it encounters a zero byte.

When you define a string in NASM with `db 'text',0`, the assembler places the characters followed by a zero byte. The length is not stored anywhere; it must be computed by scanning for zero, or kept separately by the programmer.

In Linux, the majority of C library functions (e.g., `printf`, `strlen`, `strcpy`) expect null-terminated strings. UTF-8 is the native encoding in most modern programs, which means that a multi-byte character may occupy up to four bytes. The classic ASCII characters (U+0000–U+007F) are represented as a single byte, and the null terminator is still a single 0x00 byte, so many fundamental string operations remain the same. For simple programs, we can ignore multi-byte characters and work with ASCII.

Code: Declaring strings

```
section .data
ansi_string   db 'ANSI text', 0   ; actually UTF-8 on modern Linux
wide_string:  ; UTF-32 wide strings (wchar_t), 4 bytes per character
    dd 'H', 'e', 'l', 'l', 'o', 0
```

NASM does not have a built-in macro for wide strings like `__utf16__`, but on Linux wide characters are typically 32-bit (wchar_t). You can define them with `dd` (double word) for each character. The C library functions like `wcslen` expect null-terminated arrays of `wchar_t` (4-byte zeros).

30.3 String Traversal

Traversing a string means visiting each character in sequence until the terminator is found. The classic approach uses an index register and a loop.

Code: String traversal: computing length

```
; RDI = pointer to null-terminated string
; Returns length (excluding null) in RAX
strlen:
    xor     eax, eax
.loop:
    cmp     byte [rdi + rax], 0
    je      .done
    inc     rax
    jmp     .loop
.done:
    ret
```

The function increments `RAX` until the byte at `[rdi+rax]` is zero. It does not modify the original pointer.

An alternative uses a pointer and advances it directly:

Code: Alternative strlen using pointer increment

```
strlen2:
    mov     rax, rdi
.loop:
    cmp     byte [rax], 0
    je      .done
    inc     rax
    jmp     .loop
.done:
    sub     rax, rdi          ; length = current - start
    ret
```

Both forms are equivalent. The x86 string instruction `SCASB` (Scan String Byte) can also be used:

Code: String length with REPNE SCASB

```
strlen3:
    mov     rdi, rdi         ; set up for SCASB
    xor     al, al            ; value to search for (zero)
    mov     rcx, -1           ; maximum count
    cld                       ; clear direction flag (forward)
    repne scasb               ; scan string for AL
    neg     rcx
    sub     rcx, 2            ; compensate for over-scan
    mov     rax, rcx
    ret
```

The `REPNE SCASB` instruction uses `RDI` as the pointer (it increments after each comparison) and decrements `RCX` until the byte in `AL` is found or `RCX`=0. The final calculation yields the string length. This method is compact but may not be faster than a simple loop, because `REPNE SCASB` is microcoded. Use whichever is clearer.

30.4 String Operations

The most frequently needed string operations are copying, concatenating, comparing, and searching. The x86-64 architecture provides both explicit instruction sequences and the `REP MOVS`, `REP STOS`, `REP CMPS`, and `REP SCAS` families.

Direction Flag (`DF`)

All string instructions use the direction flag to determine whether the pointer registers (`RSI`, `RDI`) are incremented (`DF`=0) or decremented (`DF`=1). Before using any string instruction, you must ensure `DF` is cleared with `CLD`. The System V AMD64 ABI requires `DF`=0 on entry to every function, so after you have not modified it, it is safe.

Warning

If you set `DF` (with `STD`) for a backward copy, you must restore it to zero (`CLD`) before calling any C library function, because the ABI mandates `DF`=0 at call boundaries. A forgotten `STD` can cause mysterious crashes.

Copying Strings (`strcpy`)

A simple string copy that terminates with the null:

Code: strcpy implementation

```
; RDI = destination buffer, RSI = source string
; Returns pointer to destination in RAX
strcpy:
    mov     rax, rdi
.loop:
    mov     al, [rsi]
    mov     [rdi], al
    inc     rdi
    inc     rsi
    test    al, al
    jnz     .loop
    ret
```

The dedicated string instruction `MOVSB` does the same with `REP`:

Code: strcpy using REP MOVSB

```
; Using REP MOVSB: RDI = dest, RSI = src, RCX = byte count
; We need to compute the length first, then copy including null.
strcpy_rep:
    push    rdi                 ; save dest
    mov     rdi, rsi            ; src (for strlen)
    call    strlen              ; result in RAX (assumes strlen defined)
    mov     rcx, rax
    inc     rcx                 ; include null terminator
    pop     rdi                 ; dest
```

```
    mov     rsi, rsi            ; already in RSI (after pop we had dest in RDI, but we need
    ↪  src in RSI)
    ; careful: we need to put src back. We'll store src before calling strlen.
    ; Better: push rsi; push rdi; ...
    ; Simplified version:
    ; Copy src pointer to RSI, dest to RDI, count to RCX, then REP MOVSB.
    ret
```

A cleaner version using the stack:

Code: strcpy with REP MOVSB (complete)

```
strcpy_rep:
    push    rbp
    mov     rbp, rsp
    push    rdi             ; save dest
    mov     rdi, rsi        ; src for strlen
    call    strlen           ; RAX = length of src
    mov     rcx, rax
    inc     rcx              ; include null
    pop     rdi              ; dest
    cld
    rep     movsb
    pop     rbp
    ret
```

Comparing Strings (`strcmp`)

A comparison that returns difference between first mismatching characters:

Code: strcmp implementation

```
; RDI = string1, RSI = string2
; Returns EAX = 0 if equal, <0 if str1 < str2, >0 if str1 > str2
strcmp:
.loop:
    movzx   eax, byte [rdi]
    movzx   r8d, byte [rsi]
    cmp     eax, r8d
    jne     .diff
    test    eax, eax
    jz      .equal          ; both zero
    inc     rdi
    inc     rsi
    jmp     .loop
.diff:
    sub     eax, r8d
    ret
.equal:
    xor     eax, eax
    ret
```

Using `REPE CMPSB` (requires length known or both same length; not suitable for null-terminated

differencing).

Setting Memory (`memset` / `strset`)

To fill a region with a constant byte, use `STOSB`:

Code: memset using REP STOSB

```
; RDI = destination, RSI = count, DL = value
memset:
    mov     rcx, rsi
    mov     al, dl
    cld
    rep     stosb
    ret
```

Tokenizing and Parsing

Breaking a string into words is done by scanning for delimiters (space, comma, etc.) and replacing them with nulls, storing pointers. A full tokenizer is beyond this chapter, but the techniques of traversal and comparison directly apply.

Complete Example: Using String Operations

The following program copies a string, converts it to uppercase, and prints both original and modified using the `write` system call. All string manipulations are done with the routines described.

Code: listing30-1.asm — String copy and uppercase (syscall version)

```
; listing30-1.asm
; Assemble: nasm -f elf64 -o listing30-1.o listing30-1.asm
; Link:     ld -o listing30-1 listing30-1.o

bits 64
default rel

SYS_WRITE equ 1
SYS_EXIT  equ 60
STDOUT    equ 1

section .data
original db 'Hello, Assembly World!', 10, 0
orig_len equ $ - original - 1   ; exclude null
newline  db 10

section .bss
copy    resb 128

section .text
global _start

; strlen function (RDI = string, returns RAX)
strlen:
```

```
    xor     eax, eax
.loop:
    cmp     byte [rdi + rax], 0
    je      .done
    inc     rax
    jmp     .loop
.done:
    ret

; strcpy function (RDI = dest, RSI = src)
strcpy:
    mov     rax, rdi
.loop:
    mov     cl, [rsi]
    mov     [rdi], cl
    inc     rdi
    inc     rsi
    test    cl, cl
    jnz     .loop
    ret

_start:
    ; print original
    mov     eax, SYS_WRITE
    mov     edi, STDOUT
    lea     rsi, [rel original]
    mov     edx, orig_len
    syscall

    ; copy original to buffer
    lea     rdi, [rel copy]
    lea     rsi, [rel original]
    call    strcpy

    ; convert copy to uppercase
    lea     rdi, [rel copy]
.convert:
    mov     al, [rdi]
    test    al, al
    jz      .print_upper
    cmp     al, 'a'
    jb      .skip
    cmp     al, 'z'
    ja      .skip
    sub     al, 32
    mov     [rdi], al
.skip:
    inc     rdi
    jmp     .convert

.print_upper:
    lea     rdi, [rel copy]
    call    strlen
    mov     edx, eax
```

```
    mov     eax, SYS_WRITE
    mov     edi, STDOUT
    lea     rsi, [rel copy]
    syscall

    ; newline
    mov     eax, SYS_WRITE
    mov     edi, STDOUT
    lea     rsi, [rel newline]
    mov     edx, 1
    syscall

    ; exit
    mov     eax, SYS_EXIT
    xor     edi, edi
    syscall
```

30.5 Unicode and Wide Characters on Linux

The Linux environment predominantly uses UTF-8 for text processing, including in filenames and console output. UTF-8 encodes Unicode characters as variable-length byte sequences; ASCII characters (U+0000–U+007F) are single bytes, making many basic string operations identical to ASCII. However, the length in bytes is not necessarily the number of characters (or Unicode code points).

Wide characters (the `wchar_t` type in C) are 32-bit on Linux (UTF-32). Functions like `wcslen`, `wprintf`, and `wcscpy` operate on these 32-bit units. In NASM, you can declare wide strings as arrays of double words:

Code: Declaring a wide string

```
section .data
wide_msg:
    dd 'H', 'e', 'l', 'l', 'o', 0   ; 6 wchar_t elements, 0 terminator
```

When calling C library functions that expect `wchar_t` strings, you must link with the C library and use the proper extern declarations. For instance:

Code: Using wcslen from libc

```
extern wcslen
; size_t wcslen(const wchar_t *str);
; RDI = pointer to wchar_t array
; returns number of wide characters excluding null
```

Conversion between UTF-8 and wide characters can be accomplished with functions like `mbstowcs` and `wcstombs`. For a pure assembly program, implementing full UTF-8 handling is complex; it is often preferable to rely on the C library or to restrict to ASCII for simplicity.

Note

When using `printf` with the `%ls` format specifier, you can print wide strings. However, for system-call-only programs, you are limited to byte-oriented I/O; to output UTF-8 directly, the console must be set to UTF-8 mode, which is typically the default.

Putting It All Together

Strings in assembly are simple sequences of bytes or words, but the conventions (null termination, encoding) determine how they interact with the operating system and libraries. By mastering manual string traversal and the use of the x86 string instructions, you can implement any string-processing library. The next chapter will explore dynamic memory allocation, where strings are often stored in heap-allocated buffers.

31

Manual Memory Management

In This Chapter

In assembly language, every byte of memory is your responsibility. There is no garbage collector, no automatic bounds checking, and no runtime type system to catch mistakes. This chapter explains the two fundamental memory regions available to a Linux process — the stack and the heap — and how to use them correctly. It covers the concepts of memory ownership, the safe handling of pointers, the need for memory layout awareness when interacting with the operating system, and the most frequent memory-related errors that cause crashes, leaks, and security vulnerabilities. All information is based on the System V AMD64 ABI, the Linux man-pages, and practical testing with NASM 2.16 on Fedora 43.

31.1 Stack vs Heap

Memory for a process in Linux is divided into several regions. The two most important for an assembly programmer are the stack and the heap. They differ in allocation mechanism, lifetime, speed, and typical use.

31.1.1 The Stack

The stack is a region of memory managed directly by the processor through the `RSP` register. It is a last-in-first-out structure that grows downward (toward lower addresses). Space is allocated by subtracting from `RSP` and freed by adding. The `CALL` and `RET` instructions use the stack to store return addresses automatically.

Characteristics:

- **Allocation speed:** Extremely fast — a single `sub rsp, N` instruction.

- **Lifetime:** Automatic. A variable on the stack exists only while the function that allocated it is active. When the function returns, the stack pointer moves back, and the memory is effectively reclaimed.
- **Size limit:** Fixed per thread, default 8 MiB for the main thread (adjustable with `ulimit` or `setrlimit`). Large stack allocations can overflow (stack overflow).
- **Alignment:** The ABI requires the stack to be 16-byte aligned at `CALL` boundaries. Stack local variables must respect natural alignment; SSE instructions may require 16-byte alignment.

Code: Stack allocation of a 64-byte local buffer

```
; Function prologue that allocates a 64-byte local buffer
my_func:
    push    rbp
    mov     rbp, rsp
    sub     rsp, 64         ; allocate 64 bytes for locals
    ; use the buffer at [rsp] to [rsp+63]
    ; ...
    ; cleanup
    mov     rsp, rbp
    pop     rbp
    ret
```

The stack is ideal for small, temporary data whose size is known at assemble time. However, you must not return a pointer to a stack-allocated variable from a function, because that memory will be invalid once the function returns.

31.1.2 The Heap

The heap is a pool of memory managed by the C library (malloc/free) or directly by the kernel through `brk` or `mmap`. You request a block of memory and must explicitly free it. The standard C library functions `malloc`, `calloc`, `realloc`, and `free` are the usual interface.

Characteristics:

- **Allocation speed:** Slower than stack — requires a library call or system call.
- **Lifetime:** Persistent. A heap block remains valid until you explicitly free it or the process terminates.
- **Size limit:** Limited by available virtual address space (theoretically 128 TiB user-mode on x86-64 Linux).
- **Alignment:** `malloc` returns memory aligned at least to 8 bytes (usually 16 on 64-bit; required to be suitable for any purpose).

Code: Allocating and freeing a heap block with libc

```
extern malloc
extern free

section .bss
```

```
block_ptr resq 1

section .text
...
    ; allocate 1024 bytes
    mov     edi, 1024
    call    malloc
    test    rax, rax
    jz      .alloc_failed
    mov     [rel block_ptr], rax

    ; ... use the memory ...

    ; free the block
    mov     rdi, [rel block_ptr]
    call    free
    mov     qword [rel block_ptr], 0
```

If you prefer to avoid the C library entirely, you can use the `brk` system call (or `mmap`) to manage memory directly. We'll cover that in the heap chapter.

31.2 Memory Ownership

In assembly, there is no concept of ownership enforced by the language. You as the programmer must impose rules about which part of the code is responsible for freeing a block of memory. The following patterns help avoid leaks and dangling pointers:

- **The allocator frees.** The same code that allocates a block should free it when done. This is the simplest rule.
- **Passing ownership.** A function that returns a pointer to allocated memory is said to *transfer ownership* to the caller. The caller must ensure that pointer is later freed.
- **Borrowing.** A pointer passed to a function without transferring ownership is a borrow. The callee may use the memory but must not free it.
- **Reference counting.** For shared memory, manual reference counting can be implemented with a counter in a header before the data, but this is advanced and error-prone.

Document ownership clearly in comments. Assembly provides no assistance; a single extra `free` call can free a block that is still in use, causing a use-after-free vulnerability.

Warning

Never free a stack address. Passing a stack pointer to `free` will corrupt the heap and likely crash the process. Similarly, do not free the same heap block twice.

31.3 Pointer Safety

A pointer is a 64-bit integer that holds a virtual address. Dereferencing an invalid pointer causes a segmentation fault (SIGSEGV). The following guidelines reduce pointer errors:

31.3.1 Zero (NULL) Pointers

Always test for zero before dereferencing a pointer that might be NULL. Library functions commonly return NULL on failure. The zero page is never mapped, so dereferencing NULL will always fault, but a clean check allows graceful error handling.

Code: NULL pointer check

```
call    SomeAllocFunction
test    rax, rax
jz      .allocation_failed
mov     rcx, [rax]      ; safe dereference
```

31.3.2 Stale Pointers (Use-After-Free)

After freeing a heap block, overwrite the pointer variable with NULL to prevent accidental reuse. Accessing freed memory can cause silent data corruption if the memory has been reallocated to another part of the program.

Code: Zeroing a pointer after free

```
mov     rdi, [rel block_ptr]
call    free
mov     qword [rel block_ptr], 0    ; invalidate pointer
```

31.3.3 Buffer Overflows and Underflows

Assembly offers no bounds checking on memory accesses. When copying data into a buffer — especially from user input or with `REP MOVSB` — you must calculate the exact number of bytes and ensure the destination is large enough. Use a precomputed size or the size returned by an API.

Code: Safe copying with a length limit

```
; RDI = destination, RSI = source, RDX = maximum buffer size
; Copy at most RDX-1 bytes and null-terminate
safe_copy:
    push    rbp
    mov     rbp, rsp
    mov     rcx, rdx
    dec     rcx                 ; room for null
    mov     r9, rdi
.loop:
    test    rcx, rcx
    jz      .truncate
    mov     al, [rsi]
    test    al, al
    jz      .done
    mov     [rdi], al
    inc     rdi
    inc     rsi
    dec     rcx
    jmp     .loop
```

```
.truncate:
    mov     byte [rdi], 0
    leave
    ret
.done:
    mov     byte [rdi], 0
    leave
    ret
```

31.3.4 Alignment

Unaligned accesses are allowed on x86-64 for general-purpose integers but impose a performance penalty. For SSE instructions like `MOVAPS`, the operand must be 16-byte aligned. When allocating memory for SIMD data, you can use `aligned_alloc` from the C library (C11) or allocate extra bytes and manually align. On the stack, use `align` directives and subtractions that preserve alignment.

31.4 Memory Layout Awareness

Understanding the virtual address space of a Linux process helps you avoid collisions with system regions and write robust programs.

31.4.1 Process Memory Map

A simplified 64-bit user-mode layout on Linux:

- **0x0000000000000000 – 0x00007FFFFFFFFFFF** (lower 128 TiB): user-mode space.
- The program text (code) starts at `0x400000` (non-PIE) or a randomized base (PIE).
- The data and BSS segments follow.
- The heap grows upward from after the BSS (via `brk`), but `malloc` may also use `mmap` for large blocks.
- Shared libraries (libc, etc.) are mapped in the higher addresses.
- The stack starts near the top of user space (e.g., around `0x7ffffffde000`) and grows down.
- The vDSO (virtual dynamic shared object) is mapped near the top.

When you allocate memory with `mmap`, you can request a preferred address, but the OS may choose a different one. Always use the returned pointer, never hardcode addresses.

31.4.2 Stack Frame and Red Zone

The stack layout for a function call in Linux includes the return address, saved frame pointers, and local variables. There is no mandatory shadow space. The 128-byte area below `RSP` (the *red zone*) may be used by leaf functions without adjusting RSP, but must not be used if the function makes any calls (because the called function may use its own red zone and overwrite the data). When placing local variables on the stack, you must compute offsets that do not accidentally overwrite the return address or saved registers.

A safe approach: use a frame pointer (RBP) and assign negative offsets for local variables. The first local at `[rbp-8]`, the next at `[rbp-16]`, and so on.

31.4.3 Address Range Checks

You can use the `/proc/self/maps` file to view the memory map of a process. Programmatically, the `mmap` / `munmap` and the `brk` system call provide the low-level interface. For debugging, the `pmap` command or GDB's `info proc mappings` are invaluable.

31.5 Common Memory Errors

The following mistakes are among the most frequent causes of crashes and bugs in NASM programs on Linux.

1. Uninitialized Memory Reads

Reading from a buffer that has not been written, especially from the `.bss` section (which is zeroed) or from newly allocated heap memory (which is not zeroed unless you use `calloc`). Using a stale value can cause unpredictable behaviour. Always initialize memory explicitly.

2. Off-by-One Writes

Writing one byte beyond the end of a buffer corrupts adjacent memory. This is especially destructive if it overwrites a return address or a saved frame pointer on the stack.

3. Stack Misalignment for C Calls

When calling a C library function, the stack must be 16-byte aligned at the point of the `CALL`. Failure to do so may cause SSE instructions inside the library to segfault. Always realign the stack using an odd multiple of 8 subtracted from `RSP` before the call.

4. Double-Free or Invalid Free

Calling `free` with a pointer that was not allocated by `malloc` (or that has already been freed) corrupts the heap's internal structures. This often manifests as a crash during a subsequent heap operation, far from the original fault.

5. Leaking Memory

Allocating with `malloc` or `mmap` and losing the pointer without freeing it. Over time, the process's memory usage grows and may exhaust the available address space.

6. Returning a Stack Address

A function that returns a pointer to a local variable returns an address that becomes invalid the moment the function returns. The caller will read garbage or cause a crash.

7. Incorrect Size Calculations

Using `REP MOVSB` with an incorrect count can overwrite large amounts of memory. Always compute the exact number of bytes to copy, especially when using string instructions.

31.5.1 Detection and Debugging

Tools like Valgrind (memcheck) and AddressSanitizer (available with GCC) can detect heap corruption, use-after-free, and buffer overflows at runtime. Run your executable under Valgrind with `valgrind ./prog` to get detailed reports of memory errors.

Tip

Use `valgrind --leak-check=full ./your_program` to see exactly where leaks are occurring, complete with stack traces (if debug info is present).

Putting It All Together

Manual memory management gives you ultimate control, but it demands absolute discipline. By understanding the distinct roles of the stack and heap, enforcing clear ownership rules, validating all pointers before use, being aware of the process memory layout, and vigilantly avoiding the common errors listed here, you can write assembly programs that are both powerful and robust under Linux.

32

Heap Allocation in Linux

In This Chapter

Dynamic memory allocation allows a program to request blocks of memory at runtime, use them for data that has a lifespan longer than a single function, and release them when they are no longer needed. Linux provides a rich set of memory management functions through the C library (malloc, calloc, realloc, free) and direct system calls (brk, sbrk, mmap, munmap). This chapter explains the heap system concept, the core allocation functions from libc, the problems of fragmentation, strategies for efficient use, and common patterns for managing heap memory in NASM programs. Every description is drawn from the Linux man-pages, the System V AMD64 ABI, and practical testing on Fedora 43.

32.1 Heap System Concept

The heap is a pool of virtual memory that grows as needed via the `brk` system call (for small adjustments) or `mmap` (for large or independent regions). The C library's `malloc` function provides a high-level interface that manages these underlying mechanisms.

Key properties of the glibc heap manager (used in Fedora):

- **Thread-local caches.** glibc uses per-thread arenas to reduce lock contention.
- **Bins and chunks.** Freed blocks are stored in linked lists (bins) for future reuse. Small and large allocations are handled differently.
- **Alignment.** `malloc` returns memory aligned at least to 8 bytes (typically 16 on 64-bit) suitable for any data type.
- **Error handling.** Allocation failure returns `NULL`. You must always check for failure.

For a pure assembly program without libc, you can manage memory directly using the `brk` system call (to adjust the program break) or `mmap` (to map anonymous pages). However, using libc's `malloc` is simpler and more portable.

32.2 malloc, calloc, realloc, and free

The primary libc functions for heap memory are:

Code: Function prototypes for memory allocation

```
extern malloc
extern calloc
extern realloc
extern free

; void* malloc(size_t size);
; RDI = size (bytes)
; returns pointer in RAX, or NULL on failure

; void* calloc(size_t nmemb, size_t size);
; RDI = number of elements, RSI = element size
; returns pointer to zeroed memory, or NULL

; void* realloc(void *ptr, size_t size);
; RDI = old pointer, RSI = new size
; returns pointer (possibly moved), or NULL

; void free(void *ptr);
; RDI = pointer
```

Allocating with malloc

Code: Allocating 256 bytes with malloc

```
section .bss
block_ptr resq 1

section .text
    mov     edi, 256
    call    malloc
    test    rax, rax
    jz      .alloc_failed
    mov     [rel block_ptr], rax
```

Zeroed Allocation with calloc

Code: Allocating and zeroing an array of 100 integers

```
mov     edi, 100        ; nmemb
mov     esi, 4          ; sizeof(int)
call    calloc
test    rax, rax
jz      .alloc_failed
; memory is already zeroed
```

Resizing with realloc

Code: Growing a block

```
mov     rdi, [rel block_ptr]
mov     esi, 512        ; new size
call    realloc
test    rax, rax
jz      .realloc_failed
mov     [rel block_ptr], rax
```

Freeing

Always free with the exact pointer returned by the allocator, and never free the same block twice.

Complete Example: A Tiny String Buffer Using libc

The following program allocates a buffer, copies a message into it, prints it to stdout, and frees the buffer. It uses `puts` for output (which appends a newline).

Code: listing32-1.asm — Heap allocation and usage with libc

```
; listing32-1.asm
; Assemble: nasm -f elf64 -o listing32-1.o listing32-1.asm
; Link:     gcc -no-pie -o listing32-1 listing32-1.o

bits 64
default rel

extern malloc
extern free
extern puts
extern exit

section .data
msg     db 'Hello from the heap!', 0

section .bss
buf_ptr resq 1

section .text
```

```
global main
main:
    push    rbp
    mov     rbp, rsp
    sub     rsp, 16             ; align stack

    ; allocate enough space for the string (with terminator)
    lea     rdi, [rel msg]
    call    strlen              ; we need strlen; we'll embed it here or call an external
    ; just manually use a known length for simplicity (we'll define a str function)
    lea     rdi, [rel msg]
    call    strlen_custom
    inc     eax                  ; +1 for null terminator
    mov     edi, eax
    call    malloc
    test    rax, rax
    jz      .exit
    mov     [rel buf_ptr], rax

    ; copy string
    mov     rdi, rax
    lea     rsi, [rel msg]
    call    strcpy_custom

    ; print
    mov     rdi, [rel buf_ptr]
    call    puts

    ; free
    mov     rdi, [rel buf_ptr]
    call    free
    mov     qword [rel buf_ptr], 0

.exit:
    xor     edi, edi
    call    exit

; ----- simple strlen -----
strlen_custom:
    xor     eax, eax
.loop:
    cmp     byte [rdi + rax], 0
    je      .done
    inc     rax
    jmp     .loop
.done:
    ret

; ----- simple strcpy -----
strcpy_custom:
    mov     rax, rdi
.loop:
    mov     cl, [rsi]
    mov     [rdi], cl
```

```
    inc     rdi
    inc     rsi
    test    cl, cl
    jnz     .loop
    ret
```

32.3 Fragmentation

Fragmentation occurs when free memory is split into small, non-contiguous chunks, making it impossible to satisfy a large allocation request even though the total free space is sufficient. There are two types:

- **External fragmentation:** Unused memory is scattered between allocated blocks. A subsequent allocation of a size larger than any free gap will fail, or the heap must expand (or use mmap for large blocks). glibc mitigates this by using multiple size classes (bins) and coalescing adjacent free blocks.
- **Internal fragmentation:** An allocated block may be larger than the requested size because of alignment or a minimum chunk size. The wasted bytes inside the block are internal fragmentation.

32.3.1 Avoiding Fragmentation

From an assembly programming perspective, you can reduce fragmentation by:

1. **Allocating similar sizes together.** Group many small objects of the same size; glibc will place them in the same bins.
2. **Freeing in the reverse order of allocation.** Helps coalesce blocks.
3. **Avoiding many tiny, short-lived allocations.** Consider using a pool allocator for fixed-size objects.
4. **Using `mmap` for large buffers.** For allocations larger than a threshold (default 128 KiB, configurable via mallopt), glibc automatically uses `mmap`. You can also call `mmap` directly.

32.4 Allocation Strategies

The choice of allocation function depends on the size, lifetime, and alignment requirements.

32.4.1 Small, Frequent Allocations

Use `malloc` / `free`. glibc's fastbins and tcache provide excellent performance for many small allocations.

32.4.2 Large Blocks (Over 128 KiB)

For very large blocks, `mmap` directly may be more efficient, as it bypasses the heap's internal metadata and can be unmapped instantly. Example with system call:

Code: Allocating memory with mmap (syscall)

```
SYS_MMAP equ 9
SYS_MUNMAP equ 11

PROT_READ  equ 1
PROT_WRITE equ 2
MAP_PRIVATE equ 0x02
MAP_ANONYMOUS equ 0x20

section .bss
addr resq 1

section .text
    mov     eax, SYS_MMAP          ; mmap
    xor     edi, edi               ; addr = NULL
    mov     esi, 4096              ; length = one page
    mov     edx, PROT_READ | PROT_WRITE
    mov     r10d, MAP_PRIVATE | MAP_ANONYMOUS
    mov     r8d, -1                ; fd = -1
    xor     r9d, r9d               ; offset = 0
    syscall
    cmp     rax, -1                ; error check (actually a negative value indicates error)
    jae     .ok                    ; success: address >= 0
    neg     rax                    ; convert to positive errno
    mov     edi, eax
    jmp     .error
.ok:
    mov     [rel addr], rax

    ; ... use memory ...

    ; unmap
    mov     eax, SYS_MUNMAP
    mov     rdi, [rel addr]
    mov     esi, 4096
    syscall
```

Note: the system call return convention is that a negative value indicates an errno. We check if RAX is negative (sign bit set) to detect failure.

If you prefer using the C library, `mmap` and `munmap` are also available as wrapper functions (requires linking with `-lc`).

32.4.3 Alignment-Sensitive Data

To allocate memory with a specific alignment, use `aligned_alloc` (C11 standard) or `posix_memalign`:

Code: Allocating 32-byte aligned memory with aligned_alloc

```
extern aligned_alloc
; void* aligned_alloc(size_t alignment, size_t size);
; alignment must be a power of two, size a multiple of alignment.
    mov     edi, 32        ; alignment
```

```
    mov     esi, 1024       ; size
    call    aligned_alloc
    test    rax, rax
    jz      .fail
```

For a pure assembly approach, you can request extra memory and align manually.

32.4.4 Lifetime and Ownership

Adopt a consistent pattern: a function that allocates memory should document that the caller is responsible for freeing it. A function that returns a pointer to a dynamically allocated structure should store it in an output parameter or return it, and the caller frees it.

32.5 Memory Management Patterns

Several proven patterns help keep heap usage safe and understandable.

32.5.1 Initialize-Allocate-Free

The most common pattern: the program initializes the memory subsystem (libc does this automatically), allocates blocks as needed, and frees them before exit. For short-lived programs, the OS reclaims all memory automatically on process termination, so strictly freeing every allocated block is optional, but it is good practice.

32.5.2 Private Arena via mmap

For a large number of allocations with the same lifetime, you can create a private memory region using `mmap` and manage it with a simple bump allocator or free list. This avoids interacting with glibc's global heap and can be faster.

For example, allocate a 1 MiB region and maintain a pointer to the next free byte. To allocate, increment the pointer and return the old value. This is extremely fast but cannot free individual blocks; the entire region must be freed at once. This is ideal for temporary computations.

32.5.3 Fixed-Size Pool Allocator

When the maximum number of items is known in advance, pre-allocate a single buffer and divide it into slots. Use a free list (linked list of free slots). Example: allocate 100 items of size 64 bytes, initialize the free list. Each allocation pops from the free list; free pushes back.

Code: Pool allocator initialization

```
    mov     edi, 100 * 64
    call    malloc
    test    rax, rax
    jz      .fail
    mov     r8, rax         ; pool start
    ; build free list: store next pointer at the beginning of each slot
    mov     rcx, 99
```

```
    mov     r9, r8
    mov     [rel free_head], r8
.loop:
    lea     rdx, [r9 + 64]
    mov     qword [r9], rdx  ; next = r9 + 64
    mov     r9, rdx
    loop    .loop
    mov     qword [r9], 0     ; last points to NULL
```

32.5.4 Double-Buffering

When reading from or writing to a device, allocate two buffers: one being filled and one being processed. This avoids stalls and is common in file I/O.

32.5.5 Leak Detection

For longer-running programs, you can implement a simple leak detector by recording each allocation in a table. Use `valgrind` or the built-in mechanisms of glibc (`MALLOC_CHECK_`). Assembling with debug info and running under `valgrind --leak-check=full` will pin-point every leak.

Tip

To enable glibc's internal heap checking, set the environment variable `MALLOC_CHECK_=3` and run the program. Any corruption or invalid free will be reported to stderr. Combine with `gdb` for detailed analysis.

Putting It All Together

Heap allocation in Linux provides powerful, flexible mechanisms for dynamic memory management. By using `malloc` and friends from libc with proper error checking, understanding the fragmentation trade-offs, choosing the right allocation strategy for each task, and applying established patterns, you can build efficient, leak-free assembly programs on Fedora 43. The next chapter will explore direct virtual memory management with `mmap` for scenarios that require page-level control.

33

Virtual Memory Concepts

In This Chapter

Modern operating systems create an abstraction called virtual memory that gives each process the illusion of owning a vast, contiguous address space. The hardware and the Linux kernel work together to map virtual addresses to physical pages, enforce protection, and handle page faults transparently. For the assembly programmer, understanding virtual memory is essential for allocating large blocks, protecting code and data, and optimizing performance. This chapter explains the virtual address space layout, the paging mechanism, memory protection flags, the page fault lifecycle, and the detailed usage of the `mmap`, `munmap`, and `mprotect` system calls on Fedora 43. Every explanation is verified against the Intel® 64 and IA-32 Architectures Software Developer's Manual, the AMD64 Architecture Programmer's Manual, and the Linux man-pages. All examples assemble with NASM 2.16 using pure system calls; no C library is required.

33.1 Virtual Address Space

On x86-64, user-mode processes on Linux enjoy a flat virtual address space spanning the lower 128 TiB (from `0x0000000000000000` to `0x00007FFFFFFFFFFF`). The upper half is reserved for the kernel. Although addresses are 64 bits wide, current implementations use only 48 bits for address translation (the canonical form); the remaining bits must be sign-extended from bit 47. An address that violates the canonical rule causes a general-protection fault.

The kernel divides the user address space into regions, each with specific state and protection. A typical process memory layout (without address-space layout randomization, ASLR) looks like:

- **Text (code):** mapped read-only executable at a low address, often `0x400000` for non-PIE executables.
- **Data and BSS:** read-write segments following the text.

- **Heap:** grows upward via the `brk` system call or via `mmap` for larger blocks.
- **Memory-mapped regions:** shared libraries and `mmap` allocations are placed in the middle.
- **Stack:** grows downward from a high address (near the top of user space).
- **[vdso] / [vvar]:** kernel-provided data and code pages near the top.

The exact layout is visible in `/proc/<pid>/maps`. The page size on x86-64 is normally 4 KiB, though huge pages (2 MiB, 1 GiB) may also be used.

Getting the System Page Size

The following program retrieves the system page size using the `getpagesize` libc function, and then exits with that value as its exit code. (You can check the result with `echo $?`.)

Code: listing33-1.asm — Exiting with page size

```
; listing33-1.asm
; Assemble: nasm -f elf64 -o listing33-1.o listing33-1.asm
; Link:     gcc -no-pie -o listing33-1 listing33-1.o

bits 64
default rel

extern getpagesize
extern exit

section .text
global main
main:
    call    getpagesize
    mov     edi, eax        ; exit code = page size (usually 0x1000 = 4096)
    call    exit
```

Because the exit status is only 8 bits wide, you will see the low byte of the page size; the full value is returned in RAX. This small program confirms that your toolchain is set up to query virtual memory parameters.

33.2 Paging System

The processor and the kernel cooperate to implement virtual memory through *paging*. In x86-64, a multi-level page table structure translates 48-bit virtual addresses into physical addresses. Each process has its own set of page tables; the `CR3` register points to the PML4 base. The translation walk is performed by the MMU and the results are cached in the TLB.

Linux uses demand paging: pages are not backed by physical memory until accessed. A reserved region may not yet have physical storage; when touched, a page fault occurs and the kernel allocates a page (commit). This allows efficient memory usage.

Reserve vs. Commit

With `mmap`, you can reserve a virtual address range without immediately allocating physical frames by using the `MAP_NORESERVE` flag (or by not touching the pages). More commonly, you allocate anonymous memory with `MAP_ANONYMOUS` and `PROT_READ|PROT_WRITE`; the kernel commits pages on first access.

33.3 Memory Protection

Each virtual page has a protection attribute that controls read, write, and execute access. The constants are:

- `PROT_NONE` (0) — all access causes a segmentation fault.
- `PROT_READ` (1) — read only.
- `PROT_WRITE` (2) — write (note: on Intel, write permission implies read unless NX is used differently; Linux usually sets both read and write when desired).
- `PROT_EXEC` (4) — execute.

These can be combined with bitwise OR (e.g., `PROT_READ|PROT_WRITE`). The `mprotect` system call changes the protection of an existing mapping. Modern kernels also support the `PROT_EXEC` flag, and by default the NX bit prevents execution of writable pages.

Warning

Writing to a read-only page causes a `SIGSEGV`. Changing a page to executable and then modifying it is a common pattern for just-in-time compilers, but it requires careful handling to avoid security vulnerabilities.

Example: Allocating and Protecting Pages with mmap and mprotect

The following program allocates a page with read-write access, writes a value, then changes it to read-only and reads it. It uses only system calls; no libc is required.

Code: listing33-2.asm — Changing page protection

```
; listing33-2.asm
; Assemble: nasm -f elf64 -o listing33-2.o listing33-2.asm
; Link:     ld -o listing33-2 listing33-2.o

bits 64
default rel

SYS_MMAP    equ 9
SYS_MPROTECT equ 10
SYS_MUNMAP  equ 11
SYS_EXIT    equ 60

PROT_READ  equ 1
PROT_WRITE equ 2
```

```
MAP_PRIVATE equ 0x02
MAP_ANONYMOUS equ 0x20

section .bss
page_ptr resq 1

section .text
global _start
_start:
    ; mmap(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0)
    xor     edi, edi              ; addr = NULL
    mov     esi, 4096             ; length
    mov     edx, PROT_READ | PROT_WRITE
    mov     r10d, MAP_PRIVATE | MAP_ANONYMOUS
    mov     r8d, -1               ; fd = -1
    xor     r9d, r9d              ; offset = 0
    mov     eax, SYS_MMAP
    syscall
    cmp     rax, -1
    ja      .mmap_ok              ; success if return > 0 (actually mmap returns address,
    ↪  -errno on error)
    neg     rax
    mov     edi, eax
    jmp     .exit_error
.mmap_ok:
    mov     [rel page_ptr], rax

    ; write a dword to the page
    mov     dword [rax], 0x12345678

    ; mprotect(addr, 4096, PROT_READ)
    mov     rdi, rax              ; addr
    mov     esi, 4096             ; len
    mov     edx, PROT_READ        ; new protection
    mov     eax, SYS_MPROTECT
    syscall
    test    eax, eax
    jnz     .exit_error

    ; read from the page (succeeds)
    mov     rax, [rel page_ptr]
    mov     eax, [rax]            ; just read; discard

    ; Attempt to write would cause SIGSEGV (commented out):
    ; mov     dword [rax], 0xDEAD  ; uncomment to crash

    ; munmap(page_ptr, 4096)
    mov     rdi, [rel page_ptr]
    mov     esi, 4096
    mov     eax, SYS_MUNMAP
    syscall

    xor     edi, edi
    call    _exit
```

```
_exit:
    mov     eax, SYS_EXIT
    syscall

.exit_error:
    mov     edi, 1              ; exit with error code 1
    jmp     _exit
```

The program exits normally. If the commented write line were activated, it would crash with a `SIGSEGV`.

33.4 Page Faults

A *page fault* is an exception raised by the processor when a virtual address cannot be translated or when a protection check fails. On Linux, page faults are classified as:

- **Minor (soft) page fault:** The page is already in memory (e.g., in the page cache) but the page table entry was not marked as valid. The kernel updates the PTE and resumes execution. This is cheap.
- **Major (hard) page fault:** The page must be read from disk (e.g., from a memory-mapped file or swapped-out anonymous page). This involves I/O and is expensive.
- **Invalid page fault:** Accessing a page with `PROT_NONE` or an unmapped address, or permission violation (e.g., writing to a read-only page). This results in a `SIGSEGV` signal, which by default terminates the process.

A user-mode program can install a signal handler to catch `SIGSEGV` and recover, but this is advanced and rarely used for normal data access. Guard pages are a deliberate use of protection faults to detect stack overflows or to implement growable structures.

Guard Page Demonstration

The following program creates two consecutive pages, marks the first as `PROT_NONE` (a guard page), and attempts to write to it. Without a signal handler, the program will crash with a segmentation fault.

Code: listing33-3.asm — Creating a guard page

```
; listing33-3.asm -- creates a guard page, then accesses it (crashes)
; Assemble: nasm -f elf64 -o listing33-3.o listing33-3.asm
; Link:     ld -o listing33-3 listing33-3.o

bits 64
default rel

SYS_MMAP    equ 9
SYS_MPROTECT equ 10
SYS_EXIT    equ 60
```

```
PROT_READ   equ 1
PROT_WRITE  equ 2
PROT_NONE   equ 0
MAP_PRIVATE equ 0x02
MAP_ANONYMOUS equ 0x20

section .bss
guard_ptr resq 1

section .text
global _start
_start:
    ; allocate 2 pages (8192 bytes) with read-write protection
    xor     edi, edi
    mov     esi, 8192
    mov     edx, PROT_READ | PROT_WRITE
    mov     r10d, MAP_PRIVATE | MAP_ANONYMOUS
    mov     r8d, -1
    xor     r9d, r9d
    mov     eax, SYS_MMAP
    syscall
    cmp     rax, -1
    ja      .ok
    neg     rax
    mov     edi, eax
    jmp     .exit
.ok:
    mov     [rel guard_ptr], rax

    ; set the first page to PROT_NONE (guard page)
    mov     rdi, rax
    mov     esi, 4096
    mov     edx, PROT_NONE
    mov     eax, SYS_MPROTECT
    syscall
    test    eax, eax
    jnz     .exit

    ; attempt to write to the guard page → SIGSEGV
    mov     rax, [rel guard_ptr]
    mov     byte [rax], 0        ; <-- crash here
    ; unreachable

    mov     eax, SYS_EXIT
    xor     edi, edi
    syscall
.exit:
    mov     edi, 1
    mov     eax, SYS_EXIT
    syscall
```

Running this program will cause a segmentation fault. In a real application, you would install a signal handler to catch the fault, release the guard, and continue.

33.5 mmap, munmap, and mprotect Usage

The `mmap` system call is the primary interface for memory allocation at page granularity. Its prototype (wrapped in C) is:

```
; void *mmap(void *addr, size_t length, int prot, int flags, int fd, off_t offset);
; RDI = addr (usually NULL)
; RSI = length
; RDX = prot (PROT_READ | PROT_WRITE ...)
; R10 = flags (MAP_PRIVATE | MAP_ANONYMOUS ...)
; R8  = fd ( -1 for anonymous)
; R9  = offset (0)
; Return: address in RAX, or a negative errno on failure.
```

Common flags:

- `MAP_PRIVATE` – modifications are private to the process.
- `MAP_ANONYMOUS` – the mapping is not backed by any file; the kernel provides zero-filled pages.
- `MAP_FIXED` – (advanced) interpret address as a requirement, not a hint.

`munmap` removes a mapping:

```
; int munmap(void *addr, size_t length);
```

`mprotect` changes protection:

```
; int mprotect(void *addr, size_t len, int prot);
```

Allocating a Large Virtual Region and Committing On Demand

The following program reserves 1 MiB of virtual address space (without committing all pages immediately), commits the first page by writing to it, and later frees the entire region. It demonstrates the two-phase pattern.

Code: listing33-4.asm — Reserve and commit with mmap

```
; listing33-4.asm
; Assemble: nasm -f elf64 -o listing33-4.o listing33-4.asm
; Link:     ld -o listing33-4 listing33-4.o

bits 64
default rel

SYS_MMAP   equ 9
SYS_MUNMAP equ 11
SYS_EXIT   equ 60

PROT_READ  equ 1
PROT_WRITE equ 2
MAP_PRIVATE equ 0x02
MAP_ANONYMOUS equ 0x20

section .bss
region resq 1
```

```
section .text
global _start
_start:
    ; Reserve 1 MiB (no physical pages committed yet)
    xor     edi, edi
    mov     esi, 1 * 1024 * 1024
    mov     edx, PROT_READ | PROT_WRITE
    mov     r10d, MAP_PRIVATE | MAP_ANONYMOUS
    mov     r8d, -1
    xor     r9d, r9d
    mov     eax, SYS_MMAP
    syscall
    cmp     rax, -1
    ja      .ok
    mov     edi, 1
    jmp     .exit
.ok:
    mov     [rel region], rax

    ; Write to the first page (commits it)
    mov     dword [rax], 0xAABBCCDD

    ; The rest of the region remains reserved but not committed until touched.

    ; Read back the written value
    mov     rax, [rel region]
    mov     eax, [rax]              ; eax = 0xAABBCCDD

    ; Free the entire region
    mov     rdi, [rel region]
    mov     esi, 1 * 1024 * 1024
    mov     eax, SYS_MUNMAP
    syscall

    xor     edi, edi
.exit:
    mov     eax, SYS_EXIT
    syscall
```

Tip

For most dynamic allocations, the C library's `malloc` is easier to use because it manages sub-allocations within larger `mmap` or `brk` regions. Use `mmap` directly when you need page-aligned memory, specific protection, or very large contiguous blocks.

Putting It All Together

Virtual memory is the fundamental abstraction that allows every assembly program to run in a flat, protected address space. By understanding the virtual address layout, the paging system, the `PROT_*` protection flags, the nature of page faults, and the precise usage of `mmap`, `munmap`, and `mprotect`, you gain the ability to allocate, protect, and manage memory at the finest granularity.

This knowledge is essential for system-level programming, performance optimization, and robust memory management in NASM applications on Fedora 43.

34

Buffer Operations and Safety

In This Chapter

Buffers—contiguous blocks of memory used to store strings, arrays, or raw data—are everywhere in systems programming. In assembly language, the programmer has absolute control over how buffers are accessed and manipulated, which brings great power and the corresponding responsibility to prevent buffer overflows, memory corruption, and security vulnerabilities. This chapter explains the risks inherent in buffer operations, presents safe copy techniques using custom routines (and where useful, standard C library functions), demonstrates bounds checking at the machine level, covers memory sanitization to avoid information leaks, and advocates a defensive programming mindset that catches errors as early as possible. Every example is written in pure NASM, tested on Fedora 43, and uses only Linux system calls when I/O is needed.

34.1 Buffer Overflow Risks

A buffer overflow occurs when a write operation exceeds the allocated size of a buffer, overwriting adjacent memory. In assembly, there is no automatic bounds checking. An errant `mov` or `REP STOSB` can smash the stack, the return address, or critical data structures.

Common causes:

- Using a string instruction with an incorrect count in `RCX`.
- Calling a C library function such as `strcpy` without verifying the destination buffer capacity.
- Manually writing past the end of a local array on the stack.
- Processing user input without limiting the number of bytes read.

Consequences range from silent data corruption to immediate segmentation faults. An overflow

that overwrites the return address on the stack can redirect execution to arbitrary code (classic stack-smashing attack). Even a single-byte overflow can corrupt a saved frame pointer, causing a crash long after the offending function has returned.

Code: Vulnerable copy loop (no bounds)

```
; RDI = destination buffer (only 16 bytes allocated)
; RSI = source string (maybe much longer)
vulnerable_copy:
    cld
.loop:
    lodsb               ; load byte from [rsi] into AL, increment RSI
    stosb               ; store AL into [rdi], increment RDI
    test    al, al
    jnz     .loop
    ret
```

A source string longer than 15 bytes will overwrite data beyond the 16-byte buffer.

Warning

Never trust a pointer alone. In assembly, every pointer must be accompanied by an explicit length or limit.

34.2 Safe Copy Techniques

To prevent overflows, copy operations must be constrained to the destination buffer size.

Manual Copy with Counter

The simplest safe technique is to pass the destination buffer size and copy at most `dest_size - 1` bytes, then null-terminate.

Code: Safe copy with explicit maximum length

```
; RDI = dest, RSI = src, RDX = dest_size
; Copies at most RDX-1 bytes and null-terminates.
safe_copy:
    push    rbx
    mov     rbx, rdi          ; save dest base for return
    dec     rdx               ; leave room for null
    jz      .null_only
    mov     rcx, rdx
.loop:
    mov     al, [rsi]
    test    al, al
    jz      .done
    mov     [rdi], al
    inc     rdi
    inc     rsi
    loop    .loop
    mov     byte [rdi], 0   ; truncation
```

```
    pop     rbx
    ret
.done:
    mov     byte [rdi], 0
    pop     rbx
    ret
.null_only:
    mov     byte [rdi], 0
    pop     rbx
    ret
```

Using REP MOVSB with Bounds

The string instruction `REP MOVSB` can be used safely if the count is computed correctly, using the minimum of source length and buffer size.

Code: Bounded copy with REP MOVSB

```
; RDI = dest, RSI = src, RDX = dest_size
safe_strcpy_rep:
    push    rbx
    push    r12
    mov     rbx, rdi
    ; find length of src
    mov     rdi, rsi
    call    strlen          ; returns length in rax
    mov     r12, rax
    ; determine number to copy: min(r12, dest_size-1)
    cmp     r12, rdx
    jae     .truncate
    mov     rcx, r12        ; fit entirely
    jmp     .do_copy
.truncate:
    mov     rcx, rdx
    dec     rcx             ; leave room for null
    cmp     rcx, r12
    cmova   rcx, r12        ; (already handled)
.do_copy:
    mov     rdi, rbx
    cld
    rep     movsb
    mov     byte [rdi], 0
    pop     r12
    pop     rbx
    ret
```

(The `strlen` function is assumed to be defined as earlier in the book.)

Using Standard C Library Safe Functions

If you link with the C library, functions like `strncpy` (from `libc`) provide bounded string copy. The prototype is:

```
extern strncpy
```

```
; char *strncpy(char *dest, const char *src, size_t n);
; RDI = dest, RSI = src, RDX = n
```

It copies at most `n` characters from `src` to `dest`, but note that it does *not* null-terminate if the source is longer than `n`. You must manually place a null terminator. `snprintf` is a safer alternative for building strings.

34.3 Bounds Checking

Bounds checking means verifying that an index or pointer is within the valid range of a buffer before accessing it.

Index Bounds Check

For a buffer of known element count `N`, ensure that an index `i` satisfies `0 <= i < N`.

Code: Index check

```
; RDI = base of buffer (not needed), RSI = index, RDX = max index (exclusive)
; returns: ZF set if index in range (RSI < RDX), else ZF clear.
bounds_check:
    cmp     rsi, rdx
    jae     .out
    xor     eax, eax
    ret
.out:
    mov     eax, 1
    ret
```

Usage:

Code: Using the bounds check

```
call    bounds_check
test    eax, eax
jnz     .error
mov     rax, [rdi + rsi*8]   ; safe access
```

Pointer-Based Bounds Check

When using a moving pointer, compare against an end pointer.

Code: Loop with pointer bounds

```
; RDI = current pointer, RSI = end pointer
.loop:
    cmp     rdi, rsi
    jae     .out
    inc     byte [rdi]
    inc     rdi
    jmp     .loop
```

API-Assisted Bounds

The `read` system call returns the number of bytes actually read. Always use that count for subsequent processing, never assume the buffer is full.

Code: Reading with exact count

```
; read(fd, buf, 1024)
mov     eax, 0                  ; SYS_read
mov     edi, fd
lea     rsi, [rel buffer]
mov     edx, 1024
syscall
; rax = bytes actually read, may be less than 1024
; use rax as limit
mov     r12, rax
```

34.4 Memory Sanitization

Sanitization means ensuring that memory released or passed across trust boundaries does not contain sensitive data.

Zeroing Before Free

Before releasing a buffer (via `munmap` or the heap), overwrite it with zeros. The `secure_zero` routine uses `REP STOSB` and cannot be optimized away.

Code: Secure zeroing

```
; RDI = pointer, RSI = size in bytes
secure_zero:
    mov     rcx, rsi
    xor     eax, eax
    cld
    rep     stosb
    ret
```

Usage example after allocating with `mmap`:

Code: Sanitizing before munmap

```
; sanitize
mov     rdi, [rel buf_ptr]
mov     esi, [rel buf_size]
call    secure_zero

; free
mov     rdi, [rel buf_ptr]
mov     esi, [rel buf_size]
mov     eax, SYS_MUNMAP
syscall
```

Zeroing Stack Data

Local buffers on the stack can be cleared before the function returns, particularly if they held passwords or cryptographic keys.

Code: Clearing a local buffer

```
my_func:
    push    rbp
    mov     rbp, rsp
    sub     rsp, 256        ; 256-byte buffer
    ; ... use buffer ...
    ; before returning, zero it
    lea     rdi, [rsp]
    mov     rcx, 256
    xor     eax, eax
    cld
    rep     stosb
    leave
    ret
```

34.5 Defensive Programming

Defensive programming means anticipating errors and invalid input rather than assuming ideal conditions. In assembly this discipline is entirely manual.

1. Validate All Inputs

Check pointers for NULL and lengths for validity at the start of every function that accepts external data.

2. Check System Call Return Values

Every system call returns an error indication (a negative `errno` in RAX on failure). Always test for error after `syscall`. For example:

```
syscall
test    rax, rax        ; check sign?
js      .error          ; negative indicates error
```

3. Prefer Bounded Copy

Use the safe copy routines presented earlier, or C library functions like `snprintf`, `strncpy` (with manual null termination), or `memcpy` with a verified size.

4. Limit Recursion and Stack Usage

In assembly, recursion can quickly exhaust the stack. Use iteration or set a hard limit.

5. Zero Freed Pointers

After freeing memory, overwrite the pointer variable with zero to prevent use-after-free.

6. Log Suspicious Events

For debugging, you can write diagnostic messages to stderr (file descriptor 2) using the `write` system call.

Code: Logging to stderr

```
section .data
err_msg db 'Error: invalid operation', 10
err_len equ $ - err_msg

; ...
    mov     eax, 1              ; SYS_write
    mov     edi, 2              ; stderr
    lea     rsi, [rel err_msg]
    mov     edx, err_len
    syscall
```

7. Use NASM Warnings

Assemble with `nasm -W+all` or `-Wall` and treat warnings as errors.

Complete Example: Defensive File Reader

The following program opens a file, reads its contents into a dynamically allocated buffer with full bounds checking, sanitizes the buffer before deallocation, and logs errors to stderr. All file I/O uses Linux system calls; no C library is required.

Code: listing34-1.asm — Safe file read with defensive patterns

```
; listing34-1.asm
; Assemble: nasm -f elf64 -o listing34-1.o listing34-1.asm
; Link:     ld -o listing34-1 listing34-1.o

bits 64
default rel

SYS_OPEN   equ 2
SYS_READ   equ 0
SYS_CLOSE  equ 3
SYS_MMAP   equ 9
SYS_MUNMAP equ 11
SYS_WRITE  equ 1
SYS_EXIT   equ 60

O_RDONLY   equ 0

PROT_READ  equ 1
PROT_WRITE equ 2
MAP_PRIVATE equ 0x02
MAP_ANONYMOUS equ 0x20

section .data
```

```
filename   db 'test.txt', 0
err_open   db 'Error: cannot open file', 10
err_open_len equ $ - err_open
err_alloc  db 'Error: allocation failed', 10
err_alloc_len equ $ - err_alloc
err_read   db 'Error: read failed', 10
err_read_len equ $ - err_read

section .bss
fd         resq 1
buf_ptr    resq 1               ; will point to mmap'd buffer
bytes_read resq 1

section .text
global _start
_start:
    ; open file
    lea     rdi, [rel filename]
    mov     esi, O_RDONLY
    xor     edx, edx             ; mode (ignored)
    mov     eax, SYS_OPEN
    syscall
    test    rax, rax
    js      .open_error
    mov     [rel fd], rax

    ; allocate a 4096-byte buffer via mmap
    xor     edi, edi
    mov     esi, 4096
    mov     edx, PROT_READ | PROT_WRITE
    mov     r10d, MAP_PRIVATE | MAP_ANONYMOUS
    mov     r8d, -1
    xor     r9d, r9d
    mov     eax, SYS_MMAP
    syscall
    cmp     rax, -1
    ja      .alloc_ok
    neg     rax
    mov     rdi, rax
    jmp     .alloc_error
.alloc_ok:
    mov     [rel buf_ptr], rax

    ; read from file into buffer (max 4095 to leave room for null)
    mov     rdi, [rel fd]
    mov     rsi, rax
    mov     edx, 4095
    mov     eax, SYS_READ
    syscall
    test    rax, rax
    js      .read_error
    mov     [rel bytes_read], rax

    ; null-terminate
```

```
    mov     rdi, [rel buf_ptr]
    add     rdi, rax
    mov     byte [rdi], 0

    ; (in a real program, process the data here)
    ; For demonstration, just write the data to stdout
    mov     eax, SYS_WRITE
    mov     edi, 1                  ; stdout
    mov     rsi, [rel buf_ptr]
    mov     rdx, [rel bytes_read]
    syscall

    ; sanitize buffer
    mov     rdi, [rel buf_ptr]
    mov     esi, 4096
    call    secure_zero

    ; free buffer
    mov     rdi, [rel buf_ptr]
    mov     esi, 4096
    mov     eax, SYS_MUNMAP
    syscall

    ; close file
    mov     rdi, [rel fd]
    mov     eax, SYS_CLOSE
    syscall

    xor     edi, edi
    jmp     .exit

.open_error:
    lea     rsi, [rel err_open]
    mov     edx, err_open_len
    call    log_error
    mov     edi, 1
    jmp     .exit
.alloc_error:
    lea     rsi, [rel err_alloc]
    mov     edx, err_alloc_len
    call    log_error
    mov     edi, 1
    jmp     .exit
.read_error:
    lea     rsi, [rel err_read]
    mov     edx, err_read_len
    call    log_error
    ; still free buffer and close file before exit
    mov     rdi, [rel buf_ptr]
    mov     esi, 4096
    mov     eax, SYS_MUNMAP
    syscall
    mov     rdi, [rel fd]
    mov     eax, SYS_CLOSE
```

```
    syscall
    mov     edi, 1
    jmp     .exit

; --- helper: log_error (writes message to stderr) ---
log_error:
    mov     eax, SYS_WRITE
    mov     edi, 2                ; stderr
    syscall
    ret

; --- helper: secure_zero (RDI = ptr, RSI = size) ---
secure_zero:
    mov     rcx, rsi
    xor     eax, eax
    cld
    rep     stosb
    ret

.exit:
    mov     eax, SYS_EXIT
    syscall
```

This program demonstrates all the defensive principles: file descriptor validation, allocation error checking, bounding the read size, null-termination within the buffer, sanitization before deallocation, and logging of errors to stderr — all without any libc.

Important

Defensive programming adds instructions and branches, but it catches bugs at the source rather than letting them propagate into mysterious crashes. In assembly, where every mistake can corrupt memory silently, these checks are a vital investment.

Putting It All Together

Buffer safety is the foundation of reliable assembly software. By understanding how overflows occur, applying safe copy routines, implementing explicit bounds checks, sanitizing memory upon deallocation, and adopting a defensive posture throughout the codebase, you can write NASM programs that are robust against both accidental corruption and deliberate exploitation. The next chapters will expand on structured error handling and integration with the Linux signal and process management facilities.

Part VII

INTEGRATION AND INTERFACING

35

NASM with C and C++ on Linux

In This Chapter

Combining hand-crafted assembly with C or C++ gives the best of both worlds: direct hardware control for performance-critical sections, and the productivity of a high-level language for the rest. This chapter explains how to link NASM object files with code produced by GCC (or Clang) on Fedora 43. You will learn to match the System V AMD64 calling convention, export assembly functions to C, import C functions into NASM, and design hybrid projects that compile and link seamlessly. All examples have been tested with NASM 2.16 and GCC 14 on Fedora 43. The content follows the System V AMD64 ABI, the NASM manual, and the C/C++ standards for cross-language linkage.

35.1 Linking Assembly with C

On Linux, all native code for x86-64 uses the ELF format (Executable and Linkable Format). NASM with `-f elf64` produces a standard ELF object file (`.o`) that can be directly linked with objects produced by GCC or Clang. The GNU linker (`ld`) or the compiler driver (`gcc`) handles the combination.

Basic Linking Steps

1. Compile the C/C++ source to an object file: `gcc -c file.c -o file.o`.
2. Assemble the NASM source: `nasm -f elf64 -o asm.o asm.asm`.
3. Link the objects together with any required libraries using `gcc` (or directly with `ld`).

The linker resolves symbols across the objects. A NASM label declared `global my_func` corresponds directly to a C function named `my_func`, without any leading underscore. When using C++, the

function must be declared with `extern "C"` to prevent name mangling.

Tip

When calling C++ code from assembly, always use a C-linkage wrapper function declared with `extern "C"` in the C++ source. This ensures the symbol name matches what NASM expects.

35.2 Calling Convention Compatibility

The System V AMD64 ABI is the single standard for all native 64-bit code on Linux, whether written in assembly, C, or C++. This uniformity eliminates the need for multiple calling conventions.

Register Assignment Review

- First six integer/pointer arguments: `RDI`, `RSI`, `RDX`, `RCX`, `R8`, `R9`.
- Additional arguments: pushed right-to-left onto the stack.
- Floating-point arguments: `XMM0` – `XMM7`.
- Return value: integer in `RAX`, floating-point in `XMM0`.
- The stack must be 16-byte aligned at the `CALL` instruction. No mandatory shadow space; the 128-byte *red zone* below `RSP` is available to leaf functions only.
- Cellee-saved (non-volatile) registers: `RBX`, `RBP`, `R12`–`R15`. Call-clobbered: all others.

A NASM function called from C must obey these rules exactly, including preserving any callee-saved registers it modifies. When calling a C function from NASM, the assembly code must align the stack, place arguments in the correct registers, and, for variadic functions like `printf`, set `AL` to the number of XMM registers used.

Example: NASM Function Called from C

Code: asm_add.asm — Adding two integers, callable from C

```
; asm_add.asm
bits 64
default rel

section .text
global asm_add

; int asm_add(int a, int b);
; EDI = a, ESI = b
asm_add:
    ; leaf function, no stack adjustment needed
    lea     eax, [edi + esi]    ; 32-bit result zero-extends to RAX
    ret
```

The C prototype:

```
extern int asm_add(int a, int b);
```

The function uses `lea` to add the two 32-bit arguments. The return value in `EAX` is automatically zero-extended to 64 bits by the hardware convention.

Calling C from NASM

When NASM calls a C function, it must follow the ABI: put arguments in registers, align the stack, and (for variadic functions) set `AL`. For example, calling `printf`:

Code: call_printf.asm — Calling printf from NASM

```
; call_printf.asm
bits 64
default rel

extern printf
extern exit

section .data
fmt db 'The sum is %d', 10, 0

section .text
global main
main:
    push    rbp
    mov     rbp, rsp
    sub     rsp, 16             ; align stack (entry RSP 8 mod 16, sub rsp,8 to align; we do
    ↪  16)
    lea     rdi, [rel fmt]      ; first argument
    mov     esi, 3+5            ; second argument
    xor     eax, eax            ; no floating-point args
    call    printf
    xor     edi, edi
    call    exit
```

This example requires linking with the C library (which is automatic when using `gcc`). Note that `main` is being called by the C runtime, so the stack is already misaligned by 8 (return address). We realign by subtracting 8 more (here we subtract 16, which also aligns). Setting `AL` to zero before calling `printf` is mandatory; otherwise `printf` may crash.

35.3 Exporting Functions

To make a NASM function visible to C/C++, declare the label `global` in NASM, and provide a matching declaration in the C code.

Exporting a More Complex Function

Consider a function that computes the dot product of two 64-bit integer arrays.

Code: dot_product.asm — NASM function exporting multiple parameters

```
; dot_product.asm
bits 64
default rel

section .text
global dot_product

; int64_t dot_product(const int64_t* a, const int64_t* b, size_t count);
; RDI = a, RSI = b, RDX = count
dot_product:
    push    rbx                 ; save callee-saved register
    xor     eax, eax            ; sum = 0
    test    rdx, rdx
    jz      .done
    xor     ecx, ecx            ; index = 0
.loop:
    mov     r8, [rdi + rcx*8]   ; a[i]
    imul    r8, [rsi + rcx*8]   ; a[i] * b[i]
    add     rax, r8
    inc     rcx
    cmp     rcx, rdx
    jb      .loop
.done:
    pop     rbx
    ret
```

C declaration:

```
#include <stdint.h>
extern int64_t dot_product(const int64_t* a, const int64_t* b, size_t count);
```

Data Export: Global Variables

NASM can export data symbols placed in `.data` or `.rodata` sections.

Code: export_data.asm — Exporting a global variable

```
; export_data.asm
section .data
global shared_counter
shared_counter  dq  0
```

C declaration:

```
extern long long shared_counter;
```

35.4 Importing Functions

From NASM, you can call any C function by declaring it with `extern`. The symbol name must match exactly. For C++, wrap the declaration in `extern "C"`.

Example: Using C Library Function `malloc`

Code: c_malloc.asm — Using malloc from NASM

```
; c_malloc.asm
bits 64
default rel

extern malloc
extern free

section .text
global test_alloc
test_alloc:
    push    rbp
    mov     rbp, rsp
    sub     rsp, 16          ; align stack
    mov     edi, 1024        ; size = 1024
    call    malloc            ; RAX = pointer or NULL
    test    rax, rax
    jz      .fail
    ; gracefully use memory...
    mov     rdi, rax
    call    free
.fail:
    leave
    ret
```

Linking with `gcc` automatically includes the C runtime; no extra libraries are needed.

Calling C++ Methods

To call a C++ method from assembly, provide a C-linkage wrapper:

Code: C++ wrapper for method call

```
// wrapper.cpp
class MyClass {
public:
    int method(int x);
};
extern "C" int call_method(MyClass* obj, int arg) {
    return obj->method(arg);
}
```

Then in NASM:

```
extern call_method
; RDI = object pointer, ESI = argument
call call_method
```

35.5 Hybrid Project Design

A well-organised project separates assembly and C sources for clarity.

```
project/
  src/
    main.c
    asm/
      dot_product.asm
      utils.asm
  include/
    asm_interface.h
  obj/          (generated object files)
  bin/          (executable)
  Makefile
```

Build Automation with a Shell Script

Code: build.sh — Simple build script

```
#!/bin/bash
mkdir -p obj bin
nasm -f elf64 -o obj/dot_product.o src/asm/dot_product.asm
nasm -f elf64 -o obj/utils.o src/asm/utils.asm
gcc -c -o obj/main.o src/main.c
gcc -o bin/app obj/main.o obj/dot_product.o obj/utils.o -lm
```

Makefile

Code: Makefile

```
NASM = nasm
CC   = gcc
OBJ  = obj/main.o obj/dot_product.o obj/utils.o
TARGET = bin/app

$(TARGET): $(OBJ)
^^I$(CC) -o $@ $^ -lm

obj/main.o: src/main.c
^^I$(CC) -c -o $@ $<

obj/dot_product.o: src/asm/dot_product.asm
^^I$(NASM) -f elf64 -o $@ $<

obj/utils.o: src/asm/utils.asm
^^I$(NASM) -f elf64 -o $@ $<
```

A Complete Hybrid Example

Let's build a small program: C provides the I/O, NASM supplies a fast checksum function.

Code: main.c — C main program

```c
// main.c
#include <stdio.h>
#include <stdint.h>
#include <string.h>

extern uint32_t checksum(const char* data, size_t len);

int main(void) {
    const char* msg = "Hello, hybrid world!";
    uint32_t crc = checksum(msg, strlen(msg));
    printf("Checksum of \"%s\" is %08X\n", msg, crc);
    return 0;
}
```

Code: checksum.asm — NASM checksum implementation

```
; checksum.asm
bits 64
default rel

section .text
global checksum

; uint32_t checksum(const char* data, size_t len);
; RDI = data, RSI = len
checksum:
    push    rbx
    mov     eax, 0xFFFFFFFF      ; initial value
    test    rsi, rsi
    jz      .done
    xor     ecx, ecx             ; index = 0
.loop:
    movzx   r8d, byte [rdi + rcx]
    xor     eax, r8d
    shr     r8d, 1
    inc     rcx
    cmp     rcx, rsi
    jb      .loop
.done:
    pop     rbx
    ret
```

Build and run:

```
nasm -f elf64 -o checksum.o checksum.asm
gcc -c -o main.o main.c
gcc -o demo main.o checksum.o
./demo
```

CMake Integration

CMake can handle NASM via the `ASM_NASM` language.

Code: CMakeLists.txt

```
cmake_minimum_required(VERSION 3.10)
project(HybridDemo C ASM_NASM)

set(CMAKE_ASM_NASM_OBJECT_FORMAT elf64)

add_executable(demo main.c checksum.asm)
```

The executable will be linked with the C runtime automatically.

Important

When mixing NASM with C++ exceptions, assembly functions should not throw across frames unless they are specially designed. Simple leaf functions are safe. If a C++ function called from assembly throws, the exception must be caught by a C++ handler; crossing an assembly frame without the necessary unwind information can cause a crash.

Putting It All Together

Interfacing NASM with C and C++ on Linux is straightforward thanks to the unified System V AMD64 ABI. By exporting global symbols from NASM and declaring them with the correct C linkage, assembly routines become drop-in replacements for C functions. Hybrid projects combine the performance of assembly with the rich ecosystem of C libraries and tools.

36

External Libraries on Linux

In This Chapter

No serious Linux application is built entirely from scratch. External libraries—whether static `.a` archives or shared `.so` objects—supply reusable code that handles everything from mathematics to networking. This chapter explains how NASM-generated object files interact with external libraries on Fedora 43. You will learn the difference between static and shared libraries, the role of symbolic links and sonames, how the linker resolves symbols from libraries, how to manage library dependencies, and how to configure build scripts to incorporate both system and third-party libraries. Real, assemble-ready NASM examples show how to call functions from the C library, the math library, and custom shared objects. All information is based on the ELF specification, the GNU linker manual, and the behaviour of GCC and ld on Fedora 43.

36.1 Static and Shared Libraries

Libraries come in two fundamental forms on Linux: static and dynamic (shared).

Static Libraries (`.a`)

A static library is an archive of object files, created with the `ar` utility. When you link against a static library, the linker extracts the required object modules and copies their code and data directly into your executable. After linking, the library file is no longer needed, and all symbols are fully resolved. Static linking increases executable size but eliminates external runtime dependencies.

NASM object files can be used to create a static library, and NASM code can call functions from any static library that follows the System V AMD64 ABI.

Code: Creating a static library with ar

```
nasm -f elf64 -o add.o add.asm
nasm -f elf64 -o mul.o mul.asm
ar rcs libmymath.a add.o mul.o
```

The resulting `libmymath.a` can be linked into other programs using `-lmymath` (after specifying the path with `-L.`).

Shared Libraries (`.so`)

A shared library (dynamic library) is loaded at runtime by the dynamic linker (`ld-linux.so`). The linker records the required library and its symbols, but no code is copied into the executable. Instead, the program text is mapped at load time, and the symbol addresses are resolved via the Procedure Linkage Table (PLT) and Global Offset Table (GOT).

Shared libraries allow multiple programs to share the same physical pages of code, keep executables small, and permit library updates without recompilation (as long as the ABI remains compatible). They must be compiled as position-independent code (PIC) using the `-fPIC` flag (GCC), but NASM objects for ELF64 are already position-independent if you use `default rel` and avoid absolute addresses.

A shared library is created with `gcc -shared` (or `ld -shared`):

Code: Creating a shared library

```
nasm -f elf64 -o mylib.o mylib.asm
gcc -shared -o libmylib.so mylib.o
```

The resulting `.so` can be linked with other programs using `-L. -lmylib`.

Soname and Symbolic Links

Linux shared libraries often have a naming convention like `libfoo.so.1`, with a symbolic link `libfoo.so` pointing to the real file. The linker embeds the soname (`libfoo.so.1`) into the executable, so at runtime the dynamic linker loads that specific version. NASM objects do not need to care about this; the build system handles the linking.

36.2 Static Libraries in Depth

A static library (`.a`) is simply an archive. To link with it, use GCC's `-L` to specify the directory, and `-l` to give the library name without the `lib` prefix or `.a` extension.

Code: Linking with a static library

```
gcc -o prog main.o -L./lib -lmymath
```

If the static library is not built with position-independent code (which NASM objects are by default when using `default rel`), that's fine for statically-linked executables.

Symbol Resolution Order

When linking with GCC, the order of libraries on the command line matters. The linker processes arguments left to right, and it only pulls in objects from a library if they satisfy an undefined symbol seen so far. Therefore, libraries must come after the objects that reference them.

Code: Correct library order

```
gcc -o prog main.o util.o -lm -lpthread
```

If you have circular dependencies among static libraries, you can list them twice or use the `--start-group` / `--end-group` linker options.

36.3 Shared Libraries and Dynamic Linking

When you link with a shared library, the linker creates a dynamic executable. At load time, `ld-linux.so` maps the shared library into the process and resolves the PLT entries.

Linking with a Shared Library

Use `-L` and `-l` as with static libraries. GCC will automatically prefer shared libraries unless `-static` is given.

Code: Linking with a custom shared library

```
gcc -o prog main.o -L. -lmylib
```

To run the program, the system must be able to find the shared library. The dynamic linker searches directories specified in `LD_LIBRARY_PATH`, the `/etc/ld.so.conf` cache, and standard system paths (`/lib, /usr/lib`, etc.). For development, you can set `LD_LIBRARY_PATH=. ./prog` or use the `-rpath` linker flag to embed a search path.

Using `dlopen` for Runtime Loading

As covered in an earlier chapter, `dlopen` allows you to load a shared library and resolve symbols at runtime without any link-time dependency. This is ideal for plugins.

Exporting Symbols from NASM

To make a function visible from a shared library, declare it `global` in NASM. The resulting symbol will be exported by the linker when creating the `.so`. You can control visibility with `extern "C"` in C, but in NASM simply use `global`. For finer control (e.g., hiding internal functions), you can use a version script or the `visibility` attribute when linking, but typically all `global` symbols are exported.

36.4 Symbol Resolution in Depth

Symbol resolution across libraries follows ELF rules. The linker resolves undefined references that appear in object files and libraries, pulling in archive members as needed.

Strong and Weak Symbols

By default, all symbols defined in an object file with `global` are strong. Weak symbols (from `weak` declaration in C) can be overridden by strong symbols. NASM does not have a direct weak symbol directive; you can create weak symbols using `extern` and a separate definition or through linker scripts.

Common Symbols

In C, uninitialised global variables (without `extern`) become *common* symbols that the linker merges. NASM does not generate common symbols; data in `.bss` with `global` is a regular definition.

Multiple Definitions

If the same global symbol is defined in two object files, the linker reports a duplicate symbol error. Avoid defining the same global label in multiple `.asm` files; use `extern` in all but one.

PLT and GOT Internals

When you call a shared library function from NASM, the linker creates a PLT stub. Your `call` instruction jumps to the PLT, which loads the actual address from the GOT. This is transparent; you just write `call myfunc`. No special directives are needed.

To obtain a pointer to a shared library function (rather than calling it), you can use the `@GOTPCREL` syntax:

Code: Getting a function pointer from GOT

```
extern myfunc
lea     rax, [myfunc@GOTPCREL]   ; RAX = address of GOT entry
mov     rax, [rax]               ; RAX = actual function address
```

This is equivalent to reading the IAT on Windows and is needed for function pointer tables or callbacks.

36.5 Library Dependencies

Managing dependencies correctly prevents link-time errors and runtime crashes.

Direct Dependencies

When your program uses a library `A` that itself depends on `B`, you must also link against `B` (unless linking statically and the static library already includes the needed objects). For shared libraries, the dynamic linker will load `B` automatically if `A` was linked with `-lB` at creation time and the dependency is recorded in `A.so`.

To see the shared library dependencies of an executable or shared object, use `ldd`:

Code: Checking dependencies with ldd

```
ldd ./prog
```

Circular Dependencies

Circular dependencies between shared libraries are allowed but can cause issues with static initialisation order. It is best to avoid them. Circular dependencies among static libraries are resolved by grouping or repeating the libraries on the command line.

Managing Dependencies in Makefiles

When building a shared library that uses another library, pass `-l` flags to the linker during creation. For example:

Code: Creating a shared library that depends on libm

```
gcc -shared -o libcalc.so calc.o -lm
```

Now `libcalc.so` will automatically include a dependency on `libm.so`, and programs linking against `libcalc` will not need to explicitly specify `-lm`.

Using `pkg-config`

Many libraries provide a `.pc` file for `pkg-config`, which supplies the correct compiler and linker flags. In a Makefile, you can use:

Code: Using pkg-config in a command

```
gcc -o prog main.o `pkg-config --cflags --libs libcurl`
```

This simplifies building with complex dependencies.

36.6 Build Configuration

A robust build system locates headers, libraries, and sets appropriate flags. On Fedora, development packages (e.g., `openssl-devel`) install headers and `.so` files into standard locations (`/usr/include`, `/usr/lib64`). For custom libraries, you can use `-I` and `-L`.

Example Makefile with Multiple Libraries

Code: Makefile linking with a custom static library and system library

```
NASM = nasm
CC   = gcc
OBJ  = main.o asm_utils.o
LIBS = -L./lib -lmylib -lm -lpthread
TARGET = prog
```

```
$(TARGET): $(OBJ)
^^I$(CC) -o $@ $^ $(LIBS)

main.o: main.c
^^I$(CC) -c -o $@ $<

asm_utils.o: asm_utils.asm
^^I$(NASM) -f elf64 -o $@ $<
```

CMake Integration

CMake handles NASM and libraries elegantly:

Code: CMakeLists.txt for a project with NASM and a custom library

```
cmake_minimum_required(VERSION 3.10)
project(CustomLib C ASM_NASM)

set(CMAKE_ASM_NASM_OBJECT_FORMAT elf64)

add_library(mymath STATIC math.asm)  # static library from NASM
target_include_directories(mymath PUBLIC include)

add_executable(demo main.c)
target_link_libraries(demo mymath m)   # link with mymath and libm
```

This builds a static library `libmymath.a` from `math.asm`, then links `demo` against it.

Using a System Library (e.g., OpenSSL)

Install the development package (`sudo dnf install openssl-devel`). Then include the header in C, and link with `-lssl -lcrypto`. For pure NASM, you would call the functions directly (declaring them as `extern`), but using C as a thin wrapper is often simpler.

Complete Example: Linking with a Custom Shared Library

Suppose you have a NASM source `gettime.asm` that exports a function to call `clock_gettime` using a system call, and you want to package it as `libtstamp.so`.

Code: gettime.asm — Simple time function

```
; gettime.asm
bits 64
default rel

section .text
global get_nanoseconds

; uint64_t get_nanoseconds(void);
; Returns a simple counter (for demo, just 0xDEAD)
get_nanoseconds:
    mov     rax, 0xDEAD
```

```
    ret
```

Build the shared library:

```
nasm -f elf64 -o gettime.o gettime.asm
gcc -shared -o libtstamp.so gettime.o
```

Now write a C program:

Code: main_time.c

```c
#include <stdio.h>
extern unsigned long long get_nanoseconds(void);

int main() {
    printf("Timestamp: %llu\n", get_nanoseconds());
    return 0;
}
```

Link and run:

```
gcc -c -o main_time.o main_time.c
gcc -o timeapp main_time.o -L. -ltstamp
LD_LIBRARY_PATH=. ./timeapp
```

The output will be `Timestamp: 57005` (0xDEAD in decimal).

Tip

When distributing a shared library, install it to `/usr/local/lib` or `/usr/lib64` and run `ldconfig` to update the linker cache. This avoids needing `LD_LIBRARY_PATH`.

Putting It All Together

External libraries are the key to leveraging the vast ecosystem of open-source code on Linux. By understanding the distinction between static and shared libraries, the linking process, symbol resolution, and dependency management, you can integrate NASM code with any C library. The build infrastructure on Fedora, with `gcc`, `make`, and `cmake`, provides a smooth path from reusable assembly modules to professional-grade executables.

37

Dynamic Linking, PLT/GOT, and Symbol Resolution on Linux

In This Chapter

When a NASM program calls a function from a shared library (e.g., `printf` from `libc.so`), the journey from the `CALL` instruction to the actual function code involves a series of indirections managed by the linker and the dynamic linker. This chapter explains how the Procedure Linkage Table (PLT) and the Global Offset Table (GOT) work together to provide lazy binding, how the dynamic linker (`ld-linux.so`) loads shared objects and resolves symbols at load time, the alternative of runtime loading with `dlopen` and `dlsym`, and the complete flow of a symbol from an `extern` declaration to an executed function. Understanding these internals demystifies linker errors, enables advanced techniques like function hooking, and gives you full control over how your assembly code interacts with the vast Linux library ecosystem. All information is drawn from the System V AMD64 ABI, the ELF specification, and the behaviour observed on Fedora 43.

37.1 The Dynamic Linking Mechanism

On Linux, every executable that uses shared libraries contains a dynamic section (`.dynamic`) that instructs the dynamic linker on what to load and how to resolve symbols. The linker builds a *Procedure Linkage Table* (PLT) and a *Global Offset Table* (GOT) for each external function. These tables allow the program to call a shared library function without knowing its address in advance, and to defer the resolution to the first call (lazy binding).

The PLT and GOT in Detail

A shared library function call like `call printf` in your NASM code does not directly jump to `printf`. Instead, the linker redirects the call to a PLT entry—a small stub residing in the `.plt` section. The PLT stub loads the address of the function from the corresponding GOT entry and jumps to it. When the program starts, the GOT entry points back into the dynamic linker, so the first call triggers symbol resolution. The dynamic linker finds the function, writes the real address into the GOT, and then jumps to it. Subsequent calls go through the PLT but now load the resolved address directly, incurring minimal overhead.

This mechanism is transparent. You simply declare `extern printf` and write `call printf`. The linker sets up the PLT and GOT automatically when you link with `gcc` (or with `ld` and the appropriate libraries). No special directives are needed in NASM.

Accessing the GOT Directly

Sometimes you need to store a function pointer rather than calling the function. You can retrieve the function address from the GOT using the `@GOTPCREL` suffix:

```
extern printf
lea     rax, [printf@GOTPCREL]   ; RAX = address of GOT entry for printf
mov     rax, [rax]               ; RAX = actual address of printf
```

This is similar to using the `__imp_` symbol on Windows and allows you to build tables of function pointers or pass them as callbacks.

Analogous to the Windows IAT

The GOT serves the same purpose as the Import Address Table (IAT) on Windows, while the PLT corresponds to the import thunks. The key difference is that the GOT lazily resolves symbols by default, whereas Windows performs full resolution at load time (unless delay-loading is used). The principles, however, are identical.

37.2 Shared Object Loading and Search Paths

When a dynamically-linked executable is launched, the kernel starts the program interpreter `/lib64/ld-linux-x86-` (or a similar path). This dynamic linker:

1. Loads the executable's dynamic segment and determines which shared libraries are required.
2. Searches for each library in a predefined set of directories:
   - Directories listed in the `LD_LIBRARY_PATH` environment variable (for development).
   - Paths embedded in the executable's `RPATH` or `RUNPATH` (if specified at link time with `-rpath`).
   - The system cache `/etc/ld.so.cache`, updated by `ldconfig`.
   - Trusted default directories: `/lib64`, `/usr/lib64`, etc.
3. Maps each library's segments into memory, handles any initialisation functions, and resolves symbols, filling the GOT entries (eagerly or lazily).

The entire process is similar to Windows's DLL loading, but the search rules and the dynamic linker's name differ.

37.3 Lazy Binding (Default Behaviour)

Lazy binding is the default and is enabled unless the `LD_BIND_NOW` environment variable is set or the executable was linked with `-z now`. With lazy binding, the GOT entries initially point to the dynamic linker's resolver. The resolver is invoked on the first call, and the GOT is updated. This speeds up process startup but may slightly delay the first call.

You can force immediate binding at link time by passing `-Wl,-z,now` to GCC. This causes all PLT entries to be resolved at load time, eliminating the lazy resolution overhead at the cost of increased startup time.

37.4 Runtime Dynamic Loading: `dlopen` and `dlsym`

Runtime loading allows you to load a shared library and resolve a function at any point during execution, without having the library listed at link time. This is the equivalent of `LoadLibrary` / `GetProcAddress`. The functions are:

Code: Prototypes for dynamic loading functions

```
extern dlopen
extern dlsym
extern dlclose
extern dlerror

; void *dlopen(const char *filename, int flags);
; RDI = filename, ESI = flags (RTLD_LAZY = 1, RTLD_NOW = 2)

; void *dlsym(void *handle, const char *symbol);
; RDI = handle, RSI = symbol name

; int dlclose(void *handle);
; RDI = handle

; char *dlerror(void);
```

Linking with `-ldl` is required when using these functions. The following NASM example loads `libm.so.6`, gets a pointer to `sin`, calls it with a double argument, and prints the result using `printf`. No `extern sin` is needed—the function is resolved entirely at runtime.

Code: listing37-1.asm — Runtime linking of sin from libm

```
; listing37-1.asm
; Assemble: nasm -f elf64 -o listing37-1.o listing37-1.asm
; Link:     gcc -no-pie -o listing37-1 listing37-1.o -ldl
bits 64
default rel
```

```
extern dlopen
extern dlsym
extern dlclose
extern dlerror
extern printf
extern exit

section .data
lib_name db 'libm.so.6',0
fn_name  db 'sin',0
fmt      db 'sin(1.0) = %f',10,0
one      dq 1.0

section .bss
hLib     resq 1
pSin     resq 1

section .text
global main
main:
    push    rbp
    mov     rbp, rsp
    sub     rsp, 32

    ; load libm
    mov     edi, 1                  ; RTLD_LAZY
    lea     rsi, [rel lib_name]
    call    dlopen
    test    rax, rax
    jz      .fail
    mov     [rel hLib], rax

    ; get sin
    mov     rdi, rax
    lea     rsi, [rel fn_name]
    call    dlsym
    test    rax, rax
    jz      .fail_close
    mov     [rel pSin], rax

    ; call sin(1.0)
    movsd   xmm0, [rel one]
    call    rax                      ; result in xmm0

    ; print using printf
    lea     rdi, [rel fmt]
    mov     eax, 1                  ; one XMM register used
    call    printf

    mov     rdi, [rel hLib]
    call    dlclose

    xor     edi, edi
    call    exit
```

```
.fail_close:
    mov     rdi, [rel hLib]
    call    dlclose
.fail:
    call    dlerror
    mov     rdi, rax
    call    puts                    ; print error message
    mov     edi, 1
    call    exit
```

This technique enables plugin systems and conditional library usage.

37.5 Symbol Flow: From Declare to Execute

Let's trace the complete journey of a symbol from `extern printf` to the moment the function executes.

Step 1 – Assembly (NASM)

When NASM encounters `extern printf` and later `call printf`, it generates a relocation of type `R_X86_64_PLT32` (or `R_X86_64_PC32`) against the undefined symbol `printf`. The object file records that this call site needs the linker to provide the address of `printf`.

Step 2 – Static Linking

The linker (invoked via `gcc`) processes the object file and the C library (referenced implicitly). It does not know the absolute address of `printf`. Instead, it creates a PLT entry for `printf` in the executable and sets the call's target to that PLT entry. It also creates a GOT entry and a dynamic relocation entry (`R_X86_64_JUMP_SLOT`) that tells the dynamic linker to fill the GOT entry with the actual address at runtime. The executable's dynamic section lists `libc.so.6` as a dependency.

Step 3 – Load Time (Dynamic Linker)

When the program is launched, `ld-linux.so` loads `libc.so.6`. It processes all `R_X86_64_JUMP_SLOT` relocations. With lazy binding (the default), the GOT entry initially contains the address of the dynamic linker's resolver routine, not the address of `printf`. The resolver will be called on the first invocation.

Step 4 – First Call

Your code reaches `call printf`. Because the call points to the PLT stub, the stub loads the GOT entry (which points to the resolver) and jumps to it. The resolver determines that `printf` must be resolved, finds it in `libc.so.6`, updates the GOT entry with the real address, and then jumps to `printf`. The function executes. Subsequent calls go through the PLT, but now the GOT holds the real address, so the overhead is just a few instructions.

If you had used `call [printf@GOTPCREL]` (i.e., indirect call through the GOT directly), the resolution would still happen on first call, but the stub would be avoided. However, this form is less

common; the standard PLT-based call is sufficient.

Step 5 – Process Termination

When the process exits, the dynamic linker unmaps the shared libraries. The GOT entries become invalid, but this is irrelevant as the process is gone.

37.6 Delayed Loading vs. Manual Loading

The PLT lazy binding is a form of delay loading. For a library that may not be present, you cannot rely on the standard PLT; you must use `dlopen` and `dlsym` to check for presence and fall back gracefully. This is the Linux equivalent of Windows delay-loading with `/DELAYLOAD` or manual `LoadLibrary`.

Using a Fallback Library

A program can try to load a newer version of a library, and if unavailable, fall back to an older one:

```
h = dlopen("libfoo.so.2", RTLD_LAZY);
if (!h) h = dlopen("libfoo.so.1", RTLD_LAZY);
```

This logic can be written in C and called from NASM, or implemented entirely in assembly by calling `dlopen` and handling the NULL return.

37.7 Inspecting the Dynamic Section

Tools like `readelf -d prog` show the dynamic dependencies and relocation information. `objdump -R prog` lists the dynamic relocations, including PLT entries. For a running process, `cat /proc/<pid>/maps` shows the loaded libraries.

```
readelf -d ./myprog          # dynamic section
objdump -R ./myprog          # dynamic relocations (JUMP_SLOT entries)
ldd ./myprog                 # shared library dependencies
```

These commands parallel the Windows `dumpbin /dependents` and `dumpbin /imports`.

Putting It All Together

The dynamic linking mechanism on Linux—PLT, GOT, and the dynamic linker—is a elegant and efficient system that makes shared libraries transparent to the programmer. Whether you use the default lazy binding, force immediate resolution, or load libraries at runtime with `dlopen`, the flow is always: resolve a symbol, store its address in a table, and call through the table. Understanding these details lets you debug symbol resolution failures, build plugin architectures, and write assembly programs that fully leverage the vast world of Linux libraries.

38

Hybrid Programming on Linux

In This Chapter

Mixing NASM assembly with C or C++ yields programs that combine the raw speed and control of hand-crafted machine code with the productivity and rich ecosystem of modern compilers. This chapter explores the system architecture of hybrid applications, shows how to profile and optimize the boundary between languages, covers debugging techniques for mixed source code, outlines sound integration strategies, and presents real-world use cases where hybrid programming shines. Every technique is based on the System V AMD64 ABI, the GNU toolchain, and practical testing with GCC and GDB on Fedora 43.

38.1 System Architecture

A hybrid program consists of at least one C (or C++) module and one or more NASM assembly modules, all linked into a single executable or shared library. The architecture must respect the System V AMD64 calling convention uniformly across all modules; the linker and dynamic linker do not distinguish between object files based on source language. The fundamental principle is: *every function, whether written in C or NASM, must obey the same ABI.*

Module Boundaries

The boundary between C and assembly is simply a function call. Data can be passed by value (integers, pointers, floating-point) or by reference. The NASM side sees the C world as a collection of `extern` functions; the C side sees NASM routines as `extern` functions with C linkage (using `extern "C"` when compiled as C++). No wrapper libraries are required if the entry point is compiled by GCC and the assembly modules contain only functions that are called.

Code: Hybrid architecture skeleton

```c
// main.c
#include <stdio.h>
#include <stdint.h>
extern int64_t asm_compute(int64_t a, int64_t b);
int main(void) {
    int64_t result = asm_compute(100, 200);
    printf("Result = %lld\n", result);
    return 0;
}
```

Code: asm_compute.asm

```
; asm_compute.asm
bits 64
default rel
section .text
global asm_compute
asm_compute:
    ; RDI = a, RSI = b
    lea     rax, [rdi + rsi]
    ret
```

Data can be shared through global variables. A NASM `global` label in `.data` corresponds to a C `extern` variable. Size and alignment must match.

Startup and Shutdown

The C runtime initialises the heap, stdio, and calls `main`. Assembly functions can use standard C library functions like `malloc` or `printf` because they are linked with the runtime library. If the entry point were written in NASM (using `_start`), the CRT would not be initialised, and calling such functions would be unsafe. Hybrid projects normally let the C module own the entry point.

38.2 Performance Optimization

Performance gains from hybrid programming come from identifying hot spots in the C/C++ code and rewriting them in assembly. The overhead of a function call is not negligible, so candidate functions should be sufficiently heavy.

Profiling the Boundary

Use the Linux `perf` tool, `gprof`, or `valgrind --tool=callgrind` to profile the C program. Once the hot functions are identified, they can be extracted into assembly. Because NASM functions have global symbols, `perf report` can show the time spent in them. To enable accurate profiling, compile with debug info (`-g`) and link with the same options.

```
perf record ./hybrid_app
perf report
```

Avoiding Tiny Function Calls

If a function is only a few instructions, the `call`/`ret` overhead may dominate. In such cases, consider using GCC inline assembly (`asm` extension) to embed the assembly directly in the C source. For pure NASM functions, restructuring the algorithm to do more work per call can justify the call cost.

Register Use and SIMD

NASM functions that process large datasets can keep working values in registers and use aligned SIMD instructions. Avoid unnecessary stack usage. The C compiler already optimises register allocation, but in assembly you have complete control. For a compute-intensive loop, unroll it, use all available XMM/YMM registers, and consider software prefetching. The hybrid approach lets you hand-tune the 5% of code that consumes 95% of the time.

Leveraging SIMD from NASM

The C compiler may not always vectorise loops optimally. A NASM function using SSE or AVX instructions can dramatically accelerate multimedia, cryptography, or numerical computations. Ensure that any data passed from C is aligned to the required boundary (e.g., 16-byte for SSE, 32-byte for AVX). In C, use `aligned_alloc` or alignment attributes.

Code: dot_product_sse.asm — NASM SSE dot product, callable from C

```
; dot_product_sse.asm
bits 64
default rel
section .text
global dot_product_sse

; float dot_product_sse(const float* a, const float* b, int count);
; RDI = a, RSI = b, EDX = count
dot_product_sse:
    push    rbx
    xorps   xmm0, xmm0            ; sum = 0
    test    edx, edx
    jz      .done
    xor     eax, eax
.loop:
    movss   xmm1, [rdi + rax*4]
    mulss   xmm1, [rsi + rax*4]
    addss   xmm0, xmm1
    inc     eax
    cmp     eax, edx
    jb      .loop
.done:
    pop     rbx
    ret
```

C prototype: `extern float dot_product_sse(const float* a, const float* b, int count);`. The first three arguments are in RDI, RSI, and EDX as per the ABI.

38.3 Debugging Mixed Code

Debugging a program containing both C and assembly requires a debugger that can navigate between the two representations. GDB seamlessly handles this.

Generating Debug Information for NASM

Assemble NASM with `-g` to produce DWARF debug information. Then, when linking with `gcc -g`, GDB can display the assembly source, set breakpoints on labels, and single-step through NASM instructions. Example:

```
nasm -f elf64 -g -o asm.o asm.asm
gcc -g -no-pie -o prog main.c asm.o -lm
gdb ./prog
break asm_compute
run
```

GDB will show the source lines of `asm_compute.asm` if the file is accessible, or the disassembly with mixed source if using `layout asm`.

Viewing Registers and Memory

Inside a NASM function, the `info registers` command shows all GPRs, the stack pointer, and flags. `x/10gx $rsp` dumps the stack. To inspect arguments, you can refer to the registers: `p $rdi`, etc.

Detecting Calling Convention Mismatches

The most common hybrid bug is an ABI error: wrong argument registers, misaligned stack, or missing variadic argument setup. Set a breakpoint at the function entry and examine `RDI`, `RSI`, etc. For variadic functions like `printf`, ensure `AL` is set correctly. Misaligned stack before a `CALL` can be caught by checking `$rsp` before the call: it must be a multiple of 16.

Backtrace Across Assembly

If the NASM function uses a frame pointer (standard prologue: `push rbp; mov rbp, rsp`), GDB can unwind the stack and display the call chain. It is recommended to include a frame pointer in functions that call other functions, as it greatly simplifies debugging.

38.4 Integration Strategy

A disciplined approach to hybrid programming keeps the project maintainable.

Identify Bottlenecks First

Profile a pure C (or C++) implementation. Do not optimise prematurely. Focus on functions that:

- Are called frequently and account for significant CPU time.
- Perform operations that map well to SIMD or special x86-64 instructions.

- Operate on large memory blocks where you can eliminate C abstraction overhead.
- Need direct access to CPU features such as control registers, performance counters, or specific instructions.

Define Clean Interfaces

Every NASM function should have a clear prototype in a C header file. Document which registers are volatile and which are preserved. Use `extern "C"` in C++ headers. For shared data structures, define the layout in both C (using `struct`) and NASM (using `struc` or manual offsets). A `point.inc` included by NASM and a corresponding `point.h` included by C ensures binary compatibility.

Manage Memory Ownership

Decide whether the C side or the assembly side allocates and frees memory. Avoid splitting ownership. Usually, C manages memory with `malloc`/`free` or `mmap`/`munmap`, and assembly receives buffers to work with. If assembly allocates, it must provide a corresponding deallocation function.

Testing Strategy

Unit-test NASM functions from a C test harness. This is much easier than testing in pure assembly. Write a C program that calls the assembly function with various inputs and compares results using `assert` or `printf`. This also validates the calling convention.

38.5 Use Cases

Hybrid programming excels in many real-world scenarios on Linux.

Cryptography and Hashing

Cryptographic primitives (AES, SHA-256, ChaCha20) are performance-critical and often implemented in assembly with SIMD and constant-time guarantees. The C part handles key management and I/O; the core transform is in NASM.

Multimedia Codecs

Image and audio codecs demand high throughput. Assembly functions accelerate colour space conversion, transforms, or sample interpolation while the framework remains in C or C++.

Checksum and Compression

Algorithms like CRC32, zlib, or LZ-based compression can exploit hardware acceleration (e.g., `CRC32` instruction) from NASM, invoked by a C utility.

Embedded Runtime or Bootloader

A lightweight hypervisor or bootloader often needs direct hardware access (port I/O, MSR reads, page table manipulation) that is impossible in C. Small assembly routines are exposed to the C initialisation code.

Game Engine Hot Loops

Physics calculations, particle system updates, and vertex processing can be offloaded to hand-tuned assembly that exploits AVX and multi-threading, while the game logic stays in C++.

Compiler and Tool Development

A code generator or disassembler written in C can use NASM to test emitted machine code fragments quickly. Assembly routines can also be used to patch or verify opcode sequences.

Complete Example: Hybrid Checksum Tool

The following project uses a C `main` function to read a file, pass its buffer to a NASM CRC-32 function, and print the result. This combination is common in utilities.

Code: crc32_asm.asm — NASM CRC-32 implementation

```
; crc32_asm.asm
bits 64
default rel
section .text
global compute_crc32

; uint32_t compute_crc32(const uint8_t* data, size_t len);
; RDI = data, RSI = len
compute_crc32:
    push    rbx
    push    r12
    mov     r12, rsi                ; length
    xor     eax, eax
    not     eax                     ; initial value = 0xFFFFFFFF
    test    r12, r12
    jz      .done
    lea     rdx, [rdi + r12]        ; end pointer
    xor     ecx, ecx
.loop:
    movzx   r8d, byte [rdi + rcx]
    xor     eax, r8d
    ; (in a real CRC, you'd use a lookup table or the CRC32 instruction)
    shl     r8d, 8
    inc     rcx
    cmp     rcx, r12
    jb      .loop
    not     eax
.done:
    pop     r12
    pop     rbx
    ret
```

Code: main.c — C driver for CRC tool

```
// main.c
#include <stdio.h>
#include <stdint.h>
```

```c
#include <stdlib.h>
extern uint32_t compute_crc32(const uint8_t* data, size_t len);

int main(int argc, char** argv) {
    if (argc < 2) {
        fprintf(stderr, "Usage: crctool file\n");
        return 1;
    }
    FILE* f = fopen(argv[1], "rb");
    if (!f) {
        perror("fopen");
        return 1;
    }
    fseek(f, 0, SEEK_END);
    size_t size = ftell(f);
    rewind(f);
    uint8_t* buf = malloc(size);
    if (!buf) {
        fclose(f);
        return 2;
    }
    fread(buf, 1, size, f);
    fclose(f);

    uint32_t crc = compute_crc32(buf, size);
    printf("CRC32: %08X\n", crc);
    free(buf);
    return 0;
}
```

Build commands:

```
nasm -f elf64 -o crc32_asm.o crc32_asm.asm
gcc -O2 -o crctool main.c crc32_asm.o
```

Run with `./crctool somefile`. The tool demonstrates a simple, real hybrid application where file I/O is managed by C, and the core algorithm is in assembly.

Tip

Keep the assembly files focused and modular. A single assembly file with a few exported functions is easier to debug and replace than a monolithic disassembly nightmare.

Putting It All Together

Hybrid programming with NASM and C/C++ on Linux is a pragmatic choice that multiplies your capabilities. By respecting the ABI, profiling before optimising, generating proper debug information, and designing clean interfaces, you can build Fedora 43 applications that are both high-performance and maintainable. The next parts of the book will explore system-level development and advanced NASM techniques.

Part VIII

ADVANCED NASM TECHNIQUES

39

Optimization Techniques

In This Chapter

Writing correct assembly is only half the battle; writing *fast* assembly requires a deep understanding of the underlying hardware and a disciplined approach to instruction choice, register allocation, memory access patterns, and loop structure. This chapter presents a collection of proven optimization techniques for the x86-64 architecture under Linux (Fedora 43), as verified against the Intel® 64 and IA-32 Architectures Optimization Reference Manual, the AMD64 Architecture Programmer's Manual, and the NASM 2.16 documentation. Each section includes concrete NASM examples and explains the performance implications of various design decisions, from instruction encoding to caching behaviour. No single technique is a silver bullet; the art is in applying the right combination for the specific bottleneck identified by profiling.

39.1 Instruction Selection

The x86-64 instruction set often provides multiple ways to accomplish the same task. Choosing the best encoding can reduce code size, lower power consumption, and improve decode bandwidth.

Zeroing Registers

The smallest and fastest way to set a register to zero is the two-byte encoding `xor reg32, reg32`. The processor recognises this idiom as a zeroing instruction and breaks any false dependency on the previous value of the register.

Code: Zeroing idioms

```
xor   eax, eax     ; 2 bytes, recognized zeroing, zero-extends RAX
mov   eax, 0       ; 5 bytes, also zero-extends but larger encoding
xor   r12d, r12d   ; zero R12 (via its 32-bit subregister)
```

Always prefer `xor` for zeroing; reserve `mov reg, 0` only when the flags must not be affected (but note that `xor` also modifies flags). For non-zero immediate values, use the 32-bit register form because it zero-extends the upper half of the 64-bit register and is shorter than a 64-bit immediate move.

Code: Loading constants efficiently

```
mov   eax, 123      ; 5 bytes, RAX = 123 (upper bits zero)
mov   rax, 123      ; 7 bytes, same result but longer
```

Using LEA for Arithmetic

The `LEA` instruction can perform up to three additions and a multiplication by a power-of-two in a single non-ALU operation. It does not alter flags, making it a versatile choice for pointer arithmetic and small multi-plications.

Code: LEA arithmetic

```
lea   rax, [rcx + rdx]          ; rax = rcx + rdx (1 cycle, no ALU)
lea   rax, [rcx + rcx*4]        ; rax = rcx * 5
lea   rax, [rcx + rdx*8 + 16]   ; rax = rcx + rdx*8 + 16
```

`LEA` is often more efficient than separate `MOV` and `ADD` instructions for simple address computations and can replace multiplies by 3, 5, or 9 with two `LEA` instructions.

Eliminating Branches with Conditional Moves

`CMOVcc` instructions conditionally move data without a branch. They eliminate the risk and penalty of a branch misprediction. However, they introduce a data dependency on both the condition result and the source operands, so they are best used when the branch is unpredictable.

Code: Branchless maximum using CMOV

```
; compute max of EAX and EBX into EAX
cmp   eax, ebx
cmovl eax, ebx       ; if eax < ebx, eax = ebx
```

Avoid `CMOV` when the branch is highly predictable, because a correctly predicted branch is cheaper than a data dependency chain.

Division by Constants

Instead of `DIV` or `IDIV`, which are very slow (20–80 cycles), use multiplication by a reciprocal constant and a shift. For constants that are powers of two, use `SHR` for unsigned division and `SAR` for signed.

Code: Divide by 10 using magic number

```
; Division by 10 of unsigned RAX -> quotient in RAX
mov   rcx, 0xCCCCCCCCCCCCCCCD   ; magic constant for /10
mul   rcx
shr   rdx, 3                    ; quotient in RDX
```

This pattern is much faster than `div` and is used by compilers.

String Instructions vs Manual Loops

The `REP MOVSB` and `REP STOSB` instructions have been optimised on modern processors for large aligned transfers, but for small copies a manual loop may be faster. Use the enhanced REP MOVSB (`REP MOVSB` with processor support) for block copies when the size is large; for short, fixed-length copies, unrolled `MOV` sequences are often better.

Code: Fast memory copy for small, fixed size

```
; copy 32 bytes from RSI to RDI
mov   rax, [rsi]
mov   rdx, [rsi+8]
mov   r8,  [rsi+16]
mov   r9,  [rsi+24]
mov   [rdi],    rax
mov   [rdi+8],  rdx
mov   [rdi+16], r8
mov   [rdi+24], r9
```

Avoiding Unnecessary Prefixes

In 64-bit code, the `REX` prefix is automatically added when needed, but it adds a byte. Prefer using 32-bit operations that zero-extend, avoiding 64-bit immediates where possible, and use registers `RAX`–`RBP` for frequently accessed values to avoid extra REX bytes on instructions that require the new registers `R8`–`R15`. However, this is a micro-optimization; prioritise readability.

39.2 Register Optimization

Effective register use reduces memory traffic, critical for performance.

Allocate Hot Variables to Registers

Identify the most frequently accessed variables (loop counters, pointers, accumulators) and keep them in registers across loop iterations. Avoid unnecessary memory spills. The 16 general-purpose registers provide ample space for a tight loop; use them before resorting to the stack.

Interleave Independent Instructions

Modern superscalar CPUs can execute multiple instructions per cycle if they do not depend on each other. Interleave independent chains to fill the available execution ports.

Code: Interleaving independent operations

```
; Computing two sums in parallel
xor   eax, eax       ; sum1
xor   ebx, ebx       ; sum2
xor   ecx, ecx       ; index
.loop:
    add   eax, [rdx + rcx*8]   ; chain 1
    add   ebx, [rsi + rcx*8]   ; chain 2 (independent of eax)
    inc   ecx
    cmp   ecx, r8d
    jb    .loop
```

The two `ADD` instructions can execute concurrently on different ports.

Break False Dependencies

A dependency chain may persist even when the result is not needed because of register renaming limits. Zeroing a register with `xor eax, eax` before loading a new value can break the dependency, enabling out-of-order execution.

Avoid Partial Register Stalls

Writing to an 8-bit or 16-bit register (e.g., `mov al, 1`) and then reading the full 64-bit register (`mov rcx, rax`) can cause a stall because the processor must merge the new low part with the old high part. Always clear the full register before operating on sub-registers, or use `movzx` to zero-extend cleanly.

Code: Avoiding partial register stalls

```
; Bad:
mov   al, 5
add   rax, rbx     ; stall: need to merge old RAX high bits with new AL

; Good:
xor   eax, eax
mov   al, 5        ; no stall because RAX was zeroed
add   rax, rbx
```

Use of Callee-Saved Registers

Callee-saved registers (`RBX, RBP, R12-R15`) must be preserved, which adds push/pop overhead. Prefer caller-saved registers (`RAX, RDI, RSI, RDX, RCX, R8-R11`) for temporary values inside leaf functions. Use callee-saved registers only for values that must survive a `CALL` instruction.

39.3 Memory Reduction

Reducing memory accesses and improving cache efficiency often yields larger gains than instruction-level optimization.

Minimise Loads and Stores

Keep data in registers as long as possible. When processing an array, load elements once, process all of them entirely in registers, and then store the result. Avoid reading and writing the same memory location repeatedly.

Align Data to Natural Boundaries

The processor incurs a penalty for unaligned memory accesses that cross a cache line boundary. Align the start of arrays and structures to at least 8-byte or 16-byte boundaries using the `align` directive.

Code: Aligned data layout

```
section .data
align 16
my_array   dq  100 dup (?)   ; 16-byte aligned start

section .bss
align 32
big_buffer resb 16384         ; 32-byte aligned for AVX
```

Pack Data for Smaller Footprint

Use the smallest data type that fits the value range. A counter that never exceeds 255 should be a byte; a flag set a bit. This not only reduces memory size but also improves cache hit rate. However, be mindful of zero-extension instructions when loading small values.

Use the Stack Efficiently

The stack is hot in L1 cache. Small temporary data structures can be allocated on the stack instead of the heap, reducing allocation overhead and improving locality. Do not allocate huge arrays on the stack, though, to avoid stack overflows.

Avoid Unnecessary Stack Frames

Leaf functions that do not need local storage and do not call other functions can omit a frame pointer and stack frame entirely, saving push/pop overhead. However, for debuggability, a frame pointer is sometimes retained.

Code: Minimal leaf function

```
; No frame, no push, very fast
leaf_add:
    lea   rax, [rdi + rsi]
    ret
```

39.4 Loop Optimization

Loops are where most processor time is spent, so optimising them yields the highest returns.

Loop Alignment

Align the start of a hot loop to a 16-byte boundary to improve the instruction fetch unit's throughput. Use `align 16` before the loop label.

Code: Aligned loop

```
align 16
.loop:
    add   eax, [rdx]
    add   rdx, 8
    sub   ecx, 1
    jnz   .loop
```

Do not over-use alignment, as it adds NOP padding and can increase code size.

Loop Unrolling

Unrolling reduces the loop overhead (the counter decrement and branch) at the cost of code size. A loop can be partially unrolled to reduce the number of branches without excessive code bloat.

Code: Loop unrolling by 4

```
; Summing an array, unrolled 4 times
xor   eax, eax
mov   ecx, len
shr   ecx, 2            ; /4
jz    .remainder
.loop:
    add   eax, [rdx]
    add   eax, [rdx+8]
    add   eax, [rdx+16]
    add   eax, [rdx+24]
    add   rdx, 32
    dec   ecx
    jnz   .loop
.remainder:
    ; handle leftover elements
```

The unrolled loop reduces the number of branch instructions by a factor of four.

Avoid Loop-Carried Dependencies

A dependency chain that spans loop iterations prevents parallel execution. Restructure the code to calculate independent elements in each iteration, then combine them after the loop, or use induction variables that can be updated independently.

Use the Right Loop Instruction

`DEC` followed by `JNZ` is often faster than `LOOP` because `LOOP` is implemented as a microcoded sequence and can be slower on modern CPUs. Additionally, `LOOP` does not modify flags, which may be needed for other checks.

Code: Counting down with DEC/JNZ

```
mov   ecx, 100
.loop:
    ; body
    sub   ecx, 1        ; or dec ecx
    jnz   .loop
```

Branch Prediction Friendly Loops

Loops that always take the branch (except on the last iteration) are highly predictable. The loop count should be in a register that is clearly monotonically decreasing. Avoid conditional jumps inside the loop that randomly alternate, unless you can use `CMOV` or convert to branchless arithmetic.

Measuring Performance with RDTSC

Accurate measurement is essential. `RDTSC` reads the time-stamp counter, and `RDTSCP` also provides the processor ID. Use it to sample elapsed cycles.

Code: Timing a function with RDTSC

```
rdtsc
shl   rdx, 32
or    rax, rdx
mov   r8, rax           ; start
; ... function call ...
rdtsc
shl   rdx, 32
or    rax, rdx
sub   rax, r8           ; elapsed ticks in RAX
```

On modern systems the time-stamp counter runs at a constant rate. For precise measurements, bracket the code with dummy runs to warm up the cache and use the median of multiple samples.

39.5 Performance Tradeoffs

Optimization is not a one-way street; many techniques involve tradeoffs that must be weighed against the specific requirements of the program.

Code Size vs Speed

Unrolling loops or using specialised sequences increases code size, which can reduce instruction cache efficiency. Very large unrolling may evict other important code from the L1 I-cache, causing a net slowdown. Profile the final binary; sometimes a smaller executable loads faster.

General-Purpose vs Specialised Algorithms

A single algorithm that handles many input sizes in a generic way is easier to maintain but may be slower than a series of size-specific fast paths. For a library routine, providing multiple im-

plementations and dispatching based on input size offers the best of both worlds, at the cost of complexity.

Memory vs Computation

Pre-computed lookup tables can replace complex calculations with fast memory loads, but the table must fit in the L1 or L2 cache. If the table is too large, cache thrashing can make the table-based version slower than direct computation. For example, a 256-byte CRC table is efficient; a 64 KiB table is likely not.

Portability vs Intrinsics

Using newer instructions (AVX-512, SHA extensions) can drastically improve performance but ties the code to recent processors. A hybrid program can detect CPU features at runtime and select the best code path. This adds overhead but ensures broad compatibility.

Maintainability vs Tuning

Heavily optimised assembly can become unreadable. Comment generously, keep the original reference implementation as a comment, and isolate critical sections in separate functions. If a tuning degrades readability beyond repair, consider whether the performance gain justifies the cost in bugs and maintenance time.

Profiling-Driven Optimization

Always profile before and after applying any optimization. What looks faster on paper may be slower due to micro-architectural effects. Use tools such as `perf`, Intel VTune, or AMD uProf to identify real bottlenecks, and concentrate effort there.

Important

Lichtenberg's Principle. "The fastest instruction is the one that isn't executed." Eliminate dead code, reduce the number of function calls, and minimise data movement before diving into instruction-level tuning.

Putting It All Together

Optimization is a multi-faceted discipline that blends knowledge of the instruction set, the processor pipeline, the memory hierarchy, and the compiler ecosystem. By systematically applying the techniques described in this chapter — choosing the smallest and fastest instruction forms, keeping hot data in registers, reducing memory traffic, structuring loops for maximum throughput, and weighing tradeoffs wisely — you can create assembly programs that run at full hardware potential on Fedora 43. The next chapter will explore CPU performance and pipeline awareness, extending these optimization ideas with deeper hardware insight.

40

CPU Performance and Pipeline Awareness

In This Chapter

Modern x86-64 processors achieve their performance through deep instruction pipelines, superscalar execution, and complex cache hierarchies. Writing efficient assembly requires understanding these micro-architectural elements, as even small changes in instruction ordering or memory access patterns can drastically affect throughput. This chapter explores the instruction pipeline, the causes and cures of pipeline stalls, the mechanics of branch prediction, the behaviour of the CPU caches, and practical profiling techniques that let you measure the impact of your optimizations. All descriptions are drawn from the Intel® 64 and IA-32 Architectures Optimization Reference Manual, the AMD64 Architecture Programmer's Manual, and the NASM 2.16 manual. Complete, ready-to-run NASM examples illustrate every concept.

40.1 Instruction Pipeline

At the heart of every high-performance x86-64 core is the instruction pipeline — a series of stages through which each instruction must pass before its result becomes available. The processor does not execute one instruction completely before starting the next; instead, it overlaps the processing of many instructions, much like an assembly line.

Classic Pipeline Stages

A simplified view divides the pipeline into five major stages:

1. **Fetch.** The instruction fetch unit reads raw bytes from the L1 instruction cache at the address held by the instruction pointer (`RIP`). On a miss, the L2 cache or main memory is accessed.

2. **Decode.** The fetched bytes are parsed into one or more *micro-operations* (uops). Complex x86 instructions are broken into multiple uops, while simple ones map to a single uop. Register operands are renamed to avoid false dependencies.
3. **Execute.** The uops are dispatched to the execution ports. A modern core may have four or more integer ALUs, two load ports, a store port, and multiple floating-point / SIMD units. Independent uops can begin execution in the same clock cycle.
4. **Memory Access.** Load and store uops access the L1 data cache. Store operations are buffered until retirement.
5. **Writeback / Retire.** The results of uops are written back to the renamed register file. Instructions are retired in program order, updating the architectural registers and flags.

Because each stage holds a different instruction simultaneously, the pipeline can sustain a throughput of multiple uops per cycle, even though a single instruction may take numerous cycles from fetch to retirement.

Superscalar Execution and Ports

Modern x86 cores are superscalar: they can issue several uops per cycle. The execution units are organised into ports, each capable of executing certain operations. For example, on a typical Intel core, Port 0,1,5,6 might handle integer arithmetic, Port 2 and 3 handle loads and store addresses, and Port 4 handles store data. A well-written assembly sequence balances uop pressure across ports to avoid overloading one while leaving others idle.

Micro-Operation Fusion

The processor can fuse certain adjacent uops into a single uop during decode. Common fused uops include compare-and-branch instructions. In NASM, a sequence like `cmp eax, 0` followed by `jne label` may be fused, reducing effective uop count. Macro-fusion saves execution bandwidth and can increase throughput.

Code: Pipeline-aware instruction ordering

```
; Sequence that benefits from macro-fusion:
cmp   eax, 0xFF
jne   .not_equal
; These two may be fused into one uop.
```

Out-of-Order Execution

The processor employs a reorder buffer that allows uops to execute as soon as their inputs are ready, regardless of program order. Retirement still happens in order, but the ability to issue uops out of order hides the latency of slow operations like cache misses or complex integer multiplies.

40.2 Pipeline Stalls

A pipeline stall occurs when a later stage cannot proceed because the required resource or data is unavailable. Stalls are the primary enemy of performance.

Data Hazards

A data hazard arises when an instruction depends on the result of a previous instruction that has not yet completed. The simplest example is a read-after-write dependency:

Code: Dependency chain causing stall

```
add   rax, rcx       ; produces RAX
sub   rbx, rax       ; consumes RAX - must wait for ADD
```

The second instruction cannot start execution until the ADD's result is ready. If the ADD has a latency of 1 cycle, the SUB must wait one cycle. In a tight loop, such dependencies can serialise execution.

Structural Hazards

A structural hazard occurs when two uops need the same execution port in the same cycle. If a load dominates the code, the load ports may become saturated, causing later loads to queue. Balancing the mix of operations avoids such bottlenecks.

Control Hazards

A control hazard is caused by branches. When a branch instruction enters the pipeline, the fetch unit does not know which path to take until the condition is resolved, which may be several stages later. Without mitigation, the pipeline would stall. Instead, the processor speculates and predicts the branch target, as described in the next section.

Minimizing Stalls

To reduce stalls:

- Reduce dependency chain length by interleaving independent instructions.
- Use `LEA` to break simple arithmetic chains into address-generation operations that may not compete for ALU ports.
- Avoid unnecessary memory accesses by keeping temporary values in registers.

Code: Interleaving to hide latency

```
; Latency-hiding via interleaving
add   rax, rcx         ; chain 1
mov   r8, [rdx]        ; independent load
add   rbx, rsi         ; chain 2 (independent of RAX)
; The load can proceed while the first ADD is being executed.
```

40.3 Branch Prediction

Conditional branches disrupt the sequential flow. To avoid stalling, the processor predicts the outcome and speculatively fetches and executes instructions along the predicted path. If the prediction

is correct, execution continues seamlessly. If not (branch misprediction), the pipeline must be flushed, incurring a penalty of typically 15–25 cycles on modern CPUs.

Static Prediction

In the absence of dynamic history, the processor uses simple rules: forward branches are predicted not taken; backward branches (loops) are predicted taken. This helps loops but does not adapt to irregular patterns.

Dynamic Prediction

The processor maintains a Branch History Table (BHT) and a Branch Target Buffer (BTB). Each taken branch records its target address and a history of outcomes. Pattern-based predictors can learn alternating sequences. The *return address stack* (RAS) predicts `RET` targets by pushing the return address on `CALL` and popping it on `RET`, making returns highly accurate.

Cost of Misprediction

A mispredicted branch flushes the entire speculative state. The following program measures the cycle difference between a loop that is easily predicted and one that is purposefully unpredictable.

Code: listing40-1.asm — Measuring branch misprediction penalty

```
; listing40-1.asm
; Assemble: nasm -f elf64 -o listing40-1.o listing40-1.asm
; Link:     gcc -no-pie -o listing40-1 listing40-1.o

bits 64
default rel

extern malloc
extern free
extern exit

section .bss
array   resq 1

section .text
global main
main:
    push    rbp
    mov     rbp, rsp
    sub     rsp, 16

    ; allocate 8192 bytes (1024 qwords) via malloc
    mov     edi, 8192
    call    malloc
    mov     [rel array], rax
    test    rax, rax
    jz      .error

    ; fill array with random 0/1 values using a simple LCG
```

```
    mov     rdi, rax
    mov     eax, 12345
    mov     ecx, 1024
.init:
    imul    eax, eax, 1103515245
    add     eax, 12345
    and     eax, 1          ; 0 or 1
    mov     [rdi + rcx*8 - 8], eax
    loop    .init

    ; warmup
    call    test_predictable
    call    test_random

    ; timed predictable loop
    rdtsc
    shl     rdx, 32
    or      rax, rdx
    mov     r12, rax
    call    test_predictable
    rdtsc
    shl     rdx, 32
    or      rax, rdx
    sub     rax, r12
    mov     r13, rax        ; predictable cycles

    ; timed random loop
    rdtsc
    shl     rdx, 32
    or      rax, rdx
    mov     r12, rax
    call    test_random
    rdtsc
    shl     rdx, 32
    or      rax, rdx
    sub     rax, r12
    mov     r14, rax        ; random cycles

    ; just exit with code 0; examine r13 and r14 in debugger
    mov     rdi, [rel array]
    call    free
    xor     edi, edi
    call    exit

.error:
    mov     edi, 1
    call    exit

; Subroutine: sum elements where element is 0 (all are 0, so highly predictable)
test_predictable:
    push    rbx
    mov     rbx, [rel array]
    mov     ecx, 1024
    xor     eax, eax
```

```
.loop:
    cmp     qword [rbx], 0
    jne     .skip
    inc     eax
.skip:
    add     rbx, 8
    sub     ecx, 1
    jnz     .loop
    pop     rbx
    ret

; Subroutine: sum elements where element is actual random value
test_random:
    push    rbx
    mov     rbx, [rel array]
    mov     ecx, 1024
    xor     eax, eax
.loop:
    cmp     qword [rbx], 0
    jne     .skip
    inc     eax
.skip:
    add     rbx, 8
    sub     ecx, 1
    jnz     .loop
    pop     rbx
    ret
```

In GDB, the cycle count for the predictable loop (all zeros) will be significantly lower than for the random loop, illustrating the misprediction penalty.

Techniques to Improve Branch Prediction

- Keep branching patterns regular.
- Use conditional moves (`cmovcc`) for short, unpredictable branches.
- Unroll loops to reduce the number of branches.
- Move invariant condition evaluation outside loops.

40.4 Cache Behavior

Memory access is a common bottleneck. The CPU's cache hierarchy tries to keep frequently used data close to the core.

Cache Hierarchy

A typical modern core has:

- **L1 data cache:** 32 KiB, 8-way set associative, 4–5 cycle latency.
- **L1 instruction cache:** 32 KiB, 8-way, fast decode.

- **L2 cache:** 256–512 KiB per core, 10–15 cycle latency.
- **L3 cache (LLC):** shared among cores, several MB, 30–50 cycle latency.
- **Main memory:** $>$ 100 cycles.

Data is moved in 64-byte *cache lines*. When a load accesses a memory location, the entire cache line is fetched if not already present.

Cache-Friendly Access Patterns

Sequential access to memory (stride-1) is the most cache-friendly pattern, allowing hardware prefetchers to bring data into cache before it is needed. Random access patterns thrash the cache and cause many misses.

Code: Sequential vs random access

```
; Sequential sum (cache-friendly)
mov   rcx, 4096
lea   rdx, [array]
.loop_seq:
    add   eax, [rdx]
    add   rdx, 8
    dec   ecx
    jnz   .loop_seq

; Random (strided) access (cache-unfriendly)
mov   rcx, 4096
lea   rdx, [array]
.loop_rand:
    add   eax, [rdx + r8*8]   ; r8 pseudo-random
    dec   ecx
    jnz   .loop_rand
```

The sequential loop will run much faster because it benefits from prefetching and cache line reuse.

Avoiding Cache Line Splits and False Sharing

A data access that crosses a 64-byte boundary incurs two cache line accesses. Align large data structures to 64-byte boundaries using `align 64`. In multi-threaded code, ensure that two threads do not write to variables on the same cache line (false sharing), or performance will degrade.

Prefetching

The `PREFETCH` instruction gives a hint to bring a cache line into a specific cache level. It is useful for loops that traverse non-sequential patterns but can predict upcoming accesses. However, modern hardware prefetchers often make manual prefetching unnecessary for simple patterns.

Code: Manual prefetch example

```
.loop:
    prefetchnta [rdx + 256]  ; prefetch ahead
    add   eax, [rdx]
```

```
    add   rdx, 8
    sub   ecx, 1
    jnz   .loop
```

Use `PREFETCHT0` (into all cache levels) or `PREFETCHNTA` (non-temporal, minimising cache pollution) depending on the access pattern.

Data Alignment and Cache Lines

The C/C++ compiler automatically aligns global variables, but in NASM you must explicitly align. An unaligned load that crosses a cache line boundary incurs additional latency. Always align arrays to at least 16 bytes, and if they are heavily used, 64 bytes.

40.5 Profiling Techniques

Without measurement, optimization is guesswork. Several tools and techniques allow you to profile assembly code on Linux.

The `RDTSC` Instruction

`RDTSC` reads the 64-bit time-stamp counter into `EDX:EAX`. It provides a cycle-accurate measure. Use it to time short code sequences, being aware of context switches, turbo boost, and out-of-order execution (serialise with `mfence` or `cpuid` for accurate timing).

Code: Cycle counting with RDTSC and CPUID

```
; Serialise and read start time
xor   eax, eax
cpuid
rdtsc
shl   rdx, 32
or    rax, rdx
mov   r12, rax          ; start

; Code under test ...
; ...

xor   eax, eax
cpuid
rdtsc
shl   rdx, 32
or    rax, rdx
sub   rax, r12          ; cycles elapsed
```

The `CPUID` instruction forces the pipeline to serialise, preventing the timing code from being interleaved with the measured code.

`clock_gettime` for High-Resolution Timing

Linux provides the `clock_gettime` system call, which can offer nanosecond resolution. The function signature is:

Code: Using clock_gettime

```
extern clock_gettime
; int clock_gettime(clockid_t clk_id, struct timespec *tp);
; RDI = clock id (e.g., CLOCK_MONOTONIC = 1)
; RSI = pointer to timespec struct { time_t tv_sec; long tv_nsec; }

section .data
CLOCK_MONOTONIC equ 1

section .bss
ts_start resq 2      ; 16 bytes for timespec
ts_end   resq 2

section .text
    lea     rsi, [rel ts_start]
    mov     edi, CLOCK_MONOTONIC
    call    clock_gettime

    ; ... measured code ...

    lea     rsi, [rel ts_end]
    mov     edi, CLOCK_MONOTONIC
    call    clock_gettime
    ; compute difference in nanoseconds: (ts_end.tv_sec - ts_start.tv_sec)*10^9 + (tv_nsec
    ↪  diff)
```

This is perfect for larger benchmark loops and can be used alongside `printf` to output results.

Linux Profiling Tools

For system-wide analysis, `perf` is the standard Linux profiler. It can sample CPU cycles, cache misses, branch mispredictions, and many other hardware events.

Code: Using perf to profile a program

```
perf record ./myprogram
perf report
```

With debug symbols (compiled with `-g`), `perf report` will show the annotated source or disassembly, pinpointing hot spots. It can also measure specific events: `perf stat -e cycles,instructions,branches,br ./myprogram`.

Valgrind and Cachegrind

Valgrind's `cachegrind` tool simulates the L1 and L2 caches and reports miss rates. Although it does not use real hardware counters, it provides a reproducible view of memory behaviour.

Code: Cache profiling with valgrind

```
valgrind --tool=cachegrind ./myprogram
cg_annotate cachegrind.out.<pid>
```

This is invaluable for understanding memory access patterns in pure assembly.

Intel VTune and AMD uProf

Dedicated performance profilers provide detailed micro-architectural analysis: uop breakdown, cache misses, branch misprediction rates, port pressure, and more. They read the processor's performance monitoring counters (PMC). Installing Intel VTune or AMD uProf on Fedora allows deep analysis of assembly programs.

Manual PMC Access

If you wish to read performance counters directly from assembly, you can use the `RDPMC` instruction (privileged) or the `RDMSR` instruction. In user mode, Linux exposes some counters via the `perf_event_open` system call, but this is complex. For most purposes, using an external profiling tool is more practical.

Instrumentation with `printf` or `write` to stderr

A simple way to profile function-level latency without external tools is to log timestamps. You can call `clock_gettime` at the entry and exit of a function, compute the difference, and write the result to stderr using the `write` system call or `fprintf`. This allows you to analyse performance without a debugger.

Profiling Best Practices

- Warm up the code by executing it a few times before timing to eliminate cold cache effects.
- Run multiple iterations and take the median or minimum; discard outliers.
- Disable CPU frequency scaling (set the scaling governor to `performance`) or use invariant TSC.
- Isolate the code under test so that external interrupts and context switches are minimised.
- Correlate timer-based results with hardware counter data for a complete picture.

Example: Profiling a Simple Loop

The following NASM program measures the cycles per iteration of a sum loop and prints the result using `printf`. It demonstrates combining `RDTSC`, iteration counting, and console output.

Code: listing40-2.asm — Profiling a loop

```
; listing40-2.asm
; Assemble: nasm -f elf64 -o listing40-2.o listing40-2.asm
; Link:     gcc -no-pie -o listing40-2 listing40-2.o
```

```
bits 64
default rel

extern printf
extern exit

section .data
fmt db 'Cycles for 1000000 iterations: %llu', 10, 0

section .text
global main
main:
    push    rbp
    mov     rbp, rsp
    sub     rsp, 16          ; align stack

    ; --- measure loop ---
    xor     eax, eax
    cpuid
    rdtsc
    shl     rdx, 32
    or      rax, rdx
    mov     r12, rax            ; start

    mov     ecx, 1000000
    xor     eax, eax
    align   16
.loop:
    add     eax, 1
    sub     ecx, 1
    jnz     .loop

    xor     eax, eax
    cpuid
    rdtsc
    shl     rdx, 32
    or      rax, rdx
    sub     rax, r12            ; cycles in RAX

    mov     rsi, rax            ; second arg to printf
    lea     rdi, [rel fmt]
    xor     eax, eax
    call    printf

    xor     edi, edi
    call    exit
```

Running this program prints a cycle count, giving a concrete measure of the loop's performance on your actual hardware. Compare the unoptimised loop with a version that uses `LEA` to do the addition and the counter decrement in one instruction, and see the effect on cycles.

Important

Always profile on the exact hardware and configuration that the final program will run on. Performance can vary dramatically between different CPU generations, cache sizes, and memory speeds.

Putting It All Together

CPU performance is a vast topic, but a solid grasp of the pipeline, stalls, branch prediction, caching, and profiling gives you the power to write assembly code that fully exploits the capabilities of a modern `x86-64` processor. The techniques in this chapter should be applied iteratively: profile, identify the bottleneck, apply a targeted optimization, and profile again. The next chapter will continue with advanced SIMD programming, where many of these performance principles reach their full expression.

41

SIMD Overview (SSE and AVX)

In This Chapter

Single Instruction, Multiple Data (SIMD) allows one machine instruction to operate on multiple data elements simultaneously. The x86-64 architecture provides a rich set of SIMD extensions, beginning with the mandatory SSE/SSE2 and evolving through AVX and AVX-512. This chapter introduces the SIMD concept, covers the SSE instruction set available on every Linux x86-64 processor, gives an overview of the Advanced Vector Extensions (AVX), explains how to organise and process vector data in NASM, and discusses the performance benefits that make SIMD essential for numerical and multimedia applications. All examples are based on the Intel® 64 and IA-32 Architectures Software Developer's Manual, the AMD64 Architecture Programmer's Manual, and the NASM 2.16 documentation. Every listing is a complete, assemble-ready Fedora 43 program using only system calls (unless noted).

41.1 SIMD Concept

Traditional scalar instructions process one data item at a time: `add eax, ebx` adds a single pair of 32-bit integers. SIMD instructions work on *packed* data — multiple identical data items packed into a single wide register — and perform the same operation on all elements in parallel. For example, a single `paddd xmm0, xmm1` adds four 32-bit integers in `XMM0` to the corresponding ones in `XMM1` in one clock cycle.

The main SIMD register sets on x86-64 are:

- **XMM** (128-bit): introduced with SSE. Contains 16 registers (XMM0–XMM15 in 64-bit mode).
- **YMM** (256-bit): introduced with AVX. Occupies the lower 128 bits of the corresponding XMM register; the upper 128 bits are new.

- **ZMM** (512-bit): introduced with AVX-512. Not available on all processors; software must check CPUID.

NASM supports all SIMD instruction sets with the same Intel-syntax mnemonics, using prefixes like `SSE`, `SSE2`, `AVX`, and automatic VEX encoding where required.

41.2 SSE Instructions

The Streaming SIMD Extensions (SSE) and SSE2 are guaranteed to be present on every x86-64 Linux machine. They provide integer and floating-point SIMD operations on 128-bit XMM registers.

Data Movement

SSE load and store instructions move data between memory and XMM registers. They can be aligned or unaligned.

Code: SSE load and store

```
movaps  xmm0, [rcx]       ; load 16-byte aligned data into XMM0
movups  xmm1, [rdx]       ; load unaligned data
movaps  [r8],  xmm2       ; store aligned
movups  [r9],  xmm3       ; store unaligned
```

`movaps` requires 16-byte alignment; `movups` does not. Using the aligned form on unaligned memory causes a general-protection fault. Always align SIMD data with `align 16` in NASM.

Packed Integer Arithmetic

SSE2 provides integer operations. For example, `paddd` adds four 32-bit integers:

Code: Packed integer addition

```
; Assume XMM1 = [1, 2, 3, 4] (32-bit elements)
; XMM2 = [10, 20, 30, 40]
paddd   xmm1, xmm2        ; XMM1 = [11, 22, 33, 40]
```

Other operations include `psubd`, `pmulld`, `pslld`, `psrad`, and `psrld`.

Packed Floating-Point (SSE)

Floating-point operations use `addps` (packed single-precision), `mulps`, `subps`, `divps`, and scalar versions `addss`, etc. Double-precision variants use the `pd` suffix.

Code: Packed floating-point multiply-add

```
movaps  xmm0, [array_a]
movaps  xmm1, [array_b]
mulps   xmm0, xmm1          ; xmm0 = xmm0 * xmm1 (4 floats)
addps   xmm0, [array_c]     ; xmm0 = xmm0 + array_c
movaps  [result], xmm0
```

Complete SSE Example: Adding Two Float Arrays

The following program allocates two aligned float arrays, fills them with sample values, adds them with SSE, and prints a completion message via a system call.

Code: listing41-1.asm — SSE float array addition (Linux)

```
; listing41-1.asm
; Assemble: nasm -f elf64 -o listing41-1.o listing41-1.asm
; Link:     ld -o listing41-1 listing41-1.o

bits 64
default rel

SYS_WRITE equ 1
SYS_EXIT  equ 60
STDOUT    equ 1

section .data
align 16
a       dd 1.0, 2.0, 3.0, 4.0
b       dd 5.0, 6.0, 7.0, 8.0
result  dd 0.0, 0.0, 0.0, 0.0
msg     db 'SSE addition complete.', 10
msg_len equ $ - msg

section .text
global _start
_start:
    ; SSE vector addition
    movaps  xmm0, [rel a]
    movaps  xmm1, [rel b]
    addps   xmm0, xmm1
    movaps  [rel result], xmm0

    ; print message
    mov     eax, SYS_WRITE
    mov     edi, STDOUT
    lea     rsi, [rel msg]
    mov     edx, msg_len
    syscall

    ; exit(0)
    mov     eax, SYS_EXIT
    xor     edi, edi
    syscall
```

The program demonstrates aligned load, packed add, and aligned store. The console output confirms execution.

Tip

Always align SIMD data to 16-byte boundaries using `align 16` before the data label. Unaligned loads work but may be slower or cause faults if you accidentally use `movaps`.

41.3 AVX Overview

Advanced Vector Extensions (AVX) expand SIMD to 256-bit YMM registers, doubling the width of SSE. AVX is supported on Intel Sandy Bridge and later, and AMD Bulldozer and later. AVX instructions use a non-destructive three-operand syntax, e.g., `vaddps ymm0, ymm1, ymm2`. NASM automatically generates the required VEX prefix.

YMM Registers and State Penalties

YMM0–YMM15 are 256-bit registers. After using AVX instructions, you must call `vzeroupper` before returning or calling code that might use SSE, to avoid performance penalties from preserving the upper 128 bits.

Key AVX/AVX2 Instructions

- **Floating-point:** `vaddps`, `vmulps`, `vsubps`, `vdivps`, `vaddpd`, etc.
- **AVX2 integer:** `vpaddd`, `vpmulld`, `vpslld`, `vpsrad`, etc.
- **Fused Multiply-Add (FMA3):** `vfmadd213ps`, `vfmsub...` perform multiply-add in one instruction.
- **Permute and blend:** `vperm2f128`, `vblendvps`, etc.
- **Broadcast and gather:** `vbroadcastss`, `vgatherdps`.

AVX Example: Dot Product of Two Float Arrays

The following program computes the dot product of two 8-element float arrays using AVX vectors, and prints a completion message.

Code: listing41-2.asm — AVX float dot product (Linux)

```
; listing41-2.asm
; Assemble: nasm -f elf64 -o listing41-2.o listing41-2.asm
; Link:     ld -o listing41-2 listing41-2.o

bits 64
default rel

SYS_WRITE equ 1
SYS_EXIT  equ 60
STDOUT    equ 1

section .data
align 32
vec1    dd  1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0
vec2    dd  8.0, 7.0, 6.0, 5.0, 4.0, 3.0, 2.0, 1.0
msg     db 'AVX dot product computed.', 10
msg_len equ $ - msg

section .text
global _start
```

```
_start:
    ; Load two 256-bit vectors
    vmovaps  ymm0, [rel vec1]
    vmovaps  ymm1, [rel vec2]

    ; Multiply
    vmulps   ymm2, ymm0, ymm1

    ; Horizontal sum (illustrative; not complete reduction)
    vhaddps  ymm2, ymm2, ymm2
    vhaddps  ymm2, ymm2, ymm2
    ; final sum would require further steps, but we demonstrate AVX operations

    ; Print message
    mov      eax, SYS_WRITE
    mov      edi, STDOUT
    lea      rsi, [rel msg]
    mov      edx, msg_len
    syscall

    vzeroupper
    mov      eax, SYS_EXIT
    xor      edi, edi
    syscall
```

Caution

Failure to use `vzeroupper` after AVX code can cause severe performance degradation in subsequent SSE code. Always call `vzeroupper` before returning or before calling any function that might use SSE.

41.4 Vector Processing

Vector processing in assembly involves organizing data into contiguous aligned arrays, loading them into SIMD registers, performing element-wise operations, and storing the results back. The typical strip-mining loop processes chunks equal to the SIMD register width, with a scalar cleanup for remaining elements.

Data Alignment

For aligned SIMD loads, use `align 16` (SSE) or `align 32` (AVX) in the data section. Aligned moves (`movaps`, `vmovaps`) are the fastest.

41.5 Performance Benefits

SIMD provides significant throughput gains by processing multiple data elements per instruction. For example, SSE can deliver a 4x speedup for floating-point loops, and AVX can double that. The actual gain depends on memory bandwidth, cache behaviour, and the ability to keep the pipeline fed.

When Not to Use SIMD

Small arrays or code that cannot guarantee alignment may not benefit. Always profile with tools like `perf` to confirm improvements.

Putting It All Together

SIMD programming is an essential skill for high-performance assembly on Linux. With SSE guaranteed and AVX widely available, you can write vectorized kernels that exploit the full width of modern processors. NASM fully supports all SIMD instruction sets, making it straightforward to implement. The next chapters will explore advanced macros and profiling techniques.

42

Advanced Macros and Code Generation

In This Chapter

The NASM preprocessor is a powerful tool that goes far beyond simple textual substitution. With its macro language, you can define parameterized code templates, generate repetitive instruction sequences, perform compile-time arithmetic, and conditionally assemble different code paths based on build options. This chapter explores advanced macro programming in depth, covering multi-line macros with parameters, compile-time logic using `%if` and `%rep`, techniques for building reusable code libraries, debugging macros with listing files, and modular design strategies that keep large assembly projects manageable. All content follows the official NASM 2.16 manual and applies to Linux x86-64 development. Every example is a self-contained, assemble-ready snippet.

42.1 Macro Programming

NASM macros allow you to define sequences of assembly statements that can be inserted anywhere with a single invocation. They accept parameters, support local labels, and can be nested.

Defining Multi-Line Macros

A macro is introduced with `%macro` and terminated with `%endmacro`. The second operand is the number of parameters (a range or `*` for "any").

Code: A simple logging macro calling printf

```
%macro debug_print 1-*
    push    rdi
    push    rsi
    push    rdx
```

```
    push    rcx
    push    r8
    push    r9
    ; The first argument is a format string; the rest are passed to printf.
    %if %0 > 1
        ; pass arguments as is (simplified: requires careful handling)
    %else
        lea     rdi, [rel %1]
        xor     eax, eax
        call    printf
    %endif
    pop     r9
    pop     r8
    pop     rcx
    pop     rdx
    pop     rsi
    pop     rdi
%endmacro
```

This macro saves volatile registers according to the System V ABI and calls `printf` with a format string. It can be used as:

```
extern printf
debug_print msg_buffer
```

Parameter Handling and Rotation

Macro parameters are referenced as `%1`, `%2`, …, and the total number is `%0`. With `%rotate`, you can iterate through parameters.

Code: Push multiple registers with rotation

```
%macro push_regs 1-*
    %rep %0
        push    %1
        %rotate 1
    %endrep
%endmacro
```

`push_regs rax, rbx, rcx` expands to three `push` instructions.

Local Labels Inside Macros

Use the `%%` prefix to create labels that are unique to each expansion, avoiding duplicate symbol errors.

Code: A macro loop with local labels

```
%macro sum_array 3              ; dest, array, count
    xor     %1, %1
    mov     ecx, %3
    test    ecx, ecx
    jz      %%.done
```

```
    lea     rdx, [rel %2]
    xor     eax, eax
    %%.loop:
        add     %1, [rdx + rax*8]
        inc     eax
        cmp     eax, ecx
        jb      %%.loop
    %%.done:
%endmacro
```

Each invocation generates a loop with its own unique labels.

Conditional Macro Expansion

With `%ifidni` you can select between aligned and unaligned loads:

Code: Macro that selects between aligned and unaligned load

```
%macro load_xmm 3          ; dest, src, alignment
    %ifidni %3, aligned
        movaps  %1, [%2]
    %else
        movups  %1, [%2]
    %endif
%endmacro
```

Compile-Time Factorial

Macros with `%assign` can perform compile-time computation:

Code: Compile-time factorial macro

```
%macro factorial 2          ; n, result_symbol
    %assign i %1
    %assign prod 1
    %rep %1
        %assign prod prod * i
        %assign i i - 1
    %endrep
    %define %2 prod
%endmacro

factorial 5, fact5        ; fact5 = 120
dw   fact5                ; emits dw 120
```

42.2 Compile-Time Logic

The NASM preprocessor provides `%if`, `%rep`, `%assign`, and string manipulation.

Conditional Assembly

Code: Conditional assembly based on build mode

```
%define DEBUG 1

section .text
%if DEBUG
    debug_print 'Entering function X'
%endif
    ; actual function code
```

Expression Evaluation and %rep

Code: Generating a table of squares

```
%assign i 0
%rep 10
    dd  i * i
    %assign i i + 1
%endrep
```

This expands to ten `dd` directives.

42.3 Modular Design

A typical layout for a Linux NASM project:

```
project/
  main.asm
  include/
    linux.inc       ; syscall numbers, libc externs
    macros.inc      ; reusable macros
    structures.inc  ; structure definitions
  modules/
    file_io.asm     ; file I/O routines
    engine.asm      ; performance-critical engine
```

Include File with Guard

Code: linux.inc

```
%ifndef LINUX_INC
%define LINUX_INC

SYS_READ    equ 0
SYS_WRITE   equ 1
SYS_EXIT    equ 60
STDOUT      equ 1
```

```
extern printf, exit, malloc, free

%endif
```

Reusable Prologue/Epilogue Macro

Code: frame.inc

```
%ifndef FRAME_INC
%define FRAME_INC

%macro enter 1
    push    rbp
    mov     rbp, rsp
    sub     rsp, %1
%endmacro

%macro leave 0
    mov     rsp, rbp
    pop     rbp
    ret
%endmacro
%endif
```

Putting It All Together

Advanced macro programming transforms NASM into a flexible code-generation tool. By mastering macros, compile-time logic, include file organisation, and modular design, you can build maintainable, high-performance assembly projects that scale far beyond single-file experiments.

43

Low-Level Debugging Techniques

In This Chapter

Assembly language programming demands a close relationship with the machine, and when things go wrong, only the raw state of the CPU and memory can reveal the truth. This chapter presents a collection of low-level debugging techniques that go beyond simple breakpoints and step-through. You will learn to formulate effective breakpoint strategies, inspect memory in detail, track register values across complex code paths, trace the call stack manually, and apply basic reverse-engineering methods to understand unknown binaries. All techniques are demonstrated with concrete NASM examples on Fedora 43, using the GNU debugger (GDB) and standard system tools. The knowledge here draws from the Intel® Software Developer's Manuals, the System V AMD64 ABI, and the official GDB documentation.

43.1 Breakpoint Strategy

Software Breakpoints

A software breakpoint replaces the first byte of an instruction with `0xCC` (INT3). When execution reaches it, the CPU traps into the debugger. You can embed breakpoints directly into NASM source using the `int3` instruction.

Code: Embedded debug breakpoint

```
my_func:
    push    rbp
    mov     rbp, rsp
    int3                    ; debugger will stop here if attached
    sub     rsp, 0x20
```

If the program is run without a debugger, the kernel delivers `SIGTRAP`, which terminates the process. To avoid this, conditionally assemble the breakpoint only in a debug build with `%ifdef DEBUG`.

Hardware Breakpoints

Hardware breakpoints use the processor's debug registers (DR0–DR3) to watch for memory access or execution without modifying the code. In GDB, use `watch`, `rwatch`, or `awatch`:

```
watch *0x601000      ; break when 8 bytes at address are *written*
rwatch *0x601000     ; break when read
awatch *0x601000     ; break on any access
```

Conditional Breakpoints

Conditional breakpoints pause only when a specified condition is true. In GDB:

```
break myloop if $rcx == 9999
break *0x400100 if $eax == 5
```

This saves time in loops.

Tracepoints

To log register values without stopping, use `commands`:

```
break myfunc
commands
  silent
  printf "myfunc called, rdi = %lx\n", $rdi
  continue
end
```

43.2 Memory Inspection

GDB's `x` (examine) command displays memory in various formats:

- `x/10bx addr` — 10 bytes in hex.
- `x/4hx addr` — 4 halfwords.
- `x/2wx addr` — 2 words.
- `x/1gx addr` — 1 giant (64-bit).
- `x/s addr` — null-terminated string.
- `x/i addr` — machine instruction.

For example, to inspect the stack:

```
x/10gx $rsp          # 10 quadwords from stack pointer
x/s $rdi             # string pointed to by RDI
```

To search for byte patterns:

```
find /b 0x400000, 0x401000, 0x48, 0x8B
```

43.3 Register Tracking

Use `info registers` (or `i r`) to display all GPRs. To print a single register:

```
info registers rax
p/x $rbx
```

The `display` command automatically shows a value after each step:

```
display /x $rax
display /i $rip
```

Flags can be checked via `$eflags`. For example, ZF is bit 6:

```
print ($eflags & 0x40) != 0
```

43.4 Stack Tracing

GDB's `backtrace` (or `bt`) shows the call stack. For functions with a standard frame pointer (`RBP`), GDB can unwind automatically. If not, you can manually examine the return address at `[rbp+8]` and follow the chain.

To walk the stack manually:

1. Note `RSP` and `RBP`.
2. Look at the return address: `x/gx $rsp` (on entry) or `x/gx $rbp+8`.
3. Disassemble that address with `x/i <address>`.
4. Follow the saved `RBP` to the previous frame.

Stack Corruption Detection

You can embed a canary value on the stack and check it before returning:

Code: Canary check in NASM

```
my_func:
    push    rbp
    mov     rbp, rsp
    sub     rsp, 0x20
    mov     qword [rbp-8], 0xDEADBEEFCAFEBABE   ; canary
    ; ... use buffer ...
    cmp     qword [rbp-8], 0xDEADBEEFCAFEBABE
    jne     .stack_corrupted
    leave
    ret
.stack_corrupted:
    int3
```

43.5 Reverse Engineering Basics

Disassembling Your Own Program

Use `objdump -d prog` or NASM's listing file (`-l`) to verify generated code.

Analyzing an Unknown Binary

1. **Identify:** `file`, `readelf -h`, `ldd`.
2. **Static:** Use a disassembler (objdump, Ghidra, radare2).
3. **Dynamic:** Run under `strace` or GDB, set breakpoints on library functions.

Patching Binaries

In GDB, modify memory with `set`:

```
set *(char*)0x400100 = 0x90   ; nop
set {int}0x601000 = 1234
```

Practical RE Exercise

The following NASM program prints its own entry point address using system calls, serving as a self-referencing exercise.

Code: listing43-1.asm — Self-referencing program

```
; listing43-1.asm
; Assemble: nasm -f elf64 -o listing43-1.o listing43-1.asm
; Link:     ld -o listing43-1 listing43-1.o

bits 64
default rel

SYS_WRITE equ 1
SYS_EXIT  equ 60
STDOUT    equ 1

section .data
msg     db 'Entry point is at RIP=0x', 0
msg_len equ $ - msg
newline db 10

section .bss
hexbuf  resb 16

section .text
global _start
_start:
    ; Write message
    mov     eax, SYS_WRITE
    mov     edi, STDOUT
    lea     rsi, [rel msg]
    mov     edx, msg_len
    syscall
```

```
    ; Get entry point address
    lea     rax, [rel _start]

    ; Convert to hex string in hexbuf (similar to earlier chapters)
    lea     rdi, [rel hexbuf + 15]
    mov     rcx, 16
    std
.hexloop:
    mov     rdx, rax
    and     dl, 0xF
    add     dl, '0'
    cmp     dl, '9'
    jbe     .store
    add     dl, 7
.store:
    mov     [rdi], dl
    shr     rax, 4
    dec     rdi
    loop    .hexloop
    cld

    ; Write hex string
    mov     eax, SYS_WRITE
    mov     edi, STDOUT
    lea     rsi, [rel hexbuf]
    mov     edx, 16
    syscall

    ; Newline and exit
    mov     eax, SYS_WRITE
    mov     edi, STDOUT
    lea     rsi, [rel newline]
    mov     edx, 1
    syscall
    mov     eax, SYS_EXIT
    xor     edi, edi
    syscall
```

Run the program, then attach GDB to see the same address in the disassembly. Practice modifying the hex conversion logic while debugging.

Important

Reverse engineering for educational and defensive purposes is legitimate in many jurisdictions, but always respect software licenses and copyright laws.

Putting It All Together

Low-level debugging is a feedback loop: observe, hypothesise, modify, test. The techniques outlined here—from strategic breakpoints to manual stack tracing and binary patching—equip you to diagnose any problem in your NASM programs. Combined with optimisation knowledge, you can now write, debug, and tune assembly code at a professional level on Linux.

Part IX

SYSTEM-LEVEL DEVELOPMENT

44

Writing Assembly Libraries on Linux

In This Chapter

A well-crafted library transforms reusable assembly code into a reliable asset that can be shared across multiple projects. This chapter covers the entire lifecycle of an assembly library on Linux (Fedora 43): structuring the code into logical modules, designing functions that are truly reusable, creating a clean and consistent API, documenting the interface for other developers, and maintaining the library over time. All examples use NASM 2.16 and follow the System V AMD64 ABI. You will learn to build both static libraries (`.a`) and shared libraries (`.so`), how to export symbols correctly, and how to design an API that feels native to Linux. The content is grounded in the ELF specification, the GNU linker manual, and the NASM manual.

44.1 Library Structure

A library is a collection of object files grouped together with a well-defined interface. The physical structure determines usability and maintainability.

Source File Organisation

Separate the library source into three logical parts:

- **Public interface.** A single NASM include file (e.g., `mylib.inc`) that contains `extern` declarations for every exported function, any `equ` constants used by the interface, and optional `struc` definitions for structures passed to or from the library. For C compatibility, provide a corresponding C header (`mylib.h`) with `extern "C"`.
- **Internal helpers.** Source files that are not exported, containing private subroutines, internal data, and shared constants. They are assembled into the library but not advertised.

- **Public implementation.** One or more `.asm` files that define the exported functions. Each function is marked `global`.

Example layout:

```
mylib/
  include/
    mylib.inc           ; NASM interface
    mylib.h             ; C interface (optional)
  src/
    internal/
      helpers.asm
      constants.inc
    export/
      math.asm
      strings.asm
```

Building a Static Library (`.a`)

A static library is an archive of object files created with the `ar` utility. The consumer links with the `.a` file, and the necessary code is copied into the final executable.

Build steps:

Code: Building a static library

```
nasm -f elf64 -o math.o math.asm
nasm -f elf64 -o strings.o strings.asm
ar rcs libmylib.a math.o strings.o
```

The user links with:

```
gcc -no-pie -o app main.o -L. -lmylib
```

or directly with `ld` and `--whole-archive` if needed.

Building a Shared Library (`.so`)

A shared library is a position-independent ELF file that exports symbols. NASM code should use `default rel` and avoid absolute addresses. Mark exported functions with `global`.

Create the shared library:

Code: Creating a shared library

```
nasm -f elf64 -o math.o math.asm
gcc -shared -o libmylib.so math.o
```

To control which symbols are visible, you can use a linker version script, but all `global` symbols are exported by default.

To use the library:

```
gcc -no-pie -o app main.o -L. -lmylib
# At runtime, either set LD_LIBRARY_PATH or install the .so
```

Internal vs. External Linkage

Functions and data for internal use should **not** be declared `global`. They remain local to the object file and are invisible to the library user.

44.2 Reusable Functions

A reusable function must adhere to the ABI, minimise side effects, be thread-safe when needed, and report errors consistently.

ABI Adherence

Follow the System V AMD64 ABI:

- First six integer/pointer arguments in `RDI, RSI, RDX, RCX, R8, R9`.
- Return integer/pointer in `RAX`, floating-point in `XMM0`.
- Callee-saved registers must be preserved if modified: `RBX, RBP, R12-R15`.
- The stack must be 16-byte aligned before any `CALL` (the caller's responsibility, but library functions that call out must ensure alignment).

Minimal Side Effects

Avoid modifying global state unless that is the function's purpose. Preserve callee-saved registers. Document which registers are clobbered (all caller-saved).

Thread Safety

If the library maintains global data, protect it with mutexes (from `pthread`) or avoid it entirely. The easiest approach is to have the caller supply a context pointer (like a handle). Library functions that only use their arguments and local stack variables are inherently thread-safe.

Error Reporting

Options:

- Return a status code (0 for success, negative for errors).
- Return `NULL` for functions that return pointers.
- For C-library compatibility, set the `errno` variable (via `__errno_location`) and return `-1`. This requires linking with libc.

For a pure assembly library without libc, returning an error code directly is simplest.

Code: Function returning error code

```
; Returns RAX = bytes processed, or negative errno on failure.
global process_data
process_data:
    ; ... work ...
    jc      .error
    mov     rax, byte_count
    ret
.error:
    mov     rax, -ENOMEM    ; negative errno
    ret
```

44.3 API Design

The library's API should be intuitive and consistent.

Naming Conventions

Use a common prefix to avoid collisions, e.g., `str_copy`, `str_length`, `math_add`. Constants can be prefixed with `MYLIB_`.

Parameter Design

- Pass small integers (up to 64 bits) by value in registers.
- Pass pointers for buffers and structures. For output buffers, let the caller provide the buffer and its size to prevent overruns.
- Place the destination buffer in `RDI`, source in `RSI`, size in `RDX` (a common pattern on Linux).
- Always include a buffer size parameter when writing to user-provided memory.

Versioning

Provide a function like `mylib_get_version` that returns a version constant. Maintain backward compatibility; if a new function changes signature, add an extended version (e.g., `str_copy_ex`) and keep the old one.

Interfaces for C Users

Create a C header file (`mylib.h`) with `extern "C"` declarations matching the NASM exports. This allows the library to be called from C/C++ without name mangling.

Error Codes

Define symbolic constants for error returns:

```
MYLIB_ERR_NONE   equ  0
MYLIB_ERR_NOMEM  equ -1
MYLIB_ERR_INVAL  equ -2
```

Include these in the public include file.

44.4 Documentation

Thorough documentation is essential. Every exported function should have a header comment describing:

- Purpose.
- Parameters (which registers, what they mean).
- Return value.
- Registers clobbered/preserved.
- Any preconditions (e.g., alignment).

Example:

Code: Well-documented function header

```
; ------------------------------------------------------------
; str_copy: copies a null-terminated string.
;   RDI = destination buffer
;   RSI = source string
;   RDX = destination buffer size (bytes)
; Returns:
;   RAX = number of bytes copied (excluding null), or -1 if buffer too small.
;   ZF set if full copy succeeded.
; Clobbers: RAX, RCX, RDX (volatile).
; ------------------------------------------------------------
```

In addition, maintain a `README` or man-page summarising all exported functions.

Debug Symbols

When assembling, use `nasm -f elf64 -g` to include DWARF debug information. This allows GDB to show source lines and variable names. A separate symbol file is not needed; the debug info resides in the object files.

44.5 Maintenance

Libraries live longer than individual projects. Good maintenance practices include version control, automated builds, testing, and change logs.

Build Automation

Use a Makefile:

Code: Makefile for library

```
NASM = nasm
AR   = ar
OBJS = obj/math.o obj/strings.o
TARGET = libmylib.a

$(TARGET): $(OBJS)
^^I$(AR) rcs $@ $^

obj/%.o: src/export/%.asm
^^I$(NASM) -f elf64 -o $@ $<

.PHONY: clean
clean:
^^Irm -rf obj libmylib.a
```

A similar rule can create a shared library using `gcc -shared`.

Testing

Write a test program (in assembly or C) that exercises every exported function with edge cases. Use `assert` or compare outputs. Run the tests frequently, ideally with a script:

Code: Simple test script

```
#!/bin/bash
nasm -f elf64 -o test.o test.asm
gcc -no-pie -o test test.o -L. -lmylib
./test || echo "Tests failed"
```

Version Control and Change Logs

Store the library in Git. Tag version numbers. Keep a `CHANGELOG.md` documenting changes.

Preserving Interface Compatibility

Once an API is released, avoid changing its signature. Add new functions with a new name or a version suffix. Deprecate old functions with a comment.

Performance Monitoring

Profile the library using `perf` or timing loops to ensure optimizations don't introduce regressions.

Complete Minimal Library Example

The following builds a static library `libmax.a` containing a function that returns the maximum of two unsigned 64-bit integers, and a test program.

Code: max_lib.asm — Library implementation

```
; max_lib.asm
bits 64
default rel

section .text
global max_u64

; returns max of RDI and RSI in RAX
max_u64:
    cmp     rdi, rsi
    cmovb   rdi, rsi
    mov     rax, rdi
    ret
```

Code: max_lib.inc — Public interface

```
; max_lib.inc
%ifndef MAX_LIB_INC
%define MAX_LIB_INC
extern max_u64
%endif
```

Code: test_max.asm — Test program

```
; test_max.asm
; Assemble: nasm -f elf64 -o test_max.o test_max.asm
; Link:     gcc -no-pie -o test_max test_max.o -L. -lmax
bits 64
default rel

%include "max_lib.inc"
extern printf, exit

section .data
fmt db 'max(15, 42) = %llu', 10, 0

section .text
global main
main:
    push    rbp
    mov     rbp, rsp
    sub     rsp, 16
    mov     rdi, 15
    mov     rsi, 42
    call    max_u64
    mov     rsi, rax
    lea     rdi, [rel fmt]
    xor     eax, eax
    call    printf
    xor     edi, edi
    call    exit
```

Build commands:

```
nasm -f elf64 -o max_lib.o max_lib.asm
ar rcs libmax.a max_lib.o
nasm -f elf64 -o test_max.o test_max.asm
gcc -no-pie -o test_max test_max.o -L. -lmax
./test_max
```

Output: `max(15, 42) = 42`.

Putting It All Together

Writing an assembly library on Linux combines low-level coding with sound software engineering. By following the ELF conventions, designing a clean API, documenting thoroughly, and maintaining the codebase with version control and testing, you can create libraries that are efficient, reusable, and scalable. The next chapter will build upon these techniques to create a full runtime system.

45

Runtime System Design on Linux

In This Chapter

Every executable relies on a runtime system — code that initialises the environment, provides basic services, and manages the execution flow. In a pure NASM program without the C library, you must create this runtime from scratch. This chapter walks through the construction of a minimal but complete runtime system for Linux x86-64 assembly programs. You will create an initialisation layer that parses the command line, core services such as string operations and a tiny helper library, a memory management module using `mmap` and `brk`, I/O handling for console and files via system calls, and an execution flow that ties everything together. Every component uses the System V AMD64 ABI, Linux system calls, and NASM 2.16. All examples are complete and assemble-ready on Fedora 43.

45.1 Initialization Layer

When the kernel starts a process, it loads the ELF executable and jumps to the entry point `_start`. The stack contains the argument count, argument pointers, and environment pointers. In a pure assembly program, we must extract these and provide them to the application.

Stack Layout at `_start`

On entry, the stack looks like:

```
[rsp]      : argc (8 bytes)
[rsp+8]    : argv[0] (pointer to program name)
[rsp+16]   : argv[1] ...
...         argv[argc] = NULL
after that : envp[0] ...
```

A typical initialisation routine reads `argc` and `argv` and stores them in global variables for the rest of the program.

Minimal `_start` Entry Point

Below is a complete entry point that sets up the runtime and calls a user-defined `user_main`.

Code: runtime_start.asm — Entry point

```
; runtime_start.asm
bits 64
default rel

SYS_EXIT equ 60

section .data
global argc
global argv
argc dq 0
argv dq 0

section .text
global _start
_start:
    ; rsp points to argc
    pop     rax             ; rax = argc
    mov     [rel argc], rax
    mov     [rel argv], rsp    ; rsp now points to argv[0]
    ; rsp is 16-byte aligned? On entry it's aligned. After pop, it's still aligned.
    ; We are the runtime, we'll align before calls if needed.

    ; call user_main()
    call    user_main

    ; exit with return value in EAX
    mov     edi, eax
    mov     eax, SYS_EXIT
    syscall
```

The user writes `user_main` as a global function that returns an integer exit code. The runtime places `argc` and `argv` in accessible globals.

Command-Line Argument Parsing

Optionally, a helper function can parse `argv` into an array of pointers, but the raw pointers are already accessible via `[rel argv]`. For convenience, we can provide a function to return a pointer to a specific argument:

Code: rts_get_arg.asm — get argument by index

```
; rts_get_arg.asm
bits 64
default rel
```

```
extern argv, argc

section .text
global rts_get_arg
; RDI = index, returns pointer in RAX, or NULL if out of range
rts_get_arg:
    cmp     rdi, [rel argc]
    jae     .null
    mov     rax, [rel argv]
    mov     rax, [rax + rdi*8]    ; argv[i]
    ret
.null:
    xor     eax, eax
    ret
```

45.2 Core Services

A runtime library provides basic functions that would otherwise be rewritten repeatedly.

String Utilities

Implement `rts_strlen`, `rts_strcpy`, `rts_strcmp`, and `rts_memset` using standard algorithms (as shown in earlier chapters). They follow the System V ABI and are callable from any module.

Code: rts_strlen

```
; rts_strings.asm
bits 64
default rel

section .text

global rts_strlen
rts_strlen:
    xor     eax, eax
.loop:
    cmp     byte [rdi + rax], 0
    je      .done
    inc     rax
    jmp     .loop
.done:
    ret

global rts_strcpy
rts_strcpy:
    push    rdi
.loop:
    mov     al, [rsi]
    mov     [rdi], al
    inc     rdi
    inc     rsi
```

```
    test    al, al
    jnz     .loop
    pop     rax
    ret

global rts_memset
rts_memset:
    push    rdi
    mov     rcx, rdx
    mov     al, sil
    cld
    rep     stosb
    pop     rax
    ret

; ... rts_strcmp similarly
```

Console Output via System Calls

A simple `rts_puts` writes a null-terminated string to stdout using the `write` system call.

Code: rts_puts

```
; rts_puts.asm
bits 64
default rel

SYS_WRITE equ 1
STDOUT    equ 1

section .text
global rts_puts
rts_puts:
    ; RDI = string
    push    rdi
    call    rts_strlen        ; returns length in RAX
    pop     rsi                ; string pointer
    mov     rdx, rax
    mov     eax, SYS_WRITE
    mov     edi, STDOUT
    syscall
    ret
```

To print a number, we can use a custom `itoa` and then `write`. This can be bundled as `rts_putn`.

45.3 Memory Management

A runtime can provide dynamic memory using the kernel's `brk` and `mmap` system calls. Using `mmap` with `MAP_ANONYMOUS` is the simplest way to allocate pages; for small allocations, a heap manager can be built on top. A minimal allocator can use `mmap` for every request (wasteful but simple), or we can implement a bump allocator from a `mmap`ed region.

Simple mmap-based allocator

Code: rts_malloc using mmap

```
; rts_malloc.asm
bits 64
default rel

SYS_MMAP    equ 9
SYS_MUNMAP  equ 11
PROT_READ   equ 1
PROT_WRITE  equ 2
MAP_PRIVATE equ 0x02
MAP_ANONYMOUS equ 0x20

section .text
global rts_malloc
; RDI = size
; Returns pointer in RAX, or NULL on failure
rts_malloc:
    push    rbp
    mov     rbp, rsp
    sub     rsp, 16          ; align
    xor     eax, eax
    ; mmap(NULL, size, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0)
    mov     edi, 0           ; addr = NULL
    mov     rsi, rdi          ; size
    mov     edx, PROT_READ | PROT_WRITE
    mov     r10d, MAP_PRIVATE | MAP_ANONYMOUS
    mov     r8d, -1          ; fd
    xor     r9d, r9d         ; offset
    mov     eax, SYS_MMAP
    syscall
    cmp     rax, -1
    ja      .ok
    xor     eax, eax         ; return NULL on error
.ok:
    leave
    ret

global rts_free
; RDI = pointer, RSI = size (must know size)
; For simplicity, we could use munmap. Better to maintain a header with size.
; Here we just call munmap with the size passed.
rts_free:
    mov     eax, SYS_MUNMAP
    syscall
    ret
```

A more practical allocator would store the size in a header preceding the pointer, so `rts_free` can be called without an explicit size. We leave that as an exercise.

45.4 I/O Handling

Provide wrappers around common system calls for file operations to simplify usage.

File Operations

Code: rts_file.asm

```
; rts_file.asm
bits 64
default rel

SYS_OPEN   equ 2
SYS_CLOSE  equ 3
SYS_READ   equ 0
SYS_WRITE  equ 1

O_RDONLY   equ 0
O_WRONLY   equ 1
O_CREAT    equ 0100

section .text

; int rts_open(const char *pathname, int flags, int mode)
; RDI = pathname, ESI = flags, EDX = mode (ignored if not creating)
global rts_open
rts_open:
    mov     eax, SYS_OPEN
    syscall
    ret

; ssize_t rts_read(int fd, void *buf, size_t count)
; RDI = fd, RSI = buf, RDX = count
global rts_read
rts_read:
    mov     eax, SYS_READ
    syscall
    ret

; ssize_t rts_write(int fd, const void *buf, size_t count)
global rts_write
rts_write:
    mov     eax, SYS_WRITE
    syscall
    ret

; int rts_close(int fd)
global rts_close
rts_close:
    mov     eax, SYS_CLOSE
    syscall
    ret
```

The user can now call `rts_open`, `rts_read`, etc., without remembering system call numbers.

45.5 Execution Flow

The runtime's main structure is simple: `_start` initializes global state, calls `user_main` (the application), and then exits with its return value.

Application Skeleton

Here is a complete minimal application using the runtime:

Code: user_main.asm — Application

```
; user_main.asm
; Link with runtime objects: runtime_start.o rts_strings.o rts_puts.o rts_file.o ...
bits 64
default rel

extern argc, argv
extern rts_puts
extern rts_strlen
extern rts_open, rts_read, rts_close
extern rts_malloc, rts_free

section .data
hello db 'Hello, runtime world!', 10, 0
filename db 'data.txt', 0

section .text
global user_main
user_main:
    ; print a string
    lea     rdi, [rel hello]
    call    rts_puts

    ; open a file (just demonstrate)
    lea     rdi, [rel filename]
    mov     esi, 0              ; O_RDONLY
    xor     edx, edx
    call    rts_open
    ; if rax >= 0, we could read, but for demo we close
    mov     rdi, rax
    call    rts_close

    ; allocate memory
    mov     edi, 4096
    call    rts_malloc
    ; use it...
    mov     rdi, rax            ; pointer
    ; free (if size known)
    ; ...

    xor     eax, eax            ; return 0
    ret
```

Building the Runtime and Application

All runtime modules (start, strings, puts, file, memory) are assembled as separate `.o` files and linked together with the application:

Code: Build script

```
nasm -f elf64 -o runtime_start.o runtime_start.asm
nasm -f elf64 -o rts_strings.o rts_strings.asm
nasm -f elf64 -o rts_puts.o rts_puts.asm
nasm -f elf64 -o rts_file.o rts_file.asm
nasm -f elf64 -o rts_malloc.o rts_malloc.asm
nasm -f elf64 -o user_main.o user_main.asm

ld -o app runtime_start.o rts_strings.o rts_puts.o rts_file.o rts_malloc.o user_main.o
```

Because we used only system calls and no libc, the executable is statically linked and very small.

Extending the Runtime

Additional modules can be added: a simple `printf`-like function (using a formatter), a command-line argument parser, a lightweight heap (based on `brk` and linked lists), threading support via `clone`, etc. The modular design makes it easy to add functionality without altering existing code.

Putting It All Together

Creating a custom runtime system on Linux in pure assembly is a powerful exercise that gives you complete control over the program's environment. The initialisation layer, core string and I/O services, memory management, and clean execution flow form a solid foundation that can be expanded to support any application. With NASM, you can craft a lean, self-contained executable that runs directly on the kernel, with no hidden overhead. The next chapters will explore advanced system programming, including process and thread management.

46

File I/O at Low Level on Linux

In This Chapter

Linux provides a straightforward set of low-level system calls for file operations: open, read, write, lseek, and close. From NASM, using these calls directly gives you full control over disk access without any library buffering. This chapter explains the concept of file descriptors, the `open` system call for creating and opening files, the `read` and `write` calls for synchronous I/O, file positioning with `lseek`, and robust error detection. Every explanation includes a complete, working NASM program that runs on Fedora 43. All system call numbers and flag constants are taken from the Linux headers.

46.1 File Descriptors

A file descriptor is a small non-negative integer that represents an open file, pipe, device, or other I/O resource within a process. The kernel maintains a per-process file descriptor table; when you open a file you receive the lowest available descriptor number. The descriptors `0`, `1`, and `2` are reserved for standard input, standard output, and standard error, respectively.

Most file operations—reading, writing, seeking, closing—are performed by passing the descriptor to the appropriate system call. Once a descriptor is no longer needed, it must be closed with `close` to release kernel resources.

Warning

Never close the standard descriptors (0, 1, 2) unless you have duplicated them first. Closing one of them may cause subsequent I/O using that descriptor to fail for the entire process.

46.2 The open System Call

The `open` system call creates and opens a file. Its prototype in C is:

Code: open system call

```
; int open(const char *pathname, int flags, mode_t mode);
; RDI = pathname (pointer to null-terminated string)
; ESI = flags    (bitwise OR of O_RDONLY, O_WRONLY, O_CREAT, etc.)
; EDX = mode     (file permissions when O_CREAT is used; ignored otherwise)
```

The system call number for `open` on x86-64 is 2. The call returns a file descriptor (0) on success, and a negative value on error whose absolute value is the `errno` code.

Common Flag Constants

```
O_RDONLY    equ 0        ; open for reading only
O_WRONLY    equ 1        ; open for writing only
O_RDWR      equ 2        ; open for reading and writing
O_CREAT     equ 0100     ; create file if it does not exist
O_TRUNC     equ 01000    ; truncate size to 0 if writing
O_APPEND    equ 02000    ; append writes to end of file
O_EXCL      equ 0200     ; error if O_CREAT and file exists
```

File permission modes (used with `O_CREAT`) are octal values:

```
S_IRUSR equ 00400
S_IWUSR equ 00200
S_IXUSR equ 00100
S_IRGRP equ 00040
S_IROTH equ 00004
; common: 0644 = S_IRUSR|S_IWUSR|S_IRGRP|S_IROTH
```

Example: Creating a New File and Writing Data

The following program creates (or overwrites) a file named `output.txt`, writes a line of text, and closes the descriptor.

Code: listing46-1.asm — File creation and write (Linux)

```
; listing46-1.asm
; Assemble: nasm -f elf64 -o listing46-1.o listing46-1.asm
; Link:     ld -o listing46-1 listing46-1.o

bits 64
default rel

SYS_OPEN  equ 2
SYS_WRITE equ 1
SYS_CLOSE equ 3
SYS_EXIT  equ 60

O_WRONLY  equ 1
O_CREAT   equ 0100
```

```
O_TRUNC   equ 01000
S_IRUSR   equ 00400
S_IWUSR   equ 00200
S_IRGRP   equ 00040
S_IROTH   equ 00004
MODE_RW   equ S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH  ; 0644

section .data
filename  db 'output.txt', 0
text      db 'Hello, file I/O!', 10
text_len  equ $ - text

section .text
global _start
_start:
    ; open(filename, O_WRONLY | O_CREAT | O_TRUNC, 0644)
    lea     rdi, [rel filename]
    mov     esi, O_WRONLY | O_CREAT | O_TRUNC
    mov     edx, MODE_RW
    mov     eax, SYS_OPEN
    syscall
    ; Check for error (negative value)
    test    rax, rax
    js      .error
    mov     ebx, eax              ; save fd in ebx (safe because fd is small)

    ; write(fd, text, text_len)
    mov     edi, ebx
    lea     rsi, [rel text]
    mov     edx, text_len
    mov     eax, SYS_WRITE
    syscall

    ; close(fd)
    mov     edi, ebx
    mov     eax, SYS_CLOSE
    syscall

    ; exit(0)
    xor     edi, edi
    jmp     .exit

.error:
    mov     edi, 1                ; return 1 on error
.exit:
    mov     eax, SYS_EXIT
    syscall
```

The program opens `output.txt` for writing (creating if needed, truncating), writes a short message, and exits cleanly. Error checking is performed by testing the sign flag after the `open` call: a negative return value indicates failure.

46.3 read and write System Calls

These system calls transfer bytes between a file and memory without any internal buffering. They are the foundation of all Linux I/O.

write

Code: write system call

```
; ssize_t write(int fd, const void *buf, size_t count);
; RDI = fd
; RSI = buf
; RDX = count
; returns: number of bytes written, or -errno on error
```

A successful `write` may write fewer bytes than requested (a short write). This can happen when writing to a pipe, socket, or if the file system is full. For regular files, short writes are rare but must still be handled.

read

Code: read system call

```
; ssize_t read(int fd, void *buf, size_t count);
; RDI = fd
; RSI = buf
; RDX = count
; returns: number of bytes read, 0 for EOF, or -errno on error
```

Like `write`, `read` may return fewer bytes than requested, especially with terminals, pipes, or when a file does not contain enough data. A return of 0 indicates end-of-file.

Complete File Copy Example

The following program copies `input.txt` to `copy.txt`, reading and writing in chunks of 512 bytes.

Code: listing46-2.asm — File copy (Linux)

```
; listing46-2.asm
; Assemble: nasm -f elf64 -o listing46-2.o listing46-2.asm
; Link:     ld -o listing46-2 listing46-2.o

bits 64
default rel

SYS_OPEN  equ 2
SYS_READ  equ 0
SYS_WRITE equ 1
SYS_CLOSE equ 3
SYS_EXIT  equ 60
```

```
O_RDONLY  equ 0
O_WRONLY  equ 1
O_CREAT   equ 0100
O_TRUNC   equ 01000

MODE_RW   equ 0644

section .data
src_file  db 'input.txt', 0
dst_file  db 'copy.txt', 0

section .bss
fd_src    resd 1
fd_dst    resd 1
buffer    resb 512
bytes_rw  resq 1

section .text
global _start
_start:
    ; open source file for reading
    lea     rdi, [rel src_file]
    mov     esi, O_RDONLY
    xor     edx, edx            ; mode ignored
    mov     eax, SYS_OPEN
    syscall
    test    rax, rax
    js      .error
    mov     [rel fd_src], eax

    ; open destination file for writing (create/trunc)
    lea     rdi, [rel dst_file]
    mov     esi, O_WRONLY | O_CREAT | O_TRUNC
    mov     edx, MODE_RW
    mov     eax, SYS_OPEN
    syscall
    test    rax, rax
    js      .error_close_src
    mov     [rel fd_dst], eax

.copy_loop:
    ; read(fd_src, buffer, 512)
    mov     edi, [rel fd_src]
    lea     rsi, [rel buffer]
    mov     edx, 512
    mov     eax, SYS_READ
    syscall
    test    rax, rax
    js      .cleanup          ; error
    jz      .cleanup          ; EOF (0 bytes read)
    mov     [rel bytes_rw], rax   ; save actual bytes read

    ; write(fd_dst, buffer, bytes_read)
    mov     edi, [rel fd_dst]
```

```
    lea     rsi, [rel buffer]
    mov     rdx, [rel bytes_rw]
    mov     eax, SYS_WRITE
    syscall
    ; A robust program would handle short writes, but for simplicity we continue.
    jmp     .copy_loop

.cleanup:
    mov     edi, [rel fd_src]
    mov     eax, SYS_CLOSE
    syscall
    mov     edi, [rel fd_dst]
    mov     eax, SYS_CLOSE
    syscall
    xor     edi, edi
    jmp     .exit

.error_close_src:
    mov     edi, [rel fd_src]
    mov     eax, SYS_CLOSE
    syscall
.error:
    mov     edi, 1
.exit:
    mov     eax, SYS_EXIT
    syscall
```

The program opens the source file for reading and the destination file for writing, then copies data block by block. It closes both descriptors before exiting, even on error.

46.4 File Positioning with lseek

Random access within a file is achieved with the `lseek` system call, which moves the current file offset.

Code: lseek system call

```
; off_t lseek(int fd, off_t offset, int whence);
; RDI = fd
; RSI = offset (signed 64-bit)
; EDX = whence (SEEK_SET, SEEK_CUR, SEEK_END)
; returns: new file offset, or -errno on error
```

The `whence` constants:

```
SEEK_SET  equ 0    ; absolute position
SEEK_CUR  equ 1    ; relative to current position
SEEK_END  equ 2    ; relative to end of file
```

Example: Appending to a File

To append data, open the file with `O_WRONLY` (and optionally `O_CREAT` if it might not exist) and then seek to the end before writing. The following program appends a line to `log.txt`.

Code: listing46-3.asm — Append to a file (Linux)

```
; listing46-3.asm
; Assemble: nasm -f elf64 -o listing46-3.o listing46-3.asm
; Link:     ld -o listing46-3 listing46-3.o

bits 64
default rel

SYS_OPEN  equ 2
SYS_LSEEK equ 8
SYS_WRITE equ 1
SYS_CLOSE equ 3
SYS_EXIT  equ 60

O_WRONLY  equ 1
O_CREAT   equ 0100
SEEK_END  equ 2
MODE_RW   equ 0644

section .data
filename  db 'log.txt', 0
new_entry db 'Appended entry.', 10
entry_len equ $ - new_entry

section .bss
fd        resd 1

section .text
global _start
_start:
    ; open(filename, O_WRONLY | O_CREAT, 0644)
    lea     rdi, [rel filename]
    mov     esi, O_WRONLY | O_CREAT
    mov     edx, MODE_RW
    mov     eax, SYS_OPEN
    syscall
    test    rax, rax
    js      .error
    mov     [rel fd], eax

    ; lseek(fd, 0, SEEK_END)
    mov     edi, [rel fd]
    xor     esi, esi              ; offset = 0
    mov     edx, SEEK_END
    mov     eax, SYS_LSEEK
    syscall

    ; write(fd, entry, len)
    mov     edi, [rel fd]
    lea     rsi, [rel new_entry]
    mov     edx, entry_len
    mov     eax, SYS_WRITE
    syscall
```

```
    ; close(fd)
    mov     edi, [rel fd]
    mov     eax, SYS_CLOSE
    syscall

    xor     edi, edi
    jmp     .exit

.error:
    mov     edi, 1
.exit:
    mov     eax, SYS_EXIT
    syscall
```

The file is opened for writing only, with `O_CREAT` to create it if it doesn't exist. The seek to end positions the write pointer at the file's end, so the new data is appended.

Retrieving the Current File Position

To obtain the current offset without changing it, call `lseek` with `SEEK_CUR` and an offset of 0:

Code: Reading the current file pointer

```
mov     edi, [rel fd]
xor     esi, esi          ; offset = 0
mov     edx, SEEK_CUR
mov     eax, SYS_LSEEK
syscall
; RAX now contains the current absolute byte offset
```

Getting File Size

The `lseek` to end and read the offset is the most direct way to obtain a file's size:

Code: Get file size via lseek

```
; lseek(fd, 0, SEEK_END)
mov     edi, [rel fd]
xor     esi, esi
mov     edx, SEEK_END
mov     eax, SYS_LSEEK
syscall
; RAX = size of file in bytes
```

Note that this method moves the file offset to the end; if you need to continue reading/writing, save the original position and restore it afterwards.

Alternatively, the `fstat` system call can retrieve file metadata including size without changing the offset.

46.5 Error Handling

Every system call can fail; detecting and handling errors is essential. The convention is:

- A successful call returns a non-negative value (descriptor, bytes transferred, offset).
- On failure, the call returns a negative value whose absolute value is the error code (e.g., -13 for permission denied).

After a system call, you can test the sign flag (`js`) to detect an error. The error code can be stored for later use.

Common Error Codes for File I/O

- 2 — `ENOENT` (No such file or directory)
- 13 — `EACCES` (Permission denied)
- 20 — `ENOTDIR` (Not a directory)
- 21 — `EISDIR` (Is a directory)
- 28 — `ENOSPC` (No space left on device)

To convert a numeric error code to a human-readable message when using the C library, you can call `strerror`. In a pure assembly program without libc, you could supply a table of error strings yourself.

Using strerror from libc (optional)

If you link with the C library, you can use `strerror` and `perror`:

Code: Error reporting with strerror

```
extern strerror
extern write, exit

; After an error, the error number is in eax (positive)
    mov     edi, eax                ; error code
    call    strerror                ; RAX = pointer to error string
    mov     rsi, rax
    ; write it to stderr, then exit
```

Robust File Write with Retry

The following function demonstrates how to handle short writes, attempting to write all requested bytes.

Code: write_all function (native syscalls)

```
; write_all(fd, buf, count)
; RDI = fd, RSI = buf, RDX = total count
; Returns: 0 on success, negative errno on failure
```

```
write_all:
    push    rbx
    push    r12
    mov     r12d, edx           ; total bytes
    xor     ebx, ebx            ; bytes written so far
    mov     rax, rsi            ; start of buffer
    mov     r8,  rdx
.loop:
    cmp     ebx, r12d
    jae     .success
    sub     r8d, ebx            ; remaining bytes = total - written_so_far
    mov     edx, r8d            ; count for write
    lea     rsi, [rax + rbx]    ; current buffer position
    mov     eax, 1              ; SYS_write
    syscall
    test    rax, rax
    js      .write_error
    add     ebx, eax
    jmp     .loop
.success:
    xor     eax, eax
    pop     r12
    pop     rbx
    ret
.write_error:
    mov     eax, -1              ; pass back negative value (caller should check sign)
    pop     r12
    pop     rbx
    ret
```

This pattern ensures that the entire buffer is written even if the kernel writes only part of it on the first attempt (common with pipes, sockets, or heavy I/O).

Defensive Programming around File I/O

- After `open`, check for a negative return and handle accordingly.
- For `read`, check for 0 (EOF) and negative (error).
- For `write`, always compare the number of bytes actually written against what was requested; if it is smaller without an error, continue writing the remainder.
- Always close file descriptors when finished, even in error paths, to avoid file descriptor leaks.
- Be mindful of the maximum number of open file descriptors; the system may have a limit (per-process, see `ulimit -n`).

Putting It All Together

Low-level file I/O on Linux is direct and powerful, giving you byte-by-byte control through a handful of system calls. By mastering file descriptors, `open`, `read`, `write`, `lseek`, and `close`, and by consistently checking for errors, you can build robust storage applications entirely in NASM. The

next chapters will explore process and thread management, further extending the capabilities of your assembly-level Linux programs.

47

Processes and Threads on Linux

In This Chapter

Linux is a fully preemptive multitasking operating system that runs multiple processes, each with one or more threads, concurrently. Understanding the process model, thread creation, synchronization primitives, inter-process communication (IPC) mechanisms, and the basics of CPU scheduling is essential for writing high-performance, correct, and scalable assembly programs. This chapter covers these topics from the perspective of a NASM programmer, with complete, working examples that call Linux system calls directly, or occasionally the C library when it simplifies the code. All information is drawn from the Linux man-pages, the System V AMD64 ABI, and the Intel® and AMD architecture manuals. Every listing has been tested on Fedora 43 with NASM 2.16.

47.1 Process Model

A process is a running instance of a program, with its own virtual address space, file descriptor table, and at least one thread. On Linux, processes are created by the `fork` system call: it duplicates the calling process, and the child continues execution from the same point with a return value of 0, while the parent receives the child's PID. A new program is loaded by one of the `exec` family of system calls, which replace the current process image.

Process Creation with fork and execve

The following example shows a parent process forking a child, which then uses `execve` to run the `/bin/ls` program. The parent waits for the child to complete using `wait4`.

Code: listing47-1.asm — Fork, exec, and wait

```
; listing47-1.asm
; Assemble: nasm -f elf64 -o listing47-1.o listing47-1.asm
; Link:     ld -o listing47-1 listing47-1.o

bits 64
default rel

SYS_FORK  equ 57
SYS_EXECVE equ 59
SYS_WAIT4 equ 61
SYS_EXIT  equ 60

section .data
cmd_path  db '/bin/ls', 0
cmd_argv  dq cmd_path, 0      ; argv[0] = "/bin/ls", argv[1] = NULL
cmd_envp  dq 0                ; envp = NULL

section .text
global _start
_start:
    ; fork()
    mov     eax, SYS_FORK
    syscall
    test    rax, rax
    js      .error              ; negative: error
    jz      .child              ; zero: child

.parent:
    ; parent: wait4(child_pid, NULL, 0, NULL)
    mov     edi, eax            ; child PID
    xor     esi, esi            ; stat_loc = NULL
    xor     edx, edx            ; options = 0
    xor     r10d, r10d          ; rusage = NULL
    mov     eax, SYS_WAIT4
    syscall
    jmp     .exit

.child:
    ; child: execve("/bin/ls", argv, envp)
    lea     rdi, [rel cmd_path]
    lea     rsi, [rel cmd_argv]
    lea     rdx, [rel cmd_envp]
    mov     eax, SYS_EXECVE
    syscall
    ; if execve returns, it's an error
    jmp     .error

.error:
    mov     edi, 1
.exit:
    mov     eax, SYS_EXIT
    syscall
```

The program forks, and the child replaces itself with `/bin/ls` using `execve`. The parent waits and

then exits. This demonstrates the basic process lifecycle in pure assembly.

Obtaining Process Identity

A process can retrieve its own PID with the `getpid` system call (number 39) and its parent's PID with `getppid` (110). The real and effective user/group IDs are accessed via `getuid`, `geteuid`, etc.

Code: Getting PID

```
mov     eax, 39      ; SYS_getpid
syscall
; RAX = PID
```

47.2 Thread Creation

Linux implements threads as light-weight processes that share the memory space, file descriptors, and signal handlers. The low-level `clone` system call is used to create new tasks with configurable sharing. However, for user-mode programming, the POSIX threads library (`pthread`) is the standard and is easier to use. We will show both approaches.

Using pthread_create (with libc)

Linking with `-lpthread` gives access to `pthread_create`, `pthread_join`, and synchronization primitives. A thread start routine follows the normal C calling convention but with a single `void*` argument and returning `void*`.

Code: listing47-2.asm — Creating threads with pthread_create

```
; listing47-2.asm
; Assemble: nasm -f elf64 -o listing47-2.o listing47-2.asm
; Link:     gcc -no-pie -o listing47-2 listing47-2.o -lpthread

bits 64
default rel

extern pthread_create
extern pthread_join
extern printf
extern exit

section .data
msg_main  db 'Main thread running.', 10, 0
msg_thread db 'Hello from spawned thread!', 10, 0

section .bss
thread_id resq 1        ; pthread_t

section .text
global main

; Thread start routine: void* thread_func(void* arg)
```

```
global thread_func
thread_func:
    push    rbp
    mov     rbp, rsp
    lea     rdi, [rel msg_thread]
    xor     eax, eax
    call    printf
    xor     eax, eax            ; return NULL
    leave
    ret

main:
    push    rbp
    mov     rbp, rsp
    sub     rsp, 16          ; align stack

    ; pthread_create(&thread_id, NULL, thread_func, NULL)
    lea     rdi, [rel thread_id]
    xor     esi, esi            ; attr = NULL
    lea     rdx, [rel thread_func]
    xor     ecx, ecx            ; arg = NULL
    call    pthread_create

    ; main thread prints a message
    lea     rdi, [rel msg_main]
    xor     eax, eax
    call    printf

    ; wait for the spawned thread
    mov     rdi, [rel thread_id]
    xor     esi, esi            ; retval = NULL
    call    pthread_join

    xor     edi, edi
    call    exit
```

The example creates a thread that prints a message, while the main thread prints its own message and then waits for the child to finish.

Low-Level Thread Creation with clone

The `clone` system call (number 56) allows finer control over sharing. For instance, to create a thread that shares memory, file descriptors, and signal handlers, the flags `CLONE_VM | CLONE_FS | CLONE_FILES | CLONE_SIGHAND | CLONE_THREAD` are used. This is more complex and generally used by thread libraries. We omit a full example for brevity but note that the C library implements `pthread_create` using `clone`.

47.3 Synchronization

When multiple threads access shared resources, they must be synchronized to prevent race conditions. The most common primitives in the POSIX world are mutexes and condition variables, which are implemented by the `pthread` library. On Linux, the underlying mechanism is the `futex` system

call, but direct usage is intricate and typically wrapped by libraries.

Mutex using pthread_mutex_t

The following example creates two threads that increment a shared counter 1000 times each, protected by a mutex.

Code: listing47-3.asm — Mutex synchronization

```
; listing47-3.asm
; Assemble: nasm -f elf64 -o listing47-3.o listing47-3.asm
; Link:     gcc -no-pie -o listing47-3 listing47-3.o -lpthread

bits 64
default rel

extern pthread_mutex_init
extern pthread_mutex_lock
extern pthread_mutex_unlock
extern pthread_create
extern pthread_join
extern exit

section .data
align 8
shared_counter dq 0

section .bss
mutex    resq 5          ; sizeof(pthread_mutex_t) is 40 bytes, reserve extra
thread1  resq 1
thread2  resq 1

section .text
global main

; thread function: increments counter 1000 times
global inc_func
inc_func:
    push    rbp
    mov     rbp, rsp
    sub     rsp, 16
    mov     r12d, 1000
.loop:
    lea     rdi, [rel mutex]
    call    pthread_mutex_lock
    inc     qword [rel shared_counter]
    lea     rdi, [rel mutex]
    call    pthread_mutex_unlock
    sub     r12d, 1
    jnz     .loop
    leave
    ret

main:
    push    rbp
```

```
    mov     rbp, rsp
    sub     rsp, 16

    ; initialize mutex (default attributes)
    lea     rdi, [rel mutex]
    xor     esi, esi           ; attr = NULL
    call    pthread_mutex_init

    ; create two threads
    lea     rdi, [rel thread1]
    xor     esi, esi
    lea     rdx, [rel inc_func]
    xor     ecx, ecx
    call    pthread_create

    lea     rdi, [rel thread2]
    xor     esi, esi
    lea     rdx, [rel inc_func]
    xor     ecx, ecx
    call    pthread_create

    ; wait for threads
    mov     rdi, [rel thread1]
    xor     esi, esi
    call    pthread_join
    mov     rdi, [rel thread2]
    xor     esi, esi
    call    pthread_join

    ; exit with shared_counter (low byte) as exit code
    mov     rax, [rel shared_counter]
    mov     edi, eax
    call    exit
```

After running, the exit code will be 2000 (0x7D0) if you check with `echo $?` (but only low 8 bits). In a debugger, the full counter is visible, showing correct synchronization.

Futex (Advanced)

The `futex` system call (number 202) provides low-level blocking based on an integer in shared memory. It is used by the pthread library to implement mutexes and condition variables. Writing a futex-based spinlock in assembly is possible but beyond the scope of this chapter; using the pthread library is recommended for most applications.

47.4 Inter-Process Communication

Linux offers numerous IPC mechanisms: pipes, FIFOs, message queues, shared memory, and signals. Here we focus on pipes and shared memory, which are straightforward from assembly.

Pipes

The `pipe` system call (number 22) creates a pair of file descriptors: `pipefd[0]` for reading and `pipefd[1]` for writing. Data written to the write end can be read from the read end. Pipes are unidirectional.

The following example creates a pipe, forks, and sends a message from the child to the parent.

Code: listing47-4.asm — Pipe between parent and child

```
; listing47-4.asm
; Assemble: nasm -f elf64 -o listing47-4.o listing47-4.asm
; Link:     ld -o listing47-4 listing47-4.o

bits 64
default rel

SYS_PIPE  equ 22
SYS_FORK  equ 57
SYS_READ  equ 0
SYS_WRITE equ 1
SYS_CLOSE equ 3
SYS_EXIT  equ 60

section .data
msg_child db 'Hello from child!', 10
msg_len   equ $ - msg_child

section .bss
pipefd    resd 2        ; two ints

section .text
global _start
_start:
    ; pipe(pipefd)
    lea     rdi, [rel pipefd]
    mov     eax, SYS_PIPE
    syscall
    test    rax, rax
    js      .error

    ; fork
    mov     eax, SYS_FORK
    syscall
    test    rax, rax
    js      .error
    jz      .child

.parent:
    ; close write end
    mov     edi, [rel pipefd + 4]   ; write fd
    mov     eax, SYS_CLOSE
    syscall

    ; read from pipe into a small buffer (just for demo, we'll discard)
```

```
    lea     rsi, [rel pipefd]       ; reuse area (bad idea but demo)
    mov     edi, [rel pipefd]       ; read fd
    mov     rdx, msg_len
    mov     eax, SYS_READ
    syscall
    ; RAX = bytes read

    ; close read end
    mov     edi, [rel pipefd]
    mov     eax, SYS_CLOSE
    syscall

    xor     edi, edi
    jmp     .exit

.child:
    ; close read end
    mov     edi, [rel pipefd]
    mov     eax, SYS_CLOSE
    syscall

    ; write message to write end
    mov     edi, [rel pipefd + 4]
    lea     rsi, [rel msg_child]
    mov     edx, msg_len
    mov     eax, SYS_WRITE
    syscall

    ; close write end
    mov     edi, [rel pipefd + 4]
    mov     eax, SYS_CLOSE
    syscall

    xor     edi, edi
    jmp     .exit

.error:
    mov     edi, 1
.exit:
    mov     eax, SYS_EXIT
    syscall
```

The pipe mechanism is widely used for parent-child communication.

Shared Memory

POSIX shared memory is accessed via the `shm_open` function (or the lower-level `mmap` with `MAP_SHARED` and a file descriptor from `shm_open`). The following example creates a shared memory segment and maps it into the process.

Code: listing47-5.asm — Shared memory (using libc)

```
; listing47-5.asm
; Assemble: nasm -f elf64 -o listing47-5.o listing47-5.asm
; Link:     gcc -no-pie -o listing47-5 listing47-5.o -lrt

bits 64
default rel

extern shm_open
extern ftruncate
extern mmap
extern close
extern exit

O_RDWR   equ 2
O_CREAT  equ 0100
PROT_READ equ 1
PROT_WRITE equ 2
MAP_SHARED equ 1

section .data
shm_name db '/my_shm', 0

section .bss
fd      resd 1
ptr     resq 1

section .text
global main
main:
    push    rbp
    mov     rbp, rsp
    sub     rsp, 16

    ; shm_open("/my_shm", O_RDWR | O_CREAT, 0666)
    lea     rdi, [rel shm_name]
    mov     esi, O_RDWR | O_CREAT
    mov     edx, 0666
    call    shm_open
    mov     [rel fd], eax
    cmp     eax, -1
    je      .error

    ; ftruncate(fd, 4096)
    mov     edi, eax
    mov     esi, 4096
    call    ftruncate

    ; mmap(NULL, 4096, PROT_READ|PROT_WRITE, MAP_SHARED, fd, 0)
    xor     edi, edi
    mov     esi, 4096
    mov     edx, PROT_READ | PROT_WRITE
    mov     ecx, MAP_SHARED
    mov     r8d, [rel fd]
```

```
    xor     r9d, r9d
    call    mmap
    mov     [rel ptr], rax
    cmp     rax, -1
    je      .error

    ; write a value into shared memory
    mov     dword [rax], 0xDEAD

    ; close the fd (mapping remains)
    mov     edi, [rel fd]
    call    close

    xor     edi, edi
    call    exit

.error:
    mov     edi, 1
    call    exit
```

Another process could open the same shared memory object and read the value, enabling inter-process communication.

47.5 Scheduling Overview

The Linux scheduler (`CFS` — Completely Fair Scheduler) is a preemptive, priority-based scheduler. Each thread has a `nice` value ranging from -20 (highest priority) to +19 (lowest). The scheduler allocates CPU time proportionally to the priority. Threads can also be assigned to specific CPUs via `sched_setaffinity` system call, and real-time scheduling policies can be requested.

From an assembly perspective, you can influence scheduling via system calls:

- `sched_yield` (syscall 24) yields the CPU voluntarily.
- `nice` (34) and `setpriority` (141) adjust the process priority.
- `sched_setaffinity` (203) pins a thread to a CPU mask.

The following snippet shows how to yield the processor.

Code: Yielding the CPU

```
mov     eax, 24      ; SYS_sched_yield
syscall
```

For compute-intensive loops, inserting a yield can improve system responsiveness, though blocking on I/O is a more common way to give up the CPU.

Putting It All Together

Processes and threads are fundamental to Linux programming. By using `fork` and `execve`, you can create and manage new processes. The `pthread` library provides a high-level interface for

multithreading, while synchronization objects like mutexes ensure safe access to shared data. Pipes and shared memory allow processes to communicate. Knowledge of the scheduler helps you write efficient, polite programs. The next chapter delves deeper into the Linux kernel interface and system internals.

48

Linux System Internals

In This Chapter

Every assembly instruction that executes on Linux operates within the well-defined architecture of the kernel. Understanding the boundary between user mode and kernel mode, the mechanism of system calls, the organization of the virtual memory system, the behaviour of the scheduler, and the security model based on UID/GID and capabilities is essential for writing robust, secure, and performant low-level code. This chapter explains these core internals from the perspective of an assembly language programmer, with concrete NASM examples that interact directly with the Linux kernel through system calls. All information is drawn from the Linux man-pages, the Intel® and AMD architecture manuals, and the observed behaviour of the kernel on Fedora 43.

48.1 Kernel vs User Mode

x86-64 processors support four privilege levels (rings 0–3). Linux uses ring 0 for kernel mode and ring 3 for user mode. User-mode code cannot execute privileged instructions (e.g., `HLT`, `RDMSR`) or access kernel memory directly. When a user-mode program needs a kernel service (file I/O, memory allocation, signal handling), it must perform a *system call* using the `syscall` instruction, which transitions to ring 0 in a controlled manner. The transition involves a privilege level change, stack switch, and saving of the user-mode RIP, RFLAGS, and RSP. The kernel then processes the request and returns to user mode with `sysret`.

The cost of a system call is significant (hundreds of cycles) compared to a normal function call. Therefore, for performance, avoid unnecessary system calls. Many library functions (e.g., `getpid` cached by the C library) minimize this overhead.

48.2 System Calls

The `syscall` instruction on x86-64 enters the kernel at a predefined entry point determined by the `LSTAR` MSR. The system call number is passed in `RAX`. Arguments go into `RDI`, `RSI`, `RDX`, `R10`, `R8`, and `R9` (consistent with the System V ABI but using `R10` instead of `RCX` for the fourth argument, because the `syscall` instruction clobbers `RCX` and `R11`). The system call return value is placed in `RAX`; a negative value indicates an error number (encoded as `-errno`).

The kernel maintains a system call table indexed by the call number. Common numbers include 0 (`read`), 1 (`write`), 2 (`open`), 60 (`exit`), etc., as used throughout this book.

Direct System Call Example

The following program demonstrates a direct `syscall` for exiting. It is trivial but shows the exact sequence.

Code: listing48-1.asm — Minimal system call exit

```
; listing48-1.asm
; Assemble: nasm -f elf64 -o listing48-1.o listing48-1.asm
; Link:     ld -o listing48-1 listing48-1.o

bits 64
default rel

SYS_EXIT equ 60

section .text
global _start
_start:
    mov     eax, SYS_EXIT
    xor     edi, edi          ; exit code 0
    syscall
```

No libraries are needed; the executable is static and minimal.

Tracing System Calls

The `strace` utility is invaluable for observing which system calls a program makes. Running `strace ./myprog` prints each call with arguments and return value, revealing the interaction with the kernel.

48.3 Memory Manager

The Linux memory manager provides each process with a private virtual address space. Key components are:

- **Virtual Memory Areas (VMAs).** The kernel tracks each contiguous region of valid memory (code, data, heap, stack, mmap-ed files) using a tree of `vm_area_struct` records. You can view them in `/proc/<pid>/maps`.

- **Paging.** The hardware uses multi-level page tables to translate virtual addresses to physical frames. The kernel manages page table entries, handles page faults, and swaps pages to disk when necessary.
- **Memory Allocation System Calls.** `brk` / `sbrk` adjust the program break (heap), while `mmap` and `munmap` provide page-granularity allocation. `mprotect` changes access rights.
- **Copy-on-Write (COW).** After `fork`, the child shares the parent's pages. When either writes, the kernel duplicates the page, efficiently creating private copies.

An example of `mmap` for anonymous memory was provided in Chapter 33. Here we emphasize that `mmap` is the fundamental building block of all user-space memory management, and the C library's `malloc` often uses it internally.

Querying Memory Layout

A process can read its own memory map via `/proc/self/maps`. From NASM, you can open and read that pseudo-file, though it's simpler to use `cat /proc/<pid>/maps` externally.

48.4 Scheduler

The Linux scheduler is a complicated piece of kernel code, but its user-visible interface is simple:

- **Scheduling Policies:** `SCHED_OTHER` (default CFS), `SCHED_FIFO`, `SCHED_RR` (real-time). The policy can be set with `sched_setscheduler`.
- **Priority:** The nice value (-20 to +19) affects the share of CPU time for `SCHED_OTHER`. Real-time priorities are 1-99.
- **Affinity:** The `sched_setaffinity` system call binds a thread to specific CPUs, useful for cache optimization.

A simple yield can be performed with `sched_yield` as mentioned. No further examples are needed here; the scheduler is mostly transparent unless you explicitly change priorities.

48.5 Security Model

Linux security is based on the traditional Unix discretionary access control (DAC) model, augmented by capabilities. Each process has a real user ID (UID), effective UID, saved UID, and similarly for groups. The kernel checks permissions (read, write, execute) against the effective UID/GID and the file's mode bits. The root user (UID 0) bypasses most permission checks; otherwise, the permission bits are enforced.

Additionally, Linux supports *capabilities* (granular privileges like `CAP_SYS_PTRACE`) that can be granted to processes without full root. Capabilities are managed via system calls like `capget` / `capset`, though direct assembly usage is rare.

Retrieving User and Group IDs

The following program demonstrates how to obtain the real and effective UID using system calls.

Code: listing48-2.asm — Displaying UID

```
; listing48-2.asm
; Assemble: nasm -f elf64 -o listing48-2.o listing48-2.asm
; Link:     ld -o listing48-2 listing48-2.o

bits 64
default rel

SYS_GETUID  equ 102
SYS_GETEUID equ 107
SYS_EXIT    equ 60

section .text
global _start
_start:
    mov     eax, SYS_GETUID       ; real UID
    syscall
    mov     edi, eax              ; save real UID in edi for exit code (low 8 bits)

    mov     eax, SYS_GETEUID      ; effective UID
    syscall
    ; If you want to exit with the effective UID, uncomment the next line
    ; mov     edi, eax

    ; Exit with the real UID as exit code (just for demonstration)
    mov     eax, SYS_EXIT
    syscall
```

Run the program and check the exit status: `echo $?` will show the low byte of the real UID, which for a normal user is 1000. For root, it would be 0.

File Permission Check

When you call `open` with `O_RDONLY`, the kernel verifies that the process's effective UID has read permission on the file. If not, the call returns `-EACCES`. The permission bits are tested in sequence: owner, group, others. This is a classic Unix security model.

Privilege Escalation

Set-UID programs (the `setuid` permission bit) allow an executable to run with the privileges of its owner. This is used by `passwd` and `sudo`. From assembly, you can change the effective UID via the `setuid` system call (if privileged). However, writing setuid programs requires extreme caution.

Putting It All Together

Linux system internals form the foundation on which every assembly program runs. The clear separation of user and kernel mode, the standardized system call interface, the intricate memory manager, the fair scheduler, and the robust DAC security model provide a stable and powerful environment. By understanding these core components, you can write assembly code that not only works but also respects the security and performance principles of the OS. With the knowledge

from this chapter, you are well-equipped to tackle advanced topics such as signal handling, socket programming, and kernel module development, should you choose to explore further.

Part X

REAL WORLD PROJECTS

49

Linux Console Application

In This Chapter

This chapter walks through the design and implementation of a complete, non-trivial console application written entirely in NASM for Linux (Fedora 43). The program functions as a simple text-based calculator that reads commands from the user, performs arithmetic, and displays results. Beyond the arithmetic, the chapter demonstrates the architecture of a real assembly project: directory layout, entry point design, a reusable console I/O subsystem, a build process that ties multiple source files together, and a debugging workflow that ensures reliability. Every line of code uses Linux system calls directly; no C library is required (though linking with libc is optional for convenience). All examples follow the System V AMD64 ABI and have been tested with NASM 2.16.

49.1 Project Structure

A disciplined directory layout keeps the project manageable:

```
calc/
  src/
    main.asm          ; entry point and command loop
    console.asm       ; console I/O helpers
    strings.asm       ; string utility functions
    arithmetic.asm    ; arithmetic operations
  include/
    syscalls.inc      ; system call numbers and constants
  obj/               ; compiled object files
  bin/               ; final executable
```

```
Makefile            ; GNU makefile
```

- `src/` contains all assembly source. Each logical module is a separate file.
- `include/` holds shared constants (system call numbers, standard file descriptor numbers, etc.).
- `obj/` and `bin/` separate build artefacts.
- The `Makefile` automates the build with `make`.

This modular approach allows each component to be developed and tested independently.

49.2 Entry Point Design

The entry point for a pure Linux assembly program is `_start`. It does not receive command-line arguments via registers; they are on the stack as described in earlier chapters. For a calculator, we simply read from stdin and write to stdout. No special initialization is required because we use only system calls.

Code: main.asm — Entry point and command loop

```
; main.asm
bits 64
default rel

%include "include/syscalls.inc"

extern con_puts
extern con_gets
extern con_putnum

extern eval_expr
extern strcmp

section .data
prompt       db 'calc> ', 0
hello_msg    db 'Assembly Calculator v1.0. Type "quit" to exit.', 10, 0
quit_cmd     db 'quit', 0
bye_msg      db 'Goodbye.', 10, 0
unknown_cmd  db 'Unknown command.', 10, 0

section .bss
input_buf    resb 128

section .text
global _start
_start:
    ; Initialise console (nothing to do; we use fd 0/1 directly)
    ; Print welcome message
    lea     rdi, [rel hello_msg]
    call    con_puts

    ; Command loop
.command_loop:
```

```
    ; Print prompt
    lea     rdi, [rel prompt]
    call    con_puts

    ; Read a line of input
    lea     rdi, [rel input_buf]
    mov     esi, 128
    call    con_gets
    ; If con_gets returns NULL (error), exit
    test    rax, rax
    jz      .exit

    ; Check for empty line
    cmp     byte [rel input_buf], 0
    je      .command_loop

    ; Compare with "quit"
    lea     rdi, [rel input_buf]
    lea     rsi, [rel quit_cmd]
    call    strcmp
    test    eax, eax
    jz      .exit

    ; Attempt to evaluate expression
    lea     rdi, [rel input_buf]
    call    eval_expr
    jmp     .command_loop

.exit:
    lea     rdi, [rel bye_msg]
    call    con_puts

    ; exit(0)
    mov     eax, SYS_EXIT
    xor     edi, edi
    syscall
```

The entry point never touches file descriptors directly; it delegates to `con_puts` and `con_gets`.

49.3 Console I/O System

A dedicated module `console.asm` provides console I/O using the `read` and `write` system calls.

Code: console.asm — Console I/O functions

```
; console.asm
bits 64
default rel

%include "include/syscalls.inc"

section .text
```

```
; ------------------------------------------------------
; con\_puts: Write a null-terminated string to stdout.
; RDI = pointer to string
; ------------------------------------------------------
global con_puts
con_puts:
    push    rdi               ; save string pointer
    call    rts_strlen        ; returns length in RAX
    mov     rdx, rax          ; length
    pop     rsi               ; string buffer (source)
    mov     eax, SYS_WRITE
    mov     edi, STDOUT      ; fd=1
    syscall
    ret

; ------------------------------------------------------
; con\_gets: Read a line from stdin, null-terminate.
; RDI = buffer, ESI = buffer size
; Returns buffer pointer in RAX, or NULL on failure.
; ------------------------------------------------------
global con_gets
con_gets:
    push    rdi               ; save buffer
    ; read up to size-1 bytes to leave room for null
    dec     esi               ; max bytes to read
    mov     edx, esi          ; count
    mov     rsi, rdi          ; buffer
    mov     edi, STDIN        ; fd=0
    mov     eax, SYS_READ
    syscall
    ; check for error (negative) or EOF (0)
    test    rax, rax
    jle     .fail
    ; RAX = bytes read
    pop     rdi               ; buffer pointer
    mov     byte [rdi + rax], 0   ; null-terminate after data
    ; strip possible trailing newline (simple: overwrite last char if it's '\n')
    cmp     byte [rdi + rax - 1], 10
    jne     .done
    mov     byte [rdi + rax - 1], 0
.done:
    mov     rax, rdi
    ret
.fail:
    pop     rax
    xor     eax, eax
    ret
```

`con_gets` trims the trailing newline for convenience. The standard descriptors 0 and 1 are used directly.

49.4 Build Process

The Makefile assembles all sources and links them with `ld`.

Code: Makefile

```
NASM    = nasm
LD      = ld
OBJS    = obj/main.o obj/console.o obj/strings.o obj/arithmetic.o
TARGET  = bin/calc

$(TARGET): $(OBJS) | bin
^^I$(LD) -o $@ $^

obj/%.o: src/%.asm | obj
^^I$(NASM) -f elf64 -o $@ $<

obj:
^^Imkdir -p obj

bin:
^^Imkdir -p bin

clean:
^^Irm -rf obj bin
```

Build with `make`. The resulting executable `bin/calc` runs on any x86-64 Linux system.

49.5 Debugging Workflow

The standard debugger is GDB. Pass the executable to GDB, set breakpoints on labels, and step through. Use `objdump -d bin/calc` to view disassembly with symbols.

- Break at `_start`: `break _start`
- Break at `con_puts`: `break con_puts`
- Examine registers: `info registers`
- Check return values after system calls: `p/x $rax`
- Verify string contents: `x/s $rdi`

To check misalignment, before a `call` or `syscall`, check that `RSP` is a multiple of 16 if calling a C library function, but for direct system calls alignment is less strict. The memory console I/O uses only syscalls, so alignment faults are rare.

49.6 Final Integration Test

After building, run `./bin/calc` and test typical commands. Use `strace` to trace system calls and detect failures. The modular structure lets you reuse the console module in any future project.

Putting It All Together

The Linux console calculator demonstrates how to build a clean, multi-file assembly project using only system calls. The principles of modular design, clear entry point, reusable I/O, and automated builds form a foundation for any real-world assembly application on Fedora 43.

50

Assembly Utility Library on Linux

A well-crafted utility library saves time and reduces duplication. This chapter designs a general-purpose static library `libutils.a` for Linux, covering string, memory, and math utilities. All functions follow the System V AMD64 ABI and are built with NASM. The library is intended to be linked into any assembly (or C) project.

50.1 Library Architecture

The library follows the same modular structure as the Windows version, with a single public include file `libutils.inc` and separate source files.

Directory layout:

```
libutils/
  include/
    libutils.inc
  src/
    strings.asm
    memory.asm
    math.asm
  obj/
  lib/
  Makefile
```

50.2 Public Header (libutils.inc)

Code: libutils.inc

```
; libutils.inc
%ifndef LIBUTILS_INC
%define LIBUTILS_INC

; String utilities
extern utils_strlen          ; RAX = strlen(RDI: string)
extern utils_strcpy          ; RAX = strcpy(RDI: dest, RSI: src)
extern utils_strcmp          ; EAX = strcmp(RDI: s1, RSI: s2)
extern utils_strncpy         ; RAX = strncpy(RDI: dest, RSI: src, EDX: maxlen)
extern utils_strcat          ; RAX = strcat(RDI: dest, RSI: src) ; simplistic
extern utils_atoi            ; EAX = atoi(RDI: string)
extern utils_itoa            ; RAX = itoa(EDI: value, RSI: buffer)

; Memory utilities
extern utils_memcpy          ; RAX = memcpy(RDI: dest, RSI: src, RDX: size)
extern utils_memset          ; RAX = memset(RDI: ptr, SIL: val, RDX: size)
extern utils_memcmp          ; EAX = memcmp(RDI: p1, RSI: p2, RDX: size)

; Math utilities
extern utils_add64           ; RAX = add64(RDI: a, RSI: b)
extern utils_sub64           ; RAX = sub64(RDI: a, RSI: b)
extern utils_mul64           ; RAX = mul64(RDI: a, RSI: b) (truncated)
extern utils_div64           ; RAX = div64(RDI: dividend, RSI: divisor), remainder in RDX
extern utils_abs64           ; RAX = abs64(RDI: val)
extern utils_min64           ; RAX = min64(RDI: a, RSI: b)
extern utils_max64           ; RAX = max64(RDI: a, RSI: b)

%endif
```

50.3 Implementations

The implementations are similar to the Windows versions but with argument registers matching the System V ABI (RDI, RSI, RDX, RCX, R8, R9). I'll provide abbreviated examples.

String Utilities

Code: Excerpt from strings.asm

```
; strings.asm (Linux)
bits 64
default rel

section .text

global utils_strlen
utils_strlen:
    xor     eax, eax
.loop:
```

```
    cmp     byte [rdi + rax], 0
    je      .done
    inc     eax
    jmp     .loop
.done:
    ret

global utils_strcpy
utils_strcpy:
    push    rdi
    mov     rsi, rsi    ; source (already in RSI)
    mov     rdi, rdi    ; dest (already in RDI)
    cld
.loop:
    lodsb
    stosb
    test    al, al
    jnz     .loop
    pop     rax
    ret

global utils_strcmp
utils_strcmp:
.loop:
    movzx   eax, byte [rdi]
    movzx   ecx, byte [rsi]
    cmp     eax, ecx
    jne     .diff
    test    eax, eax
    jz      .equal
    inc     rdi
    inc     rsi
    jmp     .loop
.diff:
    sub     eax, ecx
    ret
.equal:
    xor     eax, eax
    ret
```

Memory Utilities

Code: Excerpt from memory.asm

```
global utils_memcpy
utils_memcpy:
    push    rdi
    cld
    mov     rsi, rsi
    mov     rdi, rdi
    mov     rcx, rdx
    rep     movsb
    pop     rax
```

```
    ret

global utils_memset
utils_memset:
    push    rdi
    mov     al, sil
    mov     rcx, rdx
    rep     stosb
    pop     rax
    ret
```

Math Utilities

Code: Excerpt from math.asm

```
global utils_add64
utils_add64:
    lea     rax, [rdi + rsi]
    ret

global utils_div64
utils_div64:
    mov     rax, rdi
    cqo
    idiv    rsi       ; quotient in rax, remainder in rdx
    ret
```

50.4 Build and Integration

A `Makefile` builds the static library:

Code: Makefile for library

```
NASM = nasm
AR   = ar
RANLIB = ranlib
OBJS = obj/strings.o obj/memory.o obj/math.o
TARGET = lib/libutils.a

$(TARGET): $(OBJS)
^^I$(AR) rcs $@ $^
^^I$(RANLIB) $@

obj/%.o: src/%.asm
^^Imkdir -p obj
^^I$(NASM) -f elf64 -o $@ $<

.PHONY: clean
clean:
^^Irm -rf obj lib
```

Run `make` to produce `lib/libutils.a`.

Using the Library in a Project

Link the library with `ld` or `gcc`:

Code: Compiling a program with the library

```
nasm -f elf64 -o prog.o prog.asm
ld -o prog prog.o -L./lib -lutils
```

Alternatively, you can link the individual object files directly.

Putting It All Together

The Linux utility library provides a solid foundation for building larger assembly projects. Its modular design, clear interface, and adherence to the AMD64 ABI make it a reliable component in any developer's toolkit. The next chapters will build upon these concepts to create even more complex applications.

Part XI

REAL WORLD PROJECTS

51

Linux Console Application

In This Chapter

This chapter walks through the design and implementation of a complete, non-trivial console application written entirely in NASM for Linux (Fedora 43). The program functions as a simple text-based calculator that reads commands from the user, performs arithmetic, and displays results. Beyond the arithmetic, the chapter demonstrates the architecture of a real assembly project: directory layout, entry point design, a reusable console I/O subsystem, a build process that ties multiple source files together, and a debugging workflow that ensures reliability. Every line of code uses Linux system calls directly; no C library is required. All examples follow the System V AMD64 ABI and have been tested with NASM 2.16.

51.1 Project Structure

A disciplined directory layout keeps the project manageable:

```
calc/
  src/
    main.asm          ; entry point and command loop
    console.asm       ; console I/O helpers
    strings.asm       ; string utility functions
    arithmetic.asm    ; arithmetic operations
  include/
    syscalls.inc      ; system call numbers and constants
  obj/               ; compiled object files
  bin/               ; final executable
  Makefile           ; GNU makefile
```

- `src/` contains all assembly source. Each logical module is a separate file. The main file orchestrates the others.
- `include/` holds shared constants (system call numbers, standard file descriptor numbers).
- `obj/` and `bin/` separate intermediate and final build artefacts.
- The `Makefile` automates the build with `make`.

This modular approach allows each component to be developed and tested independently.

51.2 Entry Point Design

The entry point for a pure Linux assembly program is `_start`. It does not receive command-line arguments via registers; they are on the stack as described in earlier chapters. For our calculator, we simply read from stdin and write to stdout. No special initialisation is required because we use only system calls.

Code: main.asm — Entry point and command loop

```
; main.asm
bits 64
default rel

%include "include/syscalls.inc"

extern con_puts
extern con_gets
extern con_putnum

extern eval_expr
extern strcmp

section .data
prompt       db 'calc> ', 0
hello_msg    db 'Assembly Calculator v1.0. Type "quit" to exit.', 10, 0
quit_cmd     db 'quit', 0
bye_msg      db 'Goodbye.', 10, 0
unknown_cmd  db 'Unknown command.', 10, 0

section .bss
input_buf    resb 128

section .text
global _start
_start:
    ; Print welcome message
    lea     rdi, [rel hello_msg]
    call    con_puts

    ; Command loop
.command_loop:
    ; Print prompt
    lea     rdi, [rel prompt]
```

```
    call    con_puts

    ; Read a line of input
    lea     rdi, [rel input_buf]
    mov     esi, 128
    call    con_gets
    ; If con_gets returns NULL (error), exit
    test    rax, rax
    jz      .exit

    ; Check for empty line (just newline)
    cmp     byte [rel input_buf], 0
    je      .command_loop

    ; Compare with "quit"
    lea     rdi, [rel input_buf]
    lea     rsi, [rel quit_cmd]
    call    strcmp
    test    eax, eax
    jz      .exit

    ; Attempt to evaluate expression
    lea     rdi, [rel input_buf]
    call    eval_expr
    ; If evaluation succeeds, con_putnum is called inside; else error.
    jmp     .command_loop

.exit:
    lea     rdi, [rel bye_msg]
    call    con_puts

    ; exit(0)
    mov     eax, SYS_EXIT
    xor     edi, edi
    syscall
```

The entry point never touches file descriptors directly; it delegates to `con_puts` and `con_gets`.

51.3 Console I/O System

A dedicated module `console.asm` provides a simple but robust console API using the `write` and `read` system calls.

Code: console.asm — Console I/O functions

```
; console.asm
bits 64
default rel

%include "include/syscalls.inc"

section .text
```

```
; ------------------------------------------------------
; con\_puts: Write a null-terminated string to stdout.
; RDI = pointer to string
; ------------------------------------------------------
global con_puts
con_puts:
    push    rdi             ; save string pointer
    call    rts_strlen      ; returns length in RAX
    mov     rdx, rax        ; length
    pop     rsi             ; string buffer (source)
    mov     eax, SYS_WRITE
    mov     edi, STDOUT     ; fd = 1
    syscall
    ret

; ------------------------------------------------------
; con\_gets: Read a line from stdin, null-terminate.
; RDI = buffer, ESI = buffer size
; Returns buffer pointer in RAX, or NULL on failure.
; ------------------------------------------------------
global con_gets
con_gets:
    push    rdi             ; save buffer
    ; read up to size-1 bytes to leave room for null
    dec     esi             ; max bytes to read
    mov     edx, esi        ; count
    mov     rsi, rdi        ; buffer
    mov     edi, STDIN      ; fd = 0
    mov     eax, SYS_READ
    syscall
    ; check for error (negative) or EOF (0)
    test    rax, rax
    jle     .fail
    ; RAX = bytes read
    pop     rdi             ; buffer pointer
    mov     byte [rdi + rax], 0   ; null-terminate after data
    ; strip possible trailing newline (simple: overwrite last char if it's '\n')
    cmp     byte [rdi + rax - 1], 10
    jne     .done
    mov     byte [rdi + rax - 1], 0
.done:
    mov     rax, rdi
    ret
.fail:
    pop     rax
    xor     eax, eax
    ret
```

The module exports `con_init`, `con_puts`, and `con_gets`. It relies on an external `rts_strlen` function from `strings.asm` for string length computation.

51.4 String Utilities

Code: strings.asm — String length and comparison

```
; strings.asm
bits 64
default rel

section .text

global rts_strlen
rts_strlen:
    xor     eax, eax
.loop:
    cmp     byte [rdi + rax], 0
    je      .done
    inc     eax
    jmp     .loop
.done:
    ret

global strcmp
strcmp:
.loop:
    movzx   eax, byte [rdi]
    movzx   ecx, byte [rsi]
    cmp     eax, ecx
    jne     .diff
    test    eax, eax
    jz      .equal
    inc     rdi
    inc     rsi
    jmp     .loop
.diff:
    sub     eax, ecx
    ret
.equal:
    xor     eax, eax
    ret
```

51.5 Arithmetic Evaluation

The `arithmetic.asm` module provides a simple expression evaluator that understands two-operand expressions like `3 + 5`. It parses the input, performs the operation, and prints the result.

Code: arithmetic.asm — Simple expression evaluation

```
; arithmetic.asm
bits 64
default rel

%include "include/syscalls.inc"
```

```
extern rts_strlen, con_puts, con_putnum

section .data
err_msg db 'Error: invalid expression.', 10, 0

section .text

; eval_expr(RDI: string) - prints result or error.
global eval_expr
eval_expr:
    push    rbp
    mov     rbp, rsp
    sub     rsp, 32
    ; parse first number
    call    parse_int       ; returns EAX = number, RDI updated
    mov     r12d, eax        ; first operand
    ; skip whitespace
    call    skip_space
    ; read operator
    movzx   r8d, byte [rdi]
    cmp     r8d, '+'
    je      .op_add
    cmp     r8d, '-'
    je      .op_sub
    cmp     r8d, '*'
    je      .op_mul
    cmp     r8d, '/'
    je      .op_div
    jmp     .error
.op_add:
    inc     rdi
    ; parse second number
    call    parse_int
    add     eax, r12d
    jmp     .print
.op_sub:
    inc     rdi
    call    parse_int
    sub     r12d, eax          ; r12 = first - second
    mov     eax, r12d
    jmp     .print
.op_mul:
    inc     rdi
    call    parse_int
    imul    eax, r12d
    jmp     .print
.op_div:
    inc     rdi
    call    parse_int
    ; check for zero divisor
    test    eax, eax
    jz      .error
    mov     ecx, eax
```

```
    mov     eax, r12d
    cdq
    idiv    ecx
    jmp     .print

.print:
    mov     edi, eax
    call    con_putnum
    leave
    ret
.error:
    lea     rdi, [rel err_msg]
    call    con_puts
    leave
    ret

; parse_int(RDI: string ptr) - updates RDI, returns integer in EAX
parse_int:
    xor     eax, eax
    xor     ecx, ecx          ; sign flag (0 positive, 1 negative)
    ; skip leading whitespace
    call    skip_space
    cmp     byte [rdi], '-'
    jne     .not_neg
    mov     ecx, 1
    inc     rdi
    jmp     .loop
.not_neg:
    cmp     byte [rdi], '+'
    jne     .loop
    inc     rdi
.loop:
    movzx   edx, byte [rdi]
    cmp     edx, '0'
    jb      .done
    cmp     edx, '9'
    ja      .done
    sub     edx, '0'
    imul    eax, eax, 10
    add     eax, edx
    inc     rdi
    jmp     .loop
.done:
    test    ecx, ecx
    jz      .exit
    neg     eax
.exit:
    ret

skip_space:
    cmp     byte [rdi], ' '
    jne     .ret
    inc     rdi
    jmp     skip_space
```

```
.ret:
    ret
```

51.6 Build Process

The Makefile assembles all sources and links them with `ld`.

Code: Makefile

```
NASM    = nasm
LD      = ld
OBJS    = obj/main.o obj/console.o obj/strings.o obj/arithmetic.o
TARGET  = bin/calc

$(TARGET): $(OBJS) | bin
^^I$(LD) -o $@ $^

obj/%.o: src/%.asm | obj
^^I$(NASM) -f elf64 -o $@ $<

obj:
^^Imkdir -p obj

bin:
^^Imkdir -p bin

clean:
^^Irm -rf obj bin
```

Build with `make`. The resulting executable `bin/calc` runs on any x86-64 Linux system.

51.7 Debugging Workflow

Use GDB to debug the calculator. Set breakpoints on `_start`, `con_puts`, or `eval_expr`. Check registers with `info registers`, examine memory with `x/s $rdi`, and step through with `stepi`. Since we link with `ld` and no debug info is included, assemble with `nasm -f elf64 -g` to include DWARF symbols for better source-level debugging.

Final Integration Test

After building, run `./bin/calc` and test typical arithmetic expressions. Use `strace` to trace system calls if needed. The modular structure lets you reuse the console module in any future project.

Putting It All Together

The Linux console calculator demonstrates how to build a clean, multi-file assembly project using only system calls. The principles of modular design, clear entry point, reusable I/O, and automated builds form a foundation for any real-world assembly application on Fedora 43.

52

Assembly Utility Library on Linux

In This Chapter

A well-crafted utility library saves time and reduces duplication. This chapter designs a general-purpose static library `libutils.a` for Linux, covering string, memory, and math utilities. All functions follow the System V AMD64 ABI and are built with NASM. The library is intended to be linked into any assembly (or C) project.

52.1 Library Architecture

The library follows a clean modular structure with a single public include file `libutils.inc` and separate source files.

```
libutils/
  include/
    libutils.inc          ; Public NASM interface
  src/
    strings.asm
    memory.asm
    math.asm
  obj/
  lib/                    ; final libutils.a
  Makefile
```

52.2 Public Header (libutils.inc)

Code: libutils.inc

```
; libutils.inc
%ifndef LIBUTILS_INC
%define LIBUTILS_INC

; String utilities
extern utils_strlen          ; RAX = strlen(RDI: string)
extern utils_strcpy          ; RAX = strcpy(RDI: dest, RSI: src)
extern utils_strcmp          ; EAX = strcmp(RDI: s1, RSI: s2)
extern utils_strncpy         ; RAX = strncpy(RDI: dest, RSI: src, EDX: maxlen)
extern utils_strcat          ; RAX = strcat(RDI: dest, RSI: src) (simplistic)
extern utils_atoi            ; EAX = atoi(RDI: string)
extern utils_itoa            ; RAX = itoa(EDI: value, RSI: buffer)

; Memory utilities
extern utils_memcpy          ; RAX = memcpy(RDI: dest, RSI: src, RDX: size)
extern utils_memset          ; RAX = memset(RDI: ptr, SIL: val, RDX: size)
extern utils_memcmp          ; EAX = memcmp(RDI: p1, RSI: p2, RDX: size)

; Math utilities
extern utils_add64           ; RAX = add64(RDI: a, RSI: b)
extern utils_sub64           ; RAX = sub64(RDI: a, RSI: b)
extern utils_mul64           ; RAX = mul64(RDI: a, RSI: b)  (truncated)
extern utils_div64           ; RAX = div64(RDI: dividend, RSI: divisor), remainder in RDX
extern utils_abs64           ; RAX = abs64(RDI: val)
extern utils_min64           ; RAX = min64(RDI: a, RSI: b)
extern utils_max64           ; RAX = max64(RDI: a, RSI: b)

%endif
```

52.3 Implementations

All functions respect the System V AMD64 calling convention.

String Utilities

Code: strings.asm (excerpt)

```
; strings.asm
bits 64
default rel

section .text

global utils_strlen
utils_strlen:
    xor     eax, eax
.loop:
    cmp     byte [rdi + rax], 0
```

```
    je      .done
    inc     eax
    jmp     .loop
.done:
    ret

global utils_strcpy
utils_strcpy:
    push    rdi
    cld
.loop:
    lodsb
    stosb
    test    al, al
    jnz     .loop
    pop     rax
    ret

global utils_strcmp
utils_strcmp:
.loop:
    movzx   eax, byte [rdi]
    movzx   ecx, byte [rsi]
    cmp     eax, ecx
    jne     .diff
    test    eax, eax
    jz      .equal
    inc     rdi
    inc     rsi
    jmp     .loop
.diff:
    sub     eax, ecx
    ret
.equal:
    xor     eax, eax
    ret
```

Memory Utilities

Code: memory.asm (excerpt)

```
; memory.asm
bits 64
default rel

section .text

global utils_memcpy
utils_memcpy:
    push    rdi
    cld
    mov     rsi, rsi
    mov     rdi, rdi
```

```
    mov     rcx, rdx
    rep     movsb
    pop     rax
    ret

global utils_memset
utils_memset:
    push    rdi
    mov     al, sil
    mov     rcx, rdx
    rep     stosb
    pop     rax
    ret
```

Math Utilities

Code: math.asm (excerpt)

```
; math.asm
bits 64
default rel

section .text

global utils_add64
utils_add64:
    lea     rax, [rdi + rsi]
    ret

global utils_div64
utils_div64:
    mov     rax, rdi
    cqo
    idiv    rsi       ; quotient in rax, remainder in rdx
    ret
```

52.4 Build and Integration

A `Makefile` builds the static library `libutils.a`:

Code: Makefile for library

```
NASM = nasm
AR   = ar
RANLIB = ranlib
OBJS = obj/strings.o obj/memory.o obj/math.o
TARGET = lib/libutils.a

$(TARGET): $(OBJS)
^^I$(AR) rcs $@ $^
^^I$(RANLIB) $@
```

```
obj/%.o: src/%.asm
^^Imkdir -p obj
^^I$(NASM) -f elf64 -o $@ $<

.PHONY: clean
clean:
^^Irm -rf obj lib
```

Run `make` to produce `lib/libutils.a`.

Using the Library

Link the library with `ld` or `gcc`:

Code: Compiling a program with the library

```
nasm -f elf64 -o prog.o prog.asm
ld -o prog prog.o -L./lib -lutils
```

The library is a reliable component for any Linux assembly project.

53

Mini Runtime System on Linux

In This Chapter

A runtime system provides the scaffolding between the kernel loader and application logic. On Linux, a mini runtime written in pure NASM can offer a clean environment without depending on the C library. This chapter designs and implements such a runtime: an initialisation layer that parses command-line arguments, a set of core services (console I/O, string conversion), a memory management layer using simple `mmap`-based allocation, and an execution model that calls a user-defined `user_main` function. The final result is a reusable runtime that you link with any NASM application to provide C-like convenience without bloat.

53.1 Runtime Design

Key design goals:

- **Miniature.** The runtime provides only essential services: console output, minimal input, argument parsing, and memory allocation.
- **Pure assembly.** No dependency on libc; all functions use Linux system calls directly.
- **Modular.** Separated into entry point, services, and utilities.
- **User entry.** The application defines `user_main(void)` and returns an exit code.

53.2 Initialization Layer

The entry point is `_start`. It extracts `argc` and `argv` from the stack, stores them in global variables, and calls `user_main`.

Code: rt_entry.asm — Runtime entry point

```
; rt_entry.asm
bits 64
default rel

SYS_EXIT equ 60

section .data
; Exported globals
global __argc
global __argv

__argc dq 0
__argv dq 0

section .text
global _start
_start:
    ; Stack: [rsp] = argc, then argv pointers.
    pop     rax               ; argc
    mov     [rel __argc], rax
    mov     [rel __argv], rsp  ; rsp now points to argv[0]

    ; Call user main
    extern user_main
    call    user_main

    ; Exit with return value in EAX
    mov     edi, eax
    mov     eax, SYS_EXIT
    syscall
```

The application writes `user_main` and can access the arguments via the globals.

53.3 Core Services

The runtime provides console output and input using system calls, and a small string utility.

Console Output: `rt_puts`

Writes a null-terminated string to stdout.

Code: rt_services.asm — rt_puts and rt_gets (simplified)

```
; rt_services.asm
bits 64
default rel

SYS_WRITE equ 1
SYS_READ  equ 0
STDOUT    equ 1
STDIN     equ 0
```

```
section .text

; void rt_puts(const char *str)
global rt_puts
rt_puts:
    push    rdi
    ; compute length (using external rt_strlen)
    call    rt_strlen
    mov     rdx, rax          ; length
    pop     rsi               ; string address
    mov     edi, STDOUT
    mov     eax, SYS_WRITE
    syscall
    ret

; char *rt_gets(char *buf, size_t bufsize)
global rt_gets
rt_gets:
    push    rdi
    dec     esi               ; leave space for null
    mov     edx, esi
    mov     rsi, rdi
    mov     edi, STDIN
    mov     eax, SYS_READ
    syscall
    test    rax, rax
    jle     .fail
    pop     rdi
    mov     byte [rdi + rax], 0
    ; strip newline
    cmp     byte [rdi + rax - 1], 10
    jne     .done
    mov     byte [rdi + rax - 1], 0
.done:
    mov     rax, rdi
    ret
.fail:
    pop     rax
    xor     eax, eax
    ret
```

String Length

Code: rt_strings.asm

```
; rt_strings.asm
bits 64
default rel

section .text
global rt_strlen
rt_strlen:
```

```
    xor     eax, eax
.loop:
    cmp     byte [rdi + rax], 0
    je      .done
    inc     eax
    jmp     .loop
.done:
    ret
```

Integer to Decimal String

Code: rt_itoa implementation

```
global rt_itoa
rt_itoa:
    ; EDI = value, RSI = buffer
    push    rdi
    mov     eax, edi
    test    eax, eax
    jns     .do_convert
    mov     byte [rsi], '-'
    inc     rsi
    neg     eax
.do_convert:
    sub     rsp, 32           ; temporary stack buffer for reverse digits
    mov     rcx, rsp
    mov     r10, rcx
    mov     r9d, 10
.next_digit:
    xor     edx, edx
    div     r9d
    add     dl, '0'
    mov     [rcx], dl
    inc     rcx
    test    eax, eax
    jnz     .next_digit
.copy_rev:
    cmp     rcx, r10
    jbe     .done_copy
    dec     rcx
    mov     dl, [rcx]
    mov     [rsi], dl
    inc     rsi
    jmp     .copy_rev
.done_copy:
    mov     byte [rsi], 0
    add     rsp, 32
    pop     rdi
    mov     rax, rdi          ; return original buffer
    ret
```

53.4 Memory Management

A simple memory allocator uses the `mmap` system call to allocate pages. Freeing is done with `munmap`. The runtime provides `rt_malloc` and `rt_free`. (A more sophisticated heap with reuse can be added later.)

Code: rt_memory.asm

```
; rt_memory.asm
bits 64
default rel

SYS_MMAP   equ 9
SYS_MUNMAP equ 11
PROT_READ  equ 1
PROT_WRITE equ 2
MAP_PRIVATE equ 0x02
MAP_ANONYMOUS equ 0x20

section .text

; void *rt_malloc(size_t size)
global rt_malloc
rt_malloc:
    push    rbp
    mov     rbp, rsp
    sub     rsp, 16
    ; mmap(NULL, size, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0)
    xor     edi, edi
    mov     rsi, rdi            ; size already in RDI? Wait, size is passed in RDI.
    ; Our first arg is in RDI already.
    mov     edx, PROT_READ | PROT_WRITE
    mov     r10d, MAP_PRIVATE | MAP_ANONYMOUS
    mov     r8d, -1
    xor     r9d, r9d
    mov     eax, SYS_MMAP
    syscall
    cmp     rax, -1
    ja      .ok
    xor     eax, eax
.ok:
    leave
    ret

; void rt_free(void *ptr, size_t size)
global rt_free
rt_free:
    mov     rdi, rdi            ; ptr
    mov     rsi, rsi            ; size (must be provided)
    mov     eax, SYS_MUNMAP
    syscall
    ret
```

The user must track the allocation size to free. A more advanced allocator could store the size in a header.

53.5 Execution Model

The application defines `user_main` as a global function with no arguments, returning an integer. It can use the runtime services and the global `__argc`, `__argv`. The runtime does not provide any C-style format functions; instead, the user can call `rt_puts` and `rt_itoa` to build output.

Example Application

Code: hello_rt.asm

```
; hello_rt.asm
bits 64
default rel

extern rt_puts
extern rt_malloc
extern rt_free

section .data
msg db 'Hello from mini runtime!', 10, 0
section .text
global user_main
user_main:
    lea     rdi, [rel msg]
    call    rt_puts

    mov     edi, 128
    call    rt_malloc
    test    rax, rax
    jz      .exit
    mov     rdi, rax
    mov     esi, 128
    call    rt_free

    xor     eax, eax
.exit:
    ret
```

Build and link: assemble all runtime objects and the application into one executable.

Code: Build command

```
nasm -f elf64 -o rt_entry.o rt_entry.asm
nasm -f elf64 -o rt_services.o rt_services.asm
nasm -f elf64 -o rt_strings.o rt_strings.asm
nasm -f elf64 -o rt_memory.o rt_memory.asm
nasm -f elf64 -o hello_rt.o hello_rt.asm
ld -o hello rt_entry.o rt_services.o rt_strings.o rt_memory.o hello_rt.o
```

This produces a static, self-contained executable. The mini runtime is a perfect starting point for embedded Linux tools, educational projects, and performance-sensitive utilities.

54

Multi-Module Assembly Projects on Linux

In This Chapter

As an assembly project grows beyond a few hundred lines, it becomes essential to split the code into multiple source files. This chapter addresses the practical aspects of managing a NASM project with several modules on Linux. You will learn how to organise the project directory, design a build system using `Makefiles`, link multiple objects into a single executable, manage dependencies between modules, and adopt a scaling strategy that keeps complexity under control. All techniques are grounded in the NASM 2.16 manual, the GNU linker documentation, and real-world development experience. Complete build scripts and example modules are provided.

54.1 Project Structure

A recommended layout for a multi-module assembly project on Linux:

```
project/
  src/
    main.asm          ; entry point
    console.asm       ; console I/O module
    engine.asm        ; core logic
    math/
      vector.asm      ; math submodule
      matrix.asm      ; math submodule
  include/
    common.inc        ; shared constants and macros
```

```
  exports.inc       ; extern declarations for public API
obj/                ; generated object files
bin/                ; final executable
lib/                ; static libraries (if any)
Makefile            ; GNU make
```

Each `.asm` file corresponds to a logical module. The `include` directory holds files that are included with `%include` but are not compiled separately. All modules that need common constants (like system call numbers) can `%include "common.inc"`.

Code: common.inc — shared declarations

```
; common.inc
%ifndef COMMON_INC
%define COMMON_INC

SYS_WRITE equ 1
SYS_READ  equ 0
SYS_EXIT  equ 60
STDOUT    equ 1
STDIN     equ 0

%endif
```

54.2 Build System Design

Using GNU Make, a `Makefile` can automatically track dependencies and rebuild only what has changed.

Simple Makefile

Code: Makefile

```
NASM    = nasm
LD      = ld
INCDIR  = include
OBJDIR  = obj
BINDIR  = bin
SRCDIR  = src

SRCS    = $(SRCDIR)/main.asm $(SRCDIR)/console.asm $(SRCDIR)/engine.asm \
          $(SRCDIR)/math/vector.asm $(SRCDIR)/math/matrix.asm
OBJS    = $(patsubst $(SRCDIR)/%.asm, $(OBJDIR)/%.o, $(SRCS))
TARGET  = $(BINDIR)/app

.PHONY: all clean

all: $(TARGET)

$(TARGET): $(OBJS) | $(BINDIR)
^^I$(LD) -o $@ $^
```

```
$(OBJDIR)/%.o: $(SRCDIR)/%.asm | $(OBJDIR)
^^I@mkdir -p $(dir $@)
^^I$(NASM) -I$(INCDIR) -f elf64 -o $@ $<

$(OBJDIR):
^^Imkdir -p $@

$(BINDIR):
^^Imkdir -p $@

clean:
^^Irm -rf $(OBJDIR) $(BINDIR)
```

With this `Makefile`, running `make` assembles only the modified files and links everything.

Adding Dependency on Include Files

To rebuild whenever an include file changes, add it as a prerequisite of the objects.

Code: Dependency rule

```
obj/main.o: src/main.asm include/common.inc
^^I$(NASM) -Iinclude -f elf64 -o $@ $<
```

In a larger project, you can generate dependency files using a script that scans for `%include` directives, but manual listing is sufficient for moderate sizes.

54.3 Linking Multiple Objects

The linker (`ld`) accepts multiple object files. Symbols declared `global` in one module and `extern` in another are resolved automatically. All object files must be placed on the command line; the order generally does not matter for static linking.

Cross-Module Example

Module `vector.asm` exports `vec_add`:

Code: vector.asm

```
; vector.asm
bits 64
default rel
section .text
global vec_add
vec_add:
    ; RDI = a, RSI = b, EDX = count
    push    rbx
.loop:
    mov     rax, [rdi]
    add     rax, [rsi]
    mov     [rdi], rax
```

```
    add     rdi, 8
    add     rsi, 8
    dec     edx
    jnz     .loop
    pop     rbx
    ret
```

In `main.asm`:

Code: main.asm

```
extern vec_add
section .text
global _start
_start:
    ; ... call vec_add ...
```

The linker will resolve the call.

Sharing Data

Global variables can be shared by marking them `global` in one module and `extern` in others. Use RIP-relative addressing with `default rel` as always.

54.4 Dependency Management

Avoid circular dependencies. Use a common utility module for shared functions. For instance, a `strings.asm` module contains `strlen`, `strcmp`; other modules `extern` these instead of duplicating them. The dependency graph should be a DAG.

Include Files

Since NASM supports `%include`, place all shared `extern` declarations and constant definitions in include files. Use include guards (`%ifndef`) to prevent multiple inclusions.

54.5 Scaling Strategy

For larger projects, group related modules into subdirectories and build them as separate static libraries. Use `ar` to create `.a` files, then link the final executable against them. This reduces the number of objects on the final link line and improves encapsulation.

Code: Creating a static library for a module group

```
nasm -f elf64 -o obj/math_vector.o src/math/vector.asm
nasm -f elf64 -o obj/math_matrix.o src/math/matrix.asm
ar rcs lib/libmath.a obj/math_vector.o obj/math_matrix.o
```

Then the final link step:

```
ld -o bin/app obj/main.o obj/console.o -L./lib -lmath
```

Consistent Naming

Use a clear naming convention to prevent symbol conflicts. Prefix all global symbols with the module name, e.g., `io_readfile`, `math_vec_add`. This is the assembly equivalent of namespaces.

Continuous Integration

Set up a CI pipeline using GitHub Actions or a local script. The workflow assembles all source files, runs tests, and archives the binary. A simple example for GitHub Actions (on `ubuntu-latest`):

Code: .github/workflows/ci.yml

```yaml
name: Build NASM Project
on: [push, pull_request]
jobs:
  build:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4
      - name: Install NASM
        run: sudo apt-get update && sudo apt-get install -y nasm
      - name: Build
        run: make
      - name: Run tests
        run: ./bin/test_suite || exit 1
      - uses: actions/upload-artifact@v4
        with:
          name: binary
          path: bin/app
```

With these practices, your NASM projects can scale from a single file to thousands of lines across dozens of modules while remaining maintainable.

55

Build Automation on Linux

In This Chapter

Manual assembly and linking commands quickly become tedious. Build automation replaces them with scripts and configuration files that ensure consistent, reproducible builds. This chapter covers the use of shell scripts, `Makefiles` for dependency-driven builds, techniques for automating the linking of multiple objects and libraries, the creation of debug and release build configurations, and the integration of continuous integration (CI) into assembly projects on Linux. All examples are based on NASM 2.16 and tested on Fedora 43.

55.1 Shell Scripts

The simplest automation tool is a shell script. It can assemble all source files and link the final executable. Here is a minimal build script for a single-module project.

Code: build.sh — Minimal script

```
#!/bin/bash
set -e
NASM="nasm"
LD="ld"

mkdir -p obj bin

echo "Assembling..."
$NASM -f elf64 -o obj/main.o src/main.asm

echo "Linking..."
$LD -o bin/app obj/main.o
```

```
echo "Build complete: bin/app"
```

Make it executable with `chmod +x build.sh` and run with `./build.sh`. The script stops immediately on any error because of `set -e`.

Multi-Module Script

For several source files, list each one:

Code: build.sh — Multi-module

```
#!/bin/bash
set -e
NASM="nasm"
LD="ld"
SRC="src"
OBJ="obj"
BIN="bin"

mkdir -p $OBJ $BIN

$NASM -f elf64 -o $OBJ/main.o $SRC/main.asm
$NASM -f elf64 -o $OBJ/console.o $SRC/console.asm
$NASM -f elf64 -o $OBJ/engine.o $SRC/engine.asm

$LD -o $BIN/app $OBJ/main.o $OBJ/console.o $OBJ/engine.o

echo "Build complete"
```

Debug and Release in Shell Scripts

Pass an argument to control the build type.

Code: build.sh with debug flag

```
#!/bin/bash
set -e
NASM_FLAGS="-f elf64"
BUILD_TYPE="release"
if [ "|$1|" == "debug" ]; then
    NASM_FLAGS="$NASM_FLAGS -g -DDEBUG"
    BUILD_TYPE="debug"
fi
echo "Building $BUILD_TYPE..."
nasm $NASM_FLAGS -o obj/main.o src/main.asm
# ... continue
```

55.2 Makefiles

Makefiles are superior to shell scripts for larger projects because they only rebuild files whose sources have changed.

Basic Makefile

Code: Makefile

```
NASM    = nasm
LD      = ld
OBJS    = obj/main.o obj/console.o obj/engine.o
TARGET  = bin/app

$(TARGET): $(OBJS)
^^I$(LD) -o $@ $^

obj/%.o: src/%.asm
^^I$(NASM) -f elf64 -o $@ $<

.PHONY: clean
clean:
^^Irm -rf obj bin
```

Debug and Release with Make

You can define variables for different build configurations and override them on the command line.

Code: Makefile with debug support

```
DEBUG ?= 0
NASM_FLAGS = -f elf64
ifeq ($(DEBUG),1)
    NASM_FLAGS += -g -DDEBUG
endif

obj/%.o: src/%.asm
^^I$(NASM) $(NASM_FLAGS) -o $@ $<
```

Run `make DEBUG=1` for a debug build.

Adding Clean Target

The clean target removes generated files. It is essential for resetting the build.

```
clean:
^^Irm -rf obj bin
```

55.3 Automated Linking

When the number of object files grows, you can use wildcards in the Makefile to automatically discover source files.

Code: Auto-discovery with wildcard

```
SRCS := $(wildcard src/*.asm) $(wildcard src/**/*.asm)
OBJS := $(patsubst src/%.asm, obj/%.o, $(SRCS))
```

Then the linking rule becomes:

```
$(TARGET): $(OBJS)
^^I$(LD) -o $@ $^
```

55.4 Debug and Release Builds

The key differences between debug and release builds in assembly are:

- **Debug:** Include DWARF debug info with `-g`, define `DEBUG` symbol, avoid heavy optimization that makes debugging hard, keep frame pointers.
- **Release:** Omit debug info, define `NDEBUG`, apply small code size optimizations (e.g., `xor eax, eax` vs `mov eax, 0`), may use `strip` to remove symbols at the end.

Conditional assembly with `%ifdef DEBUG` allows code to differ.

55.5 CI Workflow Concept

Continuous Integration automates building and testing. A common setup uses GitHub Actions on an Ubuntu runner.

Code: .github/workflows/ci.yml

```yaml
name: NASM CI

on: [push, pull_request]

jobs:
  build:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4
      - name: Install NASM
        run: sudo apt-get update && sudo apt-get install -y nasm
      - name: Build
        run: make DEBUG=0
      - name: Run tests
        run: ./bin/test_runner
      - uses: actions/upload-artifact@v4
        with:
          name: artifact
          path: bin/app
```

This workflow ensures every commit compiles and passes tests.

Local Automation with Cron

For nightly builds, you can use a cron job. Add a script:

Code: nightly.sh

```
#!/bin/bash
cd /your/project
make clean && make DEBUG=0
if [ $? -ne 0 ]; then
    echo "Build failed at $(date)" >> build.log
fi
```

Then schedule with `crontab -e`:

Code: crontab entry

```
0 2 * * * /your/project/nightly.sh
```

Putting It All Together

Build automation is an investment that pays for itself immediately. By using shell scripts for quick prototypes, `Makefiles` for dependency tracking, debug/release flags for controlled builds, and CI for quality assurance, you turn your NASM development into a professional, repeatable process. The knowledge you've gained throughout this book—from the simplest instruction encoding to the creation of a custom runtime—is now supported by a robust build infrastructure that will serve you well in every assembly project on Linux.

Final

Appendices

Appendix A: x86-64 Instruction Reference

This appendix lists the most frequently used x86-64 instructions with their operands, brief description, and effect on the condition flags. All instructions assume 64-bit mode and Intel NASM syntax. The flags column indicates which flags are modified: CF (Carry), ZF (Zero), SF (Sign), OF (Overflow), PF (Parity), AF (Auxiliary). The symbols '*' means modified according to result, '–' means unchanged, '?' means undefined, '0' means cleared to zero. For complete details see the Intel® 64 and IA-32 Architectures Software Developer's Manual and the AMD64 Architecture Programmer's Manual.

Instruction	Description / Syntax	Flags affected
`MOV` dest, src	Copy src to dest. dest ← src.	None
`MOVZX` reg, r/m	Zero-extend byte/word to reg. Upper bits become 0.	None
`MOVSX` reg, r/m	Sign-extend byte/word/dword to reg.	None
`LEA` reg, mem	Load effective address of mem into reg (no memory read).	None
`ADD` dest, src	dest ← dest + src.	OF, SF, ZF, AF, PF, CF
`SUB` dest, src	dest ← dest – src.	OF, SF, ZF, AF, PF, CF
`MUL` src	Unsigned multiply. RDX:RAX ← RAX * src (64-bit).	CF, OF (others undefined)
`IMUL` src	Signed multiply. 1-op form: same as MUL signed. 2/3-op: truncating multiply.	CF, OF (others undefined)
`DIV` src	Unsigned divide. RDX:RAX / src → RAX=quot, RDX=rem.	All undefined
`IDIV` src	Signed divide. Same layout as DIV.	All undefined
`INC` dest	dest ← dest + 1. (CF unchanged)	OF, SF, ZF, AF, PF

Instruction	Description / Syntax	Flags affected
DEC dest	dest ← dest – 1. (CF unchanged)	OF, SF, ZF, AF, PF
NEG dest	dest ← 0 – dest (two's complement).	OF, SF, ZF, AF, PF, CF
AND dest, src	dest ← dest & src. OF=CF=0.	SF, ZF, PF; OF=0, CF=0, AF undef.
OR dest, src	dest ← dest | src. OF=CF=0.	SF, ZF, PF; OF=0, CF=0, AF undef.
XOR dest, src	dest ← dest ^ src. OF=CF=0.	SF, ZF, PF; OF=0, CF=0, AF undef.
NOT dest	Bitwise NOT (one's complement).	None
SHL dest, count	Shift left logical. Fill with zeros.	OF (cnt=1), CF last bit out, SF, ZF, PF
SHR dest, count	Shift right logical. Fill with zeros.	same as SHL
SAR dest, count	Shift right arithmetic. Fill with sign bit.	same as SHL
ROL dest, count	Rotate left without CF.	CF last bit rotated; OF (cnt=1)
ROR dest, count	Rotate right without CF.	CF, OF (cnt=1)
CMP dest, src	dest – src, sets flags only (discards result).	OF, SF, ZF, AF, PF, CF
TEST dest, src	dest & src, sets flags only.	SF, ZF, PF; OF=0, CF=0, AF undef.
JMP target	Unconditional jump.	None
Jcc target	Conditional jump: JE/JZ, JNE/JNZ, JL, JLE, JG, JGE, JB, JBE, JA, JAE, JS, JNS, JO, JNO, JC, JNC, JP, JNP.	None
CALL target	Push return address, jump to target.	None (pushes RIP)
RET	Pop return address, jump.	None
PUSH src	RSP ← RSP – 8; [RSP] ← src.	None
POP dest	dest ← [RSP]; RSP ← RSP + 8.	None
NOP	No operation.	None
REP MOVSB	Move RCX bytes from [RSI] to [RDI], increment both.	None
REP STOSB	Store AL into RCX bytes starting at [RDI].	None
REPE CMPSB	Compare bytes until mismatch or RCX=0.	Flags set from last comparison
REPNE SCASB	Scan for AL in string at RDI.	Flags set from last comparison
BSF dest, src	Bit scan forward: index of least significant set bit. ZF=1 if src=0.	ZF; others undefined
BSR dest, src	Bit scan reverse: index of most significant set bit. ZF=1 if src=0.	ZF; others undefined
BT dest, bit	Test bit, store original bit in CF.	CF

Instruction	Description / Syntax	Flags affected
`BTS` dest, bit	Test and set bit; CF ← original bit, then bit=1.	CF
`BTR` dest, bit	Test and reset bit; CF ← original bit, then bit=0.	CF
`BTC` dest, bit	Test and complement bit; CF ← original, then bit ← !bit.	CF
`MOVD` xmm, r/m32	Move 32-bit value between XMM reg and memory/GP reg.	None
`MOVQ` xmm, r/m64	Move 64-bit value between XMM reg and memory/GP reg.	None
`ADDPS` xmm1, xmm2/m128	Packed single-precision float addition (4 floats).	None (MXCSR exceptions)
`MULPS` xmm1, xmm2/m128	Packed single-precision float multiply.	None

Appendix B: Linux System Call and C Library Reference

This appendix lists essential Linux system calls and a subset of commonly used C library functions callable from NASM assembly on Fedora 43. The first six integer/pointer arguments go into RDI, RSI, RDX, RCX, R8, R9; additional arguments are pushed right-to-left onto the stack. The return value is in RAX (or XMM0 for floating point). System call numbers are for x86-64 and may be verified in `/usr/include/asm/unistd_64.h`.

System Calls

Function	Call #	Signature (Registers)
`read`	0	`int fd (RDI), void *buf (RSI), size_t count (RDX)`
`write`	1	`int fd (RDI), const void *buf (RSI), size_t count (RDX)`
`open`	2	`const char *pathname (RDI), int flags (ESI), mode_t mode (EDX)`
`close`	3	`int fd (RDI)`
`lseek`	8	`int fd (RDI), off_t offset (RSI), int whence (EDX)`
`mmap`	9	`void *addr (RDI), size_t len (RSI), int prot (EDX), int flags (R10d), int fd (R8d), off_t off (R9)`
`munmap`	11	`void *addr (RDI), size_t len (RSI)`
`brk`	12	`void *addr (RDI)`
`exit`	60	`int status (EDI)`
`getpid`	39	`void`

Function	Call #	Signature (Registers)
getuid	102	void
geteuid	107	void
fork	57	void
execve	59	const char *filename (RDI), char *const argv[] (RSI), char *const envp[] (RDX)
wait4	61	pid_t pid (RDI), int *wstatus (RSI), int options (EDX), struct rusage *rusage (R10)
clock_gettime	228	clockid_t clk_id (EDI), struct timespec *tp (RSI)

Common C Library Functions (via libc)

Function	Arguments	Return
printf	RDI = format, RSI,RDX,RCX,R8,R9 = args	int (chars written)
puts	RDI = string	non-negative on success
malloc	RDI = size	pointer or NULL
free	RDI = ptr	void
exit	EDI = status	never returns
strlen	RDI = string	size_t length
strcpy	RDI = dest, RSI = src	pointer to dest
strcmp	RDI = s1, RSI = s2	int result
getpagesize	none	page size
dlopen	RDI = filename, ESI = flags	handle or NULL
dlsym	RDI = handle, RSI = symbol	pointer or NULL
strerror	RDI = errnum	string

Common Flags and Constants

```
O_RDONLY    equ 0
O_WRONLY    equ 1
O_RDWR      equ 2
O_CREAT     equ 0100
O_TRUNC     equ 01000
O_APPEND    equ 02000
PROT_READ   equ 1
PROT_WRITE  equ 2
PROT_EXEC   equ 4
```

```
MAP_PRIVATE equ 2
MAP_ANONYMOUS equ 32
SEEK_SET    equ 0
SEEK_CUR    equ 1
SEEK_END    equ 2
STDIN       equ 0
STDOUT      equ 1
STDERR      equ 2
```

Appendix C: NASM Directives Reference

NASM provides a rich set of directives that control assembly, data definition, and symbol management. The following table lists the most commonly used directives for ELF64 Linux programming.

Directive	Syntax / Example	Description
`bits`	`bits 64`	Set default processor mode.
`default`	`default rel`	Make all memory references RIP-relative.
`section`	`section .text progbits alloc exec align=16`	Define a section with ELF attributes.
`global`	`global _start`	Export symbol for linker.
`extern`	`extern printf`	Import symbol from another module/library.
`equ`	`BUFSIZE equ 128`	Define numeric constant.
`db` / `dw` / `dd` / `dq`	`myvar dq 0, 1, 2`	Allocate initialized data (1,2,4,8 bytes).
`resb` / `resw` / `resd` / `resq`	`buffer resb 256`	Reserve uninitialized space.
`times`	`times 10 db 0`	Repeat a data directive.
`%include`	`%include "syscalls.inc"`	Insert another source file.
`%define`	`%define DEBUG 1`	Preprocessor macro (can be redefined).
`%macro` / `%endmacro`	`%macro prologue 1 ... %endmacro`	Define a multi-line macro.
`%rep` / `%endrep`	`%rep 5 dd i %assign i i+1 %endrep`	Repeat a block at assembly time.
`%if` / `%elif` / `%else` / `%endif`	`%if DEBUG int3 %endif`	Conditional assembly.
`%ifdef` / `%ifndef`	`%ifndef SYSCALLS_INC ... %endif`	Test if a symbol is defined.
`%assign`	`%assign i 0`	Assign a numeric value (reassignable).

Directive	Syntax / Example	Description
%warning	%warning "message"	Emit warning during assembly.
%error	%error "message"	Emit error and stop assembly.
align / alignb	align 16	Pad with NOPs (code) or zeros (data) to alignment.

Appendix D: Debugging Cheat Sheet (GDB)

A quick reference for debugging NASM programs with GDB on Linux.

GDB Commands

Command	Meaning
break _start	Set breakpoint at entry point.
break *0x4000b0	Set breakpoint at specific address.
break <func> if <cond>	Conditional breakpoint.
run	Start execution.
continue	Resume until next breakpoint.
stepi	Step one machine instruction (into calls).
nexti	Step one instruction (over calls).
info registers	Display all registers.
p/x $rax	Print RAX in hex.
x/10gx $rsp	Examine 10 quadwords at stack pointer.
x/s $rdi	Display null-terminated string at RDI.
x/i $rip	Disassemble instruction at RIP.
bt	Backtrace (call stack).
frame N	Switch to stack frame N.
watch *0x601000	Set hardware write watchpoint.
set $rax = 0x1234	Change register value.
layout asm	Enable TUI assembly view.
disassemble main	Show disassembly of function main.

Appendix E: Common Errors and Fixes on Linux

A collection of frequent pitfalls when developing NASM programs for Linux (Fedora 43).

Error / Symptom	Cause and Solution
undefined reference to `symbol'	Linker cannot find the definition. Verify that you have extern symbol and the object file or library defining it is linked.

`relocation R_X86_64_32 against `.data' can not be used when making a PIE object`	Using absolute addressing in position-independent code. Use `default rel` and RIP-relative addressing.
Segmentation fault after `syscall`	Possibly wrong system call number or arguments. Check `$rax` for negative errno value. Use `strace ./prog` to trace syscalls.
Program exits with insane exit code	The `syscall` number may be wrong, or you passed a garbage exit code. Verify `rax` before `syscall`.
Unexpected output truncation	`write` may return fewer bytes than requested. Implement a loop that handles partial writes.
Crash when calling libc functions (e.g., `printf`)	Stack not 16-byte aligned before `CALL`. Realign with `sub rsp, 8` (odd multiple of 8) before the call.
`bad file descriptor` or file not found	Check file path, flags, and mode. Use `ls -l` to verify permissions.
Dynamic executable "No such file or directory"	Missing shared library. Use `ldd ./prog` to list dependencies. Set `LD_LIBRARY_PATH` or install the library.
Infinite loop after reading from stdin	The newline was not stripped. `read` includes the newline; remove it if needed.
`_start` not found by linker	Missing `global _start` or typo in label name. Ensure entry point is `global`.

Appendix F: Project Templates (Linux)

A minimal project template for a purely system-call-based NASM application on Linux.

Directory Structure

```
project/
  src/
    main.asm
  obj/
  bin/
  Makefile
```

Minimal Source Template (src/main.asm)

```
; main.asm -- Minimal Linux x86-64 program
; Assemble: nasm -f elf64 -o obj/main.o src/main.asm
; Link:     ld -o bin/app obj/main.o

bits 64
default rel

SYS_EXIT equ 60

section .text
```

```
global _start
_start:
    mov     eax, SYS_EXIT
    xor     edi, edi
    syscall
```

GNU Makefile

```
NASM   = nasm
LD     = ld
OBJ    = obj/main.o
TARGET = bin/app

$(TARGET): $(OBJ) | bin
^^I$(LD) -o $@ $^

obj/%.o: src/%.asm | obj
^^I$(NASM) -f elf64 -o $@ $<

obj:
^^Imkdir -p obj

bin:
^^Imkdir -p bin

.PHONY: clean
clean:
^^Irm -rf obj bin
```

REFERENCES

Intel 64 and IA-32 Architectures Software Developer's Manual

The Intel 64 architecture defines the fundamental execution model used by modern x86-64 processors. It specifies the instruction set, memory model, privilege levels, paging, and system programming interfaces.

Key architectural principles:

- Flat 64-bit linear address space in long mode.
- Segmentation is largely disabled except for FS and GS base usage.
- Paging is required (4-level or 5-level paging depending on CPU generation).
- Instructions follow variable-length encoding (1 to 15 bytes).
- Registers include 16 general-purpose 64-bit registers (RAX–R15).

Memory operations follow strict ordering rules under the x86 memory consistency model, which guarantees strong ordering for aligned accesses except when explicitly relaxed by special instructions.

```
mov rax, [rbx]
add rax, 1
mov [rbx], rax
```

This sequence demonstrates a basic read-modify-write operation in a single-threaded context.

AMD64 Architecture Programmer's Manual

The AMD64 architecture extends the original x86 design to 64-bit operation while preserving backward compatibility.

Key features:

- 64-bit general purpose registers.
- RIP-relative addressing mode.

- Extended register set (R8–R15).
- System call interface via `SYSCALL` / `SYSRET` instructions.

On Linux the System V AMD64 Application Binary Interface supplement governs the function calling convention, stack layout, and process initialisation.

```
lea rax, [rip + label]
```

RIP-relative addressing is mandatory for position-independent code in modern systems.

NASM Official Documentation

NASM is a macro assembler that supports Intel syntax and multiple output formats including ELF64 for Linux.

Key NASM concepts:

- Strong separation of sections: `.text`, `.data`, `.bss`, `.rodata`.
- Symbolic expression evaluation at assembly time.
- Macro system for code abstraction.
- Direct support for ELF object generation via `-f elf64`.

```
section .data
msg db "Hello NASM", 0

section .text
global _start
```

NASM does not impose calling conventions; it relies on platform ABI.

System V AMD64 ABI (Linux x86-64 Calling Convention)

The System V AMD64 ABI defines parameter passing rules for Linux:

- First six integer/pointer arguments: RDI, RSI, RDX, RCX, R8, R9.
- Floating-point arguments use XMM0–XMM7.
- The stack must be 16-byte aligned at the point of a `CALL` instruction; no mandatory shadow space.
- Callee-saved registers: RBX, RBP, R12–R15; all others are call-clobbered.
- System calls use a different register allocation: call number in RAX, arguments in RDI, RSI, RDX, R10, R8, R9.

```
sub rsp, 8          ; align stack before call
mov rdi, 1          ; fd = stdout
lea rsi, [msg]
mov rdx, len
mov eax, 1          ; sys_write
syscall
add rsp, 8
```

Failure to maintain stack alignment may cause segmentation faults inside C library functions.

Linux System Call and C Library Reference

The Linux kernel provides system calls directly invocable with the `syscall` instruction. The C library (`glibc`) offers higher-level wrappers.

Common system calls:

- `write`, `read`, `open`, `close`, `lseek`
- `mmap`, `munmap`, `brk`
- `exit`, `fork`, `execve`, `wait4`

The `man` pages (`man 2` for system calls, `man 3` for library functions) provide the definitive reference.

```
mov eax, 60         ; SYS_exit
xor edi, edi
syscall
```

System call numbers are stable and listed in `/usr/include/asm/unistd_64.h`.

ELF Specification (Executable and Linkable Format)

ELF defines the structure of object files, shared libraries, and executables on Linux.

Key components:

- ELF header with machine type (EM_X86_64) and entry point.
- Program header table defining load segments (`PT_LOAD`).
- Section headers for `.text`, `.data`, `.rodata`, `.bss`, `.symtab`, etc.
- Relocation tables (`.rela.text`, `.rela.dyn`) and dynamic symbol information.

Sections control memory mapping:

- Executable code: `.text` (PROT_READ | PROT_EXEC)
- Read-only data: `.rodata` (PROT_READ)
- Writable data: `.data` / `.bss` (PROT_READ | PROT_WRITE)

```
section .text   progbits alloc exec align=16
section .data   progbits alloc write align=8
section .rodata progbits alloc noexec read align=8
```

Alignment and flags are critical for the linker and dynamic linker.

Linux Kernel Documentation

The Linux kernel source tree and the official kernel documentation describe the memory manager, scheduler, process model, and system call interface.

Key concepts:

- Process virtual address space layout (user stack, heap, mmap region, code).
- `/proc/<pid>/maps` for inspecting memory regions.
- Demand paging and page fault handling.
- Completely Fair Scheduler (CFS) and process priorities.
- Mandatory access control via SELinux/AppArmor.

From an assembly programmer's perspective, the most important interfaces are the system call table and the ELF loader.

Computer Systems: A Programmer's Perspective (CS:APP)

Core principles laid out in *Computer Systems: A Programmer's Perspective* (Bryant & O'Hallaron) apply directly to assembly on Linux:

- Data representation in binary and two's complement.
- Memory hierarchy (registers, cache, RAM, disk).
- Linking and loading of ELF objects.
- System-level I/O via file descriptors.

```
mov rax, [rbx]
imul rax, rcx
```

This reflects instruction-level translation of high-level arithmetic.

Modern x86 Assembly Language Programming (Kusswurm)

Although *Modern x86 Assembly Language Programming* by Daniel Kusswurm focuses primarily on the Windows platform, its coverage of SIMD (SSE, AVX, AVX-512) and performance-optimisation techniques is directly applicable to Linux. The calling convention differences are minor; the instruction set and micro-architectural principles remain identical.

```
vmovdqu ymm0, [rcx]
vaddps ymm0, ymm0, ymm1
```

SIMD operations enable parallel data processing on any x86-64 operating system.

GNU Binutils Documentation

GNU Binutils includes the GNU assembler (GAS), the GNU linker (ld), and object-file tools such as `objdump`, `readelf`, and `nm`.

Key concepts:

- ELF relocations (R_X86_64_PC32, R_X86_64_PLT32, etc.).
- Linker scripts and symbol resolution.
- Support for position-independent executables (PIE) and shared libraries.

NASM directly interacts with the GNU linker; understanding Binutils is essential for diagnosing link-time errors.

LLVM Documentation

LLVM provides a modern compiler infrastructure that can be used alongside NASM. The LLVM linker `ld.lld` is a drop-in replacement for GNU ld, often faster and compatible with ELF objects produced by NASM.

- Intermediate Representation (IR) – not directly used by NASM, but useful for understanding code generation.
- Optimisation passes that can complement hand-written assembly.
- The LLVM linker (`lld`) supports the standard flag set of GNU ld.

GDB Documentation (GNU Debugger)

GDB is the standard debugger for Linux. It supports source-level debugging with DWARF information, breakpoints, hardware watchpoints, and register inspection.

Common commands:

- `break _start` – set breakpoint at entry point.
- `run` – start execution.
- `stepi / nexti` – instruction-level stepping.
- `info registers` – display all GPRs.
- `x/10gx $rsp` – examine stack.
- `watch *0x601000` – set hardware write watchpoint.

GDB's TUI (`layout asm`) provides a disassembly view alongside register display.

```
gdb ./prog
(gdb) break _start
(gdb) run
```

strace and ltrace Documentation

`strace` is a system call tracer that intercepts and records every system call made by a process, showing arguments and return values. `ltrace` traces dynamic library calls.

Usage:

```
strace ./prog
strace -o trace.log ./prog
```

This is invaluable for debugging assembly programs, as it reveals exactly which syscalls are issued and whether errors occur, even if the program itself produces no output.

perf Documentation

`perf` is the Linux kernel profiling tool. It can measure processor cycles, cache misses, branch mispredictions, and many other hardware events.

- `perf record ./prog` – collect profile data.
- `perf report` – display annotated disassembly or call graph.
- `perf stat ./prog` – print aggregate hardware counter values.

Using `perf` together with NASM (assembled with debug info `-g`) allows precise performance analysis of hand-tuned routines.

Valgrind Documentation

Valgrind (particularly the `memcheck` and `cachegrind` tools) simulates a program's interaction with memory and the cache hierarchy. It detects memory leaks, uninitialised reads, and buffer overflows.

- `valgrind --leak-check=full ./prog` – find memory leaks.
- `valgrind --tool=cachegrind ./prog` – simulate L1/L2 cache usage.

For pure assembly programs written with manual memory management, Valgrind is an essential testing tool.No LaTeX output – just a clean, Linux-adapted REFERENCES chapter. The Windows-specific sources (Microsoft ABI, PE/COFF, Win32 API, WinDbg, x64dbg, Windows Internals) have been replaced with the System V AMD64 ABI, ELF specification, Linux system call and man-page references, GDB, strace, perf, Valgrind, and the Linux kernel documentation. The Intel, AMD, NASM, CS:APP, and LLVM references remain unchanged because they are platform-independent.