Low-Level Programming on Linux (x86-64)

Linux System Calls and Process Control

redhat

debian

3

Prepared by Ayman Alheraki

https://simplifycpp.org

Low-Level Programming on Linux (x86-64) :

Linux System Calls and Process Control

Prepared by Ayman Alheraki

simplifycpp.org

November 2025

Contents

Author's Introduction

The purpose of this booklet is to present a clear and foundational understanding of **Linux system calls and process control** from a low-level perspective, without relying on external libraries or high-level abstractions. In modern software development, many developers interact with operating systems indirectly through frameworks, runtime environments, or managed languages. While these tools are valuable, they often conceal the underlying mechanisms that make program execution, file management, memory protection, and process scheduling possible. To write robust, efficient, and secure software, it is essential to study these mechanisms directly.

This work focuses on **what the kernel actually provides** and **how user space programs request services** through the system call interface. By examining the register conventions, return value semantics, process creation, program replacement, environment handling, and signaling behavior at the assembly level, we establish a precise model of how Linux manages execution. This model is not theoretical; it is the operational reality beneath every program that runs on a Linux system, from command-line utilities to graphical applications, servers, and containers.

Throughout the chapters, the explanations emphasize clarity, minimalism, and correctness. Every code sample is explicit, does not depend on libc or compiler-inserted scaffolding, and is written in **x86-64 Intel assembly** to ensure complete transparency of execution. This approach allows us to see exactly how arguments are passed, how memory is laid out, how control transfers between processes, and how the kernel interprets and responds to requests.

The culmination of this booklet is the construction of a **fully functional minimal shell** that supports interactive input, tokenization, environment inheritance, PATH-based program lookup, forking, execution, and process synchronization. This implementation is not a demonstration artifact—it is a direct expression of the UNIX process model, reduced to its essential and correct form.
This booklet is written for readers who wish to:

- Deepen their understanding of how Linux executes programs.
- Strengthen system-level reasoning for debugging and performance tuning.
- Learn to work without reliance on high-level libraries.
- Develop tools that require precise control over execution and process behavior.
- Prepare for more advanced system topics such as dynamic linking, IPC, namespaces, kernel interfaces, or runtime design.

It is my hope that this work serves both as a technical guide and as an invitation to explore system programming at a level where design decisions are guided by clarity rather than convention. Mastery of these fundamentals allows one to use high-level abstractions with intention instead of dependency.
This booklet is one part of a broader series dedicated to **low-level programming on Linux**, each unit focusing on a different foundational layer. Together, they form a structured path from the circuitry of the processor, through assembly and system call interfaces, up to executable formats and runtime environments.

Stay Connected
For more discussions and valuable content about **Linux System Calls and Process Control)**, I invite you to follow me on **LinkedIn**:
https://linkedin.com/in/aymanalheraki

You can also visit my personal website:

https://simplifycpp.org

Ayman Alheraki

Chapter 1

User Space vs Kernel Space

1.1 Privilege Levels and the Ring Model

Modern x86-64 processors implement a hierarchical privilege model known as the **ring model**, designed to isolate software components based on their ability to perform sensitive operations. Linux aligns this hardware-defined mechanism into two primary execution domains: **user space** and **kernel space**. While the hardware supports four privilege levels (Ring 0 to Ring 3), Linux typically uses **Ring 0** for the kernel and **Ring 3** for user-level processes. Rings 1 and 2 are generally unused by Linux itself; they are not required for a monolithic kernel design. (Separately, virtualization technologies may introduce additional privilege modes such as VMX root mode on Intel VT-x, which is distinct from Rings 0–3.)

1.1.1 Core Rationale Behind Privilege Separation

The CPU must protect the integrity of the system from malfunctioning or malicious user programs. Instructions that manipulate hardware state, memory maps, interrupts, I/O ports, or control registers must only be executed by trusted code. User applications execute in a

constrained environment, issuing **system calls** to request the kernel to perform privileged operations on their behalf.
This separation enables:

- Isolation between user programs and the kernel (fault containment).
- Prevention of direct hardware access (I/O ports, control registers, privileged MSRs).
- Memory protection enforced primarily by paging (with segmentation mostly used for legacy and TLS/FS/GS bases on x86-64).
- Controlled communication via well-defined entry mechanisms (exceptions, interrupts, `syscall`).
- A consistent protection model across multicore systems.

1.1.2 Hardware Enforcement

Privilege enforcement occurs through multiple architectural mechanisms:

- **CPL (Current Privilege Level)** derived from the low bits of the CS selector.
- **Control registers** (CR0, CR4) enabling protection features, and CR3 selecting the active page-table root.
- **Page tables** marking pages as user-accessible or supervisor-only using the U/S bit in paging structures.
- **Entry/exit rules** for privilege transitions (e.g., IDT gates, `syscall`/`sysretq`, `iretq`).

When executing in **Ring 3**, attempts to execute privileged instructions trigger faults. For example, accessing a control register such as CR3 from user mode raises a `#GP` (General Protection) fault.

Example: privileged instruction (illegal in user mode).

```
.intel_syntax noprefix
# Attempt to read CR3 in user space (will trigger #GP)
mov rax, cr3
```

1.1.3 Transitioning Across Rings

The transition from user space to kernel space is achieved via controlled mechanisms. Historically, 32-bit Linux used `int 0x80`. On x86-64 Linux, the standard fast path is **`syscall`** (entry) paired with **`sysretq`** (return) for most cases.

Example: userspace system call entry (Ring 3 → Ring 0).

```
.intel_syntax noprefix
# write(1, message, message_len)
mov eax, 1              # SYS_write
mov edi, 1              # fd = STDOUT
lea rsi, [rip + message]
mov edx, message_len
syscall                 # enter kernel
```

On x86-64 Linux, the system call convention is:

- System call number in **RAX**.
- Arguments in **RDI, RSI, RDX, R10, R8, R9**.
- Return value in **RAX** (negative values typically represent `-errno`).

1.1.4 What `syscall` Actually Does (Architecturally)

Executing `syscall` causes the CPU to:

- Switch privilege level to Ring 0 using kernel code segment selectors configured by the OS (via MSRs).
- Jump to the kernel entry address stored in `IA32_LSTAR`.
- Save the user return RIP into **RCX** and save RFLAGS into **R11**.
- Mask selected flags as specified by `IA32_FMASK`.

Important correction: `syscall` does *not* inherently perform a full hardware stack switch on its own. Linux is responsible for establishing a safe kernel stack early in the entry path. On x86-64, Linux commonly uses mechanisms such as the TSS `RSP0` and/or per-CPU data (often via `swapgs`) as part of the entry stub to ensure kernel execution uses a kernel-controlled stack.

1.1.5 Returning to User Mode

For the fast path, the kernel returns to user mode using **`sysretq`** (with some corner cases using `iretq`). Architecturally, `sysretq` returns to the address in **RCX** and restores flags from **R11**, while restoring user-mode segment selectors as configured by the OS.

Example (conceptual): kernel → userspace return.

```
.intel_syntax noprefix
# RAX holds the system call return value.
# RCX holds the user RIP, R11 holds user RFLAGS (set up by entry/exit
↪  stubs).
sysretq
```

1.1.6 Example: Determining CPL in Assembly

CPL can be observed by reading the low two bits of the CS selector. In user mode on Linux, this yields 3; in kernel mode, it yields 0.

Example: read CS and isolate CPL bits.

```
.intel_syntax noprefix
xor eax, eax
mov ax, cs              # move CS selector into AX (16-bit)
and eax, 3              # isolate CPL bits
# eax = 3 in user mode, eax = 0 in kernel mode
```

Correction: The prior "`rax |=| 3`" blocks are not valid assembly and should be removed. The check above is the correct and complete demonstration.

1.1.7 Summary

Linux adopts a strict **two-domain execution model** built on the x86-64 privilege architecture: the **kernel** executes with full authority in **Ring 0**, while **applications** execute in **Ring 3**, using system calls as the only legitimate mechanism to request privileged services. Hardware enforcement (paging permissions, privileged instruction checks, and controlled entry/exit rules) provides strong isolation and prevents user code from directly controlling core system state.

1.2 Memory Separation and Safety

Memory separation between user space and kernel space is a fundamental property of modern Linux systems on x86-64. The goal is to ensure that applications cannot directly read, write, or execute memory that belongs to the kernel or other processes. This isolation prevents accidental corruption and forms a primary boundary against malicious code. It is enforced chiefly by the **virtual memory system**: each process has its own address space, while the kernel remains mapped in a way that is accessible only when running with supervisor privileges.

1.2.1 Virtual Address Space Layout

On x86-64, processes use **canonical** virtual addresses (sign-extended), meaning not all 64-bit patterns are valid addresses. Linux splits the virtual address space into user and kernel regions. The exact split depends on kernel configuration, but the conceptual separation is:

- **Lower canonical addresses**: user space mappings.
- **Upper canonical addresses**: kernel space mappings.

Even when kernel pages are mapped into a process, they are marked as **supervisor-only** in the page tables. Any attempt to access them from Ring 3 triggers a page fault.

1.2.2 Supervisor and User Permissions in Page Tables

Paging protection uses the U/S (User/Supervisor) permission bit present in page-table entries:

- **U/S = 1**: accessible from user mode (Ring 3).
- **U/S = 0**: supervisor-only (kernel mode).

Attempting to access a supervisor page from Ring 3 triggers a page fault (`#PF`) where the error code indicates a protection violation.

Example: user code reading a supervisor-only address (will fault).

```
.intel_syntax noprefix
# Assume RSI contains an address mapped supervisor-only
mov rax, qword ptr [rsi]      # triggers #PF in user mode
```

1.2.3 System Call Boundary and Controlled Access

User processes cannot modify page tables directly. Operations such as mapping memory, allocating pages, or changing permissions are performed by the kernel via system calls such as `mmap`, `mprotect`, or `brk`.

Example: request anonymous memory via `mmap` (userspace).

```
.intel_syntax noprefix
# void* mmap(NULL, 4096,
↪  PROT_READPROT_WRITE, MAP_PRIVATEMAP_ANONYMOUS, -1, 0)
mov eax, 9                # SYS_mmap
xor edi, edi              # addr = NULL
mov esi, 4096             # length
mov edx, 3                # prot = 12 (READWRITE)
mov r10d, 0x22            # flags = 0x020x20 (PRIVATEANONYMOUS)
mov r8, -1                # fd = -1
xor r9d, r9d              # offset = 0
syscall                   # RAX = result pointer or -errno
```

1.2.4 Kernel Memory Isolation in Modern Designs

Modern Linux kernels deploy multiple hardening mechanisms to reduce the risk of information disclosure and privilege escalation. One significant measure is **Kernel Page-Table Isolation (KPTI)**, introduced as a mitigation for certain speculative execution attacks. With KPTI enabled, user mode runs with page tables that map far less kernel memory, and the kernel switches page-table context when entering privileged execution.

1.2.5 Validating User Pointers in the Kernel

When servicing a system call, the kernel must treat user pointers as untrusted. Kernel code commonly uses guarded copy routines (conceptually `copy_from_user`/`copy_to_user`) that handle faults and return errors rather than crashing the kernel.

Conceptual illustration (not literal Linux source).

```
.intel_syntax noprefix
# rdi = user pointer, rsi = kernel buffer, rcx = length
# A real kernel uses hardened helpers and exception fixups.
copy_loop:
    test rcx, rcx
    je   done
    mov  al, byte ptr [rdi]     # may fault if user pointer is invalid
    mov  byte ptr [rsi], al
    inc  rdi
    inc  rsi
    dec  rcx
    jmp  copy_loop
done:
    ret
```

If the user pointer is invalid or points to supervisor-only memory, the load triggers `#PF`. The kernel handles the fault and returns an error to the caller rather than allowing corruption.

1.2.6 Summary

Memory separation in Linux is rooted in x86-64 paging permissions and privilege enforcement. User processes execute in Ring 3 and cannot access supervisor-only pages, even if those pages are mapped. Changes to mappings and permissions are mediated by the kernel

through validated system calls. This model provides stability, safety, and a strong security boundary across the system.

1.3 How System Calls Bridge the Two

The system call interface is the controlled gateway that allows user space to request operations from kernel space. Since user programs execute in Ring 3 and the kernel executes in Ring 0, direct calls into kernel code are forbidden. A system call performs a hardware-mediated transition to a predefined kernel entry point, with the kernel handling argument validation, permission checks, and the privileged operation.

1.3.1 Architectural Mechanism

On x86-64 Linux, `syscall` is paired with `sysretq` for the fast path. The transition behavior is defined by MSRs set up by the kernel during initialization:

- `IA32_LSTAR`: 64-bit system call entry RIP (kernel entry point).
- `IA32_STAR`: segment selector configuration for `syscall`/`sysretq`.
- `IA32_FMASK`: flags masked on system call entry.

User mode cannot write these registers.

1.3.2 User-Space Invocation

Linux x86-64 system call ABI:

- **RAX**: system call number.
- **RDI, RSI, RDX, R10, R8, R9**: arguments 1–6.
- **RAX**: return value (or `-errno`).

Example: minimal `write` and `exit` in GAS (Intel syntax).

```
.intel_syntax noprefix
.section .rodata
message:
    .ascii "Hello from user space.\n"
message_len = . - message

.section .text
.globl _start
_start:
    mov eax, 1                  # SYS_write
    mov edi, 1                  # fd = STDOUT
    lea rsi, [rip + message]
    mov edx, message_len
    syscall

    mov eax, 60                 # SYS_exit
    xor edi, edi
    syscall
```

1.3.3 Kernel Entry Path (Conceptual)

Upon entry, the CPU begins execution at the kernel's system call entry stub. The stub saves state, establishes a safe kernel execution context, and dispatches to the system call handler. A simplified conceptual view:

```
.intel_syntax noprefix
syscall_entry:                  # conceptual only
    swapgs                      # switch GS base (per-CPU / per-thread
    ↪  data)
    # establish kernel stack / save user context / speculation barriers...
    call syscall_dispatcher
```

```
    swapgs
    sysretq
```

The real kernel path includes additional details (interrupt state, scheduler integration, speculation mitigations), but the conceptual flow is correct.

1.3.4 Returning to User Mode

The kernel prepares **RAX** as the return value, restores user state as required, and returns with `sysretq` (or `iretq` for paths requiring full state restoration). Invalid return state is handled as a fault to prevent privilege escalation.

1.3.5 System Calls vs Function Calls

Conceptually:

- A **function call** transfers control within the same privilege domain.
- A **system call** transfers control across privilege domains via a hardware-defined gate.

Unlike a function call, a system call implies a privileged entry path, kernel-side validation, and potentially different stacks, interrupt handling, and scheduling semantics.

1.3.6 Summary

System calls are the only legitimate bridge between unprivileged user programs and the privileged kernel. The CPU enforces the transition rules using dedicated instructions and MSR-defined entry points, while Linux enforces security by validating arguments, controlling memory permissions, and executing privileged operations only within kernel mode.

Chapter 2

The System Call Interface

2.1 Register-Based Calling Convention

System calls in Linux on x86-64 use a strictly **register-based calling convention**, distinct from both the System V ABI used for ordinary function calls and the Windows x64 calling convention. This convention defines how user-space code passes the system call number and arguments to the kernel before executing the `syscall` instruction.
Using registers instead of a stack-based frame minimizes overhead, avoids memory dependencies during privilege transitions, and simplifies hardware enforcement at the user–kernel boundary.

2.1.1 System Call Number and Argument Registers

Linux x86-64 defines a fixed register mapping for system calls:

- **RAX** — system call number
- **RDI** — argument 1

- **RSI** — argument 2
- **RDX** — argument 3
- **R10** — argument 4
- **R8** — argument 5
- **R9** — argument 6
- **RAX** — overwritten with return value (or `-errno`)

Arguments beyond six must be passed indirectly, typically by passing a pointer to a memory structure.
This differs from the standard System V function ABI, which uses **RCX** as the fourth argument. System calls deliberately use **R10** instead, because the `syscall` instruction overwrites **RCX** with the user return address.

2.1.2 Minimal User-Space System Call Example

The following example issues `sys_write` and `sys_exit` directly, without linking against the C library:

```
.intel_syntax noprefix

.section .rodata
msg:
    .ascii "Direct syscall in x86-64\n"
msg_len = . - msg

.section .text
.globl _start
_start:
```

```
    mov eax, 1              # SYS_write
    mov edi, 1              # stdout
    lea rsi, [rip + msg]
    mov edx, msg_len
    syscall

    mov eax, 60             # SYS_exit
    xor edi, edi
    syscall
```

No stack manipulation is required. The kernel reads arguments directly from registers at the moment the privilege transition occurs.

2.1.3 Register Semantics Across the `syscall` Boundary

During the `syscall` transition (Ring 3 $\rightarrow$ Ring 0):

- **RCX** is overwritten with the user return RIP
- **R11** is overwritten with user RFLAGS
- Other general-purpose registers are preserved
- Execution jumps to the kernel entry point in `IA32_LSTAR`
- Selected flags are masked according to `IA32_FMASK`

Important correction: the CPU does *not* automatically push arguments onto the kernel stack. Linux establishes a valid kernel stack early in the entry path (via per-CPU data and TSS configuration).
Because no stack frame is transferred across the boundary, register passing is the only supported mechanism for syscall parameters.

2.1.4 Example With Six Arguments (`sys_sendto`)

```
.intel_syntax noprefix
# ssize_t sendto(int sockfd, void *buf, size_t len,
#                int flags, struct sockaddr *dest, socklen_t addrlen)

mov eax, 44              # SYS_sendto
mov edi, sockfd
mov rsi, buffer
mov rdx, length
mov r10d, flags          # must use r10, not rcx
mov r8,  dest_addr
mov r9,  addrlen
syscall
```

If the fourth argument is mistakenly placed in **RCX**, the kernel will receive corrupted arguments, leading to undefined behavior or immediate failure.

2.1.5 Return Values and Error Codes

All system calls return their result in **RAX**:

- Non-negative value — success
- Negative value — `-errno`

The kernel does not set a global error variable.

Checking for failure in assembly:

```
cmp rax, 0
jge .success
neg rax          # convert -errno to errno
```

2.1.6 Summary

Linux system calls use a minimal and hardware-aligned ABI: arguments are passed exclusively through registers, the system call number is placed in RAX, and the result is returned in RAX. No stack frame crosses the privilege boundary. This simplicity ensures correctness, performance, and strict separation between user and kernel execution domains.

2.2 Return Values and Error Codes

Every system call returns its result through **RAX**. The kernel uses a unified mechanism to report both successful outcomes and failure conditions, avoiding implicit memory state or global variables during the privilege transition.

2.2.1 Successful Return Values

For successful calls:

- Calls that produce a result place it directly in RAX
- Calls with no logical return value set RAX to zero

Example:

```
mov eax, 1
mov edi, 1
lea rsi, [rip + msg]
mov edx, msg_len
syscall
# rax = number of bytes written
```

2.2.2 Error Signaling

On failure, the kernel returns **negative errno values**:

- `-EPERM` — permission denied
- `-EFAULT` — invalid pointer
- `-EBADF` — invalid file descriptor

Detection is performed via signed comparison:

```
cmp rax, 0
jge .ok
```

2.2.3 Example: Handling Errors in Pure Assembly

```
.intel_syntax noprefix

.section .rodata
msg:
    .ascii "Test\n"
msg_len = . - msg

.section .text
.globl _start
_start:
    mov eax, 1              # SYS_write
    mov edi, -1             # invalid fd
    lea rsi, [rip + msg]
    mov edx, msg_len
    syscall

    cmp rax, 0
    jge .ok
    neg rax                 # rax = errno

.ok:
    mov eax, 60
    xor edi, edi
    syscall
```

2.2.4 No Automatic `errno`

The kernel does not maintain `errno`. That abstraction exists only in the C library, which translates negative RAX values into thread-local error storage.

2.2.5 Summary

System call return semantics are deliberately simple: RAX communicates both success and failure. This design avoids hidden state, ensures robustness across privilege transitions, and provides a clean foundation for low-level programming.

2.3 Inspecting System Calls with `strace`

System calls form the execution boundary between user space and kernel space, but they are often hidden beneath library abstractions. `strace` exposes this boundary by tracing system call entry and exit using the `ptrace` interface.

2.3.1 Observing System Calls

```
.intel_syntax noprefix

.section .rodata
msg:
    .ascii "Tracing syscalls\n"
msg_len = . - msg

.section .text
.globl _start
_start:
    mov eax, 1
    mov edi, 1
    lea rsi, [rip + msg]
    mov edx, msg_len
    syscall

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

Run with:

```
strace ./a.out
```

Typical output:

```
write(1, "Tracing syscalls\n", 17) = 17
exit(0) = ?
```

2.3.2 What `strace` Shows

`strace` decodes:

- RAX — system call number and return value
- RDI, RSI, RDX, R10, R8, R9 — arguments
- Entry and exit points only (not kernel internals)

2.3.3 Attaching to a Running Process

`strace -p <pid>`

This allows inspection without restarting the program and without granting arbitrary memory access.

2.3.4 Summary

`strace` reveals the exact register-level contract between user space and the kernel. It is an essential tool for validating manual system call construction, diagnosing low-level failures, and understanding how high-level APIs translate into kernel interactions.

Chapter 3

Input / Output via Syscalls

3.1 `read` and `write` Basics

All input and output operations in Linux ultimately rely on the system calls `read` and `write`. These calls operate on **file descriptors**, which abstract input and output targets such as terminals, files, pipes, sockets, and devices. Every process begins execution with three predefined file descriptors:

- **0** — standard input (`stdin`)
- **1** — standard output (`stdout`)
- **2** — standard error (`stderr`)

Both `read` and `write` operate on raw byte streams. They perform no buffering, formatting, encoding, or line handling. All higher-level I/O abstractions—such as buffered I/O, formatted output, or character encodings—are implemented entirely in user space on top of these primitives.

3.1.1 System Call Interface

On Linux x86-64, the system call numbers are:

- `read` → **RAX = 0**
- `write` → **RAX = 1**

Both system calls accept three arguments:

Register	Meaning
`RDI`	File descriptor
`RSI`	Pointer to user buffer
`RDX`	Number of bytes requested

The return value is placed in **RAX** and represents the number of bytes actually transferred or a negative error code (`-errno`).

3.1.2 Example: Writing to Standard Output

```
.intel_syntax noprefix

.section .rodata
msg:
    .ascii "Hello via write()\n"
msg_len = . - msg

.section .text
.globl _start
_start:
    mov eax, 1              # SYS_write
    mov edi, 1              # stdout
```

```
    lea rsi, [rip + msg]
    mov edx, msg_len
    syscall

    mov eax, 60             # SYS_exit
    xor edi, edi
    syscall
```

The kernel writes exactly the bytes provided. No newline insertion, buffering, or formatting is performed.

3.1.3 Example: Reading from Standard Input

`read` attempts to copy up to the requested number of bytes into the provided buffer. The actual return value may be smaller.

```
.intel_syntax noprefix

.section .bss
buffer:
    .skip 64

.section .text
.globl _start
_start:
    mov eax, 0              # SYS_read
    mov edi, 0              # stdin
    lea rsi, [rip + buffer]
    mov edx, 64
    syscall                 # rax = bytes read

    mov edx, eax            # use actual byte count
    mov eax, 1              # SYS_write
```

```
    mov edi, 1              # stdout
    lea rsi, [rip + buffer]
    syscall

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

This program echoes raw input without interpretation or modification.

3.1.4 Characteristics of `read` and `write`

- No internal buffering is performed.
- Data is treated as opaque bytes.
- Fewer bytes than requested may be returned.
- Partial transfers are normal and must be handled explicitly.

3.1.5 Summary

`read` and `write` are the foundational I/O primitives in Linux. They operate on file descriptors, transfer raw bytes, and return precise counts of completed work. All higher-level I/O mechanisms are built upon these syscalls.

3.2 Working with File Descriptors

A file descriptor is a small integer that indexes a kernel-managed open file description. It does not represent a file directly, but rather a reference to kernel state describing an open resource, including access mode, offset, and underlying object type.
Every process begins with descriptors 0, 1, and 2 already open. Additional descriptors are created using system calls such as `openat`, `socket`, or `pipe`, or inherited across `fork`.

3.2.1 Opening a File

On modern Linux systems, `openat` is the preferred interface.

System call number:

`SYS_openat = 257`

Arguments:

- `RDI` — directory file descriptor (`AT_FDCWD = -100`)
- `RSI` — pointer to pathname
- `RDX` — flags
- `R10` — file mode (only when creating)

```
.intel_syntax noprefix

.section .rodata
path:
    .ascii "test.txt\0"
```

```
.section .text
.globl _start
_start:
    mov eax, 257            # SYS_openat
    mov edi, -100           # AT_FDCWD
    lea rsi, [rip + path]
    mov edx, 1              # O_WRONLY
    mov r10d, 0644          # permissions
    syscall                 # rax = fd or -errno
```

3.2.2 Using a File Descriptor

Once opened, the descriptor is passed to `read` or `write` like any other I/O endpoint.

```
.intel_syntax noprefix

# assume rdi holds an open file descriptor
lea rsi, [rip + msg]
mov edx, msg_len
mov eax, 1                  # SYS_write
syscall
```

3.2.3 Closing a File Descriptor

Descriptors must be explicitly released using `close`:

```
.intel_syntax noprefix
mov eax, 3                  # SYS_close
# rdi = file descriptor
syscall
```

Using a closed descriptor results in `-EBADF`.

3.2.4 Duplication and Redirection

Descriptors may be duplicated using `dup2` or `dup3`.

```
.intel_syntax noprefix
mov eax, 33                 # SYS_dup2
mov edi, 1                  # old fd (stdout)
mov esi, 2                  # new fd (stderr)
syscall
```

This technique underlies shell redirection and pipelines.

3.2.5 Lifetime and Inheritance

File descriptors are inherited across `fork`. After `exec`, descriptors remain open unless marked with `FD_CLOEXEC`. This allows precise control over I/O environments.

3.2.6 Summary

File descriptors provide a uniform abstraction for all I/O resources. They are:

- Created via explicit syscalls
- Used by `read` and `write`
- Duplicated for redirection
- Closed to release kernel resources

3.3 Handling Partial Reads and Writes

System calls do not guarantee that all requested bytes are processed in a single invocation. Partial transfers are normal and must be handled explicitly.

3.3.1 Partial Reads

```
.intel_syntax noprefix

.section .bss
buffer:
    .skip 128

.section .text
.globl _start
_start:
    mov edi, 0              # stdin
    lea rsi, [rip + buffer]
    mov edx, 128

.read_loop:
    mov eax, 0              # SYS_read
    syscall
    cmp rax, 0
    je .done                # EOF
    js .error               # -errno

    sub rdx, rax
    add rsi, rax
    test rdx, rdx
    jg .read_loop
```

```
.done:
    mov eax, 60
    xor edi, edi
    syscall

.error:
    mov eax, 60
    mov edi, 1
    syscall
```

3.3.2 Partial Writes

```
.intel_syntax noprefix
# rdi = fd, rsi = buffer, rdx = length

.write_loop:
    mov eax, 1              # SYS_write
    syscall
    js .fail
    sub rdx, rax
    add rsi, rax
    test rdx, rdx
    jg .write_loop
    ret

.fail:
    ret
```

3.3.3 Importance of Looping

- Terminal input
- Pipes and sockets

- Network streams
- Large or non-blocking I/O

Partial handling is not optional. At the syscall level, correctness must be explicitly implemented.

3.3.4 Summary

- Partial I/O is normal and expected.
- The return value in RAX must always be checked.
- Buffer pointers and remaining byte counts must be updated.
- Correct looping is essential for reliability.

Chapter 4

File Operations

4.1 `open`, `close`, and `lseek`

File operations in Linux revolve around manipulating **file descriptors**, which are small integers representing open files or I/O endpoints. The system calls `openat`, `close`, and `lseek` form the foundation for accessing files, releasing kernel resources, and repositioning the file offset associated with a file descriptor.
All of these operations are mediated by the kernel's Virtual File System (VFS) layer, making them independent of the underlying filesystem type or storage device.

4.1.1 `openat`

On modern Linux systems, `openat` is the canonical interface for opening files. It supports directory-relative path resolution and safer file-handling patterns than the legacy `open` system call.

System call number

SYS_openat = 257

Register usage

- RAX — system call number (257)
- RDI — directory file descriptor (AT_FDCWD = -100 for current working directory)
- RSI — pointer to pathname
- RDX — flags (access mode, creation, truncation, etc.)
- R10 — file mode (permissions), used only with O_CREAT

Example: open a file for writing (create if it does not exist)

```
.intel_syntax noprefix

.section .rodata
filepath:
    .ascii "example.txt\0"

.section .text
.globl _start
_start:
    mov eax, 257            # SYS_openat
    mov edi, -100           # AT_FDCWD
    lea rsi, [rip + filepath]
    mov edx, 0x41           # O_WRONLY | O_CREAT
    mov r10d, 0644          # permissions: rw-r--r--
    syscall                 # rax = fd or -errno

    mov r12, rax            # save file descriptor
```

On success, **RAX** contains a non-negative file descriptor.

4.1.2 `close`

Every open file descriptor consumes kernel resources and must be explicitly released using `close`. Failing to close descriptors leads to resource leaks and can prevent filesystem cleanup.

System call number

`SYS_close = 3`

Register usage

- `RAX` — system call number (3)
- `RDI` — file descriptor to close

```
.intel_syntax noprefix

mov eax, 3                  # SYS_close
mov edi, r12                # close saved fd
syscall
```

After closing, the descriptor is invalid. Any further use results in `-EBADF`.

4.1.3 `lseek` — Repositioning the File Offset

The `lseek` system call adjusts the current file offset associated with a file descriptor. It does not perform any data I/O; instead, it determines where subsequent reads or writes will occur.

System call number

`SYS_lseek = 8`

Register usage

- `RAX` — system call number (8)
- `RDI` — file descriptor
- `RSI` — signed 64-bit offset
- `RDX` — origin (`SEEK_SET = 0`, `SEEK_CUR = 1`, `SEEK_END = 2`)

Example: move file offset forward by 32 bytes

```
.intel_syntax noprefix

mov eax, 8                  # SYS_lseek
mov edi, r12                # fd
mov rsi, 32                 # offset
mov edx, 1                  # SEEK_CUR
syscall                     # rax = new offset or -errno
```

Example: rewind file to the beginning

```
.intel_syntax noprefix

mov eax, 8                  # SYS_lseek
mov edi, r12                # fd
xor rsi, rsi                # offset = 0
mov edx, 0                  # SEEK_SET
syscall
```

4.1.4 Operational Behavior

- `openat` initializes the file offset to zero.

- `lseek` applies only to seekable descriptors (regular files and some character devices). Pipes and sockets return `-ESPIPE`.
- `close` decrements the kernel reference count; when no references remain, the file is eligible for cleanup.

4.1.5 Summary

- `openat` acquires file descriptors and configures access.
- `close` releases kernel resources.
- `lseek` repositions the file offset for controlled I/O.
- These syscalls form the core interface between user processes and the kernel filesystem layer.

4.2 Reading and Writing Binary Data

Binary file operations use the same primitives as text I/O: `read` and `write`. The kernel treats all data as **raw bytes** and performs no interpretation, alignment, or encoding.

4.2.1 Reading Fixed-Size Binary Blocks

```
.intel_syntax noprefix

.section .bss
binbuf:
    .skip 16

.section .text
# rdi = file descriptor

read_16:
    lea rsi, [rip + binbuf]
    mov edx, 16
.read_loop:
    mov eax, 0              # SYS_read
    syscall
    cmp rax, 0
    je .eof
    js .error
    sub rdx, rax
    add rsi, rax
    test rdx, rdx
    jg .read_loop
    ret

.eof:
```

```
.error:
    ret
```

4.2.2 Writing Binary Data

```
.intel_syntax noprefix
# rdi = fd, rsi = buffer, rdx = length

write_loop:
    mov eax, 1              # SYS_write
    syscall
    js .fail
    sub rdx, rax
    add rsi, rax
    test rdx, rdx
    jg write_loop
    ret

.fail:
    ret
```

4.2.3 Copying Binary Data Between Files

```
.intel_syntax noprefix

.section .bss
buffer:
    .skip 256

.section .text
# rdi = input fd, rsi = output fd

copy_loop:
```

```
    mov eax, 0              # SYS_read
    lea rdx, [rip + buffer]
    mov edx, 256
    syscall
    cmp rax, 0
    jle .done

    mov rcx, rax
    lea rdx, [rip + buffer]
.write_chunk:
    mov eax, 1              # SYS_write
    syscall
    js .done
    sub rcx, rax
    add rdx, rax
    test rcx, rcx
    jg .write_chunk
    jmp copy_loop

.done:
    ret
```

4.2.4 Alignment and Endianness

Binary data interpretation depends entirely on application logic:

- x86-64 is little-endian.
- Integer size and layout must be defined by the program.
- Network data requires explicit byte-order conversion.

4.2.5 Summary

- Text and binary I/O share the same syscall interface.
- Partial reads and writes are normal and must be handled explicitly.
- Buffer management and data interpretation are user-space responsibilities.

4.3 Creating and Truncating Files

File creation and truncation are controlled by flags passed to `openat` or by the `ftruncate` system call. These operations modify filesystem metadata and data extent.

4.3.1 Creating a New File

```
.intel_syntax noprefix

.section .rodata
fname:
    .ascii "newfile.bin\0"

.section .text
.globl _start
_start:
    mov eax, 257            # SYS_openat
    mov edi, -100           # AT_FDCWD
    lea rsi, [rip + fname]
    mov edx, 0x241          # O_WRONLY  O_CREATO_EXCL
    mov r10d, 0644
    syscall                 # rax = fd or -errno

    mov r12, rax
```

4.3.2 Truncating on Open

```
.intel_syntax noprefix

.section .rodata
fname2:
    .ascii "reset.log\0"
```

```
.section .text
truncate_on_open:
    mov eax, 257            # SYS_openat
    mov edi, -100
    lea rsi, [rip + fname2]
    mov edx, 0x201          # O_WRONLY | O_TRUNC
    mov r10d, 0644
    syscall
    ret
```

4.3.3 Truncating via `ftruncate`

```
.intel_syntax noprefix
# rdi = fd, rsi = new length

mov eax, 77               # SYS_ftruncate
syscall
```

4.3.4 Summary

- File creation and truncation are controlled by flags or `ftruncate`.
- `O_CREAT` and `O_EXCL` prevent accidental overwrite.
- `O_TRUNC` clears file contents on open.
- These mechanisms form the basis of safe and controlled file lifecycle management.

Chapter 5

Process Management

5.1 `fork` and Process Duplication

Linux process creation is historically performed using the `fork` system call, which **duplicates the calling process** to form a parent–child pair. While modern kernels implement this internally using the more general `clone` mechanism, the observable semantics of `fork` remain stable and central to UNIX process control.
The child process receives a near-identical execution context: virtual memory mappings, register state, open file descriptors, and signal handlers. Physical memory is not immediately duplicated; instead, Linux uses **Copy-on-Write (CoW)** so that pages are copied only if one process modifies them.

5.1.1 Return Value and Control-Flow Divergence

The defining characteristic of `fork` is that it returns twice:

- In the **parent**, `fork` returns the child's PID.

- In the **child**, `fork` returns **zero**.

This return-value divergence allows parent and child to follow different execution paths using ordinary control flow.

5.1.2 System Call Interface

On Linux x86-64, the system call number is:

$$\texttt{SYS\_fork} = 57$$

5.1.3 Minimal `fork` Example (Direct Syscall)

```
.intel_syntax noprefix

.section .text
.globl _start
_start:
    mov eax, 57              # SYS_fork
    syscall                  # returns twice

    test rax, rax
    je child_branch          # rax == 0 → child

# Parent branch
parent_branch:
    mov edi, eax             # child PID as exit code (illustrative)
    mov eax, 60              # SYS_exit
    syscall

child_branch:
    xor edi, edi             # exit code 0
    mov eax, 60
```

```
    syscall
```

The kernel makes no guarantees about whether the parent or child runs first after the call returns.

5.1.4 Clone-Based Implementation (Conceptual)

Internally, Linux implements `fork` using `clone` with a carefully chosen flag set. User programs should not attempt to re-create `fork` manually via `clone` unless they fully understand the ABI.
For reference, the syscall interface for `clone` is:

```
rax = 56        # SYS_clone
rdi = flags
rsi = child_stack
rdx = parent_tidptr
r10 = child_tidptr
r8  = tls
```

The kernel supplies appropriate flags internally to ensure correct `fork` semantics, including signal delivery and CoW behavior.

5.1.5 Resource Sharing After `fork`

- Virtual memory pages are shared read-only until modified.
- Open file descriptors are shared and refer to the same open file descriptions.
- File offsets may diverge if one process performs a seek.
- Parent and child are independently scheduled.

5.1.6 Summary

- `fork` duplicates a process logically, not physically.
- Copy-on-Write enables efficient process creation.
- Parent and child diverge based on the return value.
- `fork` underlies pipelines, shells, and service models.

5.2 `execve` and Program Replacement

While `fork` duplicates a process, `execve` **replaces** the current process image with a new executable. No new process is created; the calling process retains its PID and identity while its memory image is discarded and replaced.

5.2.1 System Call Interface

`SYS_execve = 59`

- `RDI` — pointer to pathname
- `RSI` — pointer to `argv[]` (NULL-terminated)
- `RDX` — pointer to `envp[]` (NULL-terminated)

On success, `execve` **does not return**.

5.2.2 Minimal `execve` Example

```
.intel_syntax noprefix

.section .rodata
path:
    .ascii "/bin/echo\0"
arg:
    .ascii "Hello from execve\0"

argv:
    .quad arg
    .quad 0
```

```
envp:
    .quad 0

.section .text
.globl _start
_start:
    mov eax, 59              # SYS_execve
    lea rdi, [rip + path]
    lea rsi, [rip + argv]
    lea rdx, [rip + envp]
    syscall                  # does not return on success

    # execve failed
    mov edi, 1
    mov eax, 60
    syscall
```

5.2.3 Preserved vs Replaced State

Replaced:

- User memory (code, heap, stack)
- Instruction pointer and registers

Preserved:

- PID and parent relationship
- Open file descriptors (unless `FD_CLOEXEC`)
- Scheduling parameters

5.2.4 Classic `fork` → `execve` Pattern

```
.intel_syntax noprefix

.section .rodata
cmd:
    .ascii "/bin/date\0"

argv:
    .quad cmd
    .quad 0

envp:
    .quad 0

.section .text
.globl _start
_start:
    mov eax, 57              # SYS_fork
    syscall
    test rax, rax
    je child

# Parent exits
    xor edi, edi
    mov eax, 60
    syscall

child:
    mov eax, 59              # SYS_execve
    lea rdi, [rip + cmd]
    lea rsi, [rip + argv]
    lea rdx, [rip + envp]
```

```
    syscall

    mov edi, 1
    mov eax, 60
    syscall
```

5.2.5 Summary

- `execve` replaces the process image.
- Success implies no return.
- Typically used after `fork`.

5.3 `wait` and Process Synchronization

When a child process exits, the kernel retains minimal metadata until the parent acknowledges it via a `wait` syscall. Failure to do so produces **zombie processes**.

5.3.1 `wait4` System Call

`SYS_wait4 = 61`

- `RDI` — PID to wait for (-1 = any child)
- `RSI` — pointer to status integer (or NULL)
- `RDX` — options (0 or `WNOHANG`)
- `R10` — pointer to `struct rusage` (or NULL)

5.3.2 Minimal `fork` + `wait4` Example

```
.intel_syntax noprefix

.section .bss
status:
    .skip 4

.section .text
.globl _start
_start:
    mov eax, 57              # SYS_fork
    syscall
    test rax, rax
    je child
```

```
# Parent
    mov edi, -1
    lea rsi, [rip + status]
    xor edx, edx
    xor r10d, r10d
    mov eax, 61               # SYS_wait4
    syscall

    xor edi, edi
    mov eax, 60
    syscall

child:
    mov edi, 5
    mov eax, 60
    syscall
```

5.3.3 Decoding Exit Status

The exit code is stored in bits 8–15:

```
.intel_syntax noprefix

mov eax, dword ptr [status]
shr eax, 8
```

5.3.4 Non-Blocking Wait

`WNOHANG = 1`

- `rax = 0` → child still running
- `rax > 0` → child reaped

5.3.5 Why Waiting Matters

- Zombies consume PID slots and kernel task structures
- Long-running parents must reap children
- Essential for shells, daemons, and supervisors

5.3.6 Summary

- `fork` creates a child
- `execve` replaces its program
- `wait` reaps the result
- This lifecycle defines UNIX process management

Chapter 6

Signals

6.1 Signal Types and Default Actions

Signals provide a mechanism for **asynchronous event notification** within Linux. A signal is a small numeric identifier delivered to a process or thread to indicate that a specific event has occurred, such as user interruption, illegal execution, memory faults, or explicit termination requests.

Signals do not carry payload data in their basic form. Instead, their semantics are defined by the kernel and by established UNIX conventions. Each signal has a **default action**, which is taken if the process does not explicitly override it.

Default actions include:

- Terminating the process
- Terminating and generating a core dump
- Ignoring the signal
- Stopping or resuming execution

Most signals may be handled or ignored by user code. However, `SIGKILL` and `SIGSTOP` are **enforced by the kernel** and cannot be caught, blocked, or overridden.

6.1.1 Common Signal Classes

Termination Signals

Used to request orderly shutdown, such as `SIGTERM` and `SIGHUP`.

Fatal Error Signals

Generated by illegal execution conditions, such as `SIGSEGV` (invalid memory access) or `SIGILL` (illegal instruction). These usually cause termination with a core dump.

Control Signals

Used for job control, including `SIGSTOP` and `SIGCONT`.

User-Defined Signals

`SIGUSR1` and `SIGUSR2` are reserved for application-defined meaning.

6.1.2 Default Behavior Examples

Signal	Meaning	Default Action
`SIGTERM`	Termination request	Terminate
`SIGINT`	Keyboard interrupt (`Ctrl+C`)	Terminate
`SIGKILL`	Forced termination	Terminate (cannot be changed)
`SIGSEGV`	Invalid memory access	Terminate + core dump
`SIGSTOP`	Stop execution	Stop (cannot be changed)
`SIGCHLD`	Child state change	Ignore or notify parent

6.1.3 Sending Signals Programmatically

Signals are delivered using the `kill` system call.

`SYS_kill = 62`

Register interface

- `RDI` — target PID
- `RSI` — signal number

Example: sending `SIGTERM` to a process

```
.intel_syntax noprefix

.section .text
.globl _start
_start:
    mov eax, 62          # SYS_kill
    mov edi, 1234        # target PID
    mov esi, 15          # SIGTERM
    syscall

    xor edi, edi
    mov eax, 60          # SYS_exit
    syscall
```

6.1.4 Signals Generated by Hardware

Some signals originate from CPU exceptions rather than explicit user requests:

- **#PF** → `SIGSEGV` (page fault)
- **#UD** → `SIGILL` (illegal instruction)
- **#DE** → `SIGFPE` (divide by zero)

These are delivered when control is about to return to user space.

6.1.5 Delivery Semantics

Signals are delivered only at safe execution points:

- On return from system calls
- On return from interrupts
- When transitioning from kernel to user mode

The kernel never interrupts an instruction mid-execution.

6.1.6 Summary

- Signals provide asynchronous notifications to processes
- Each signal has a kernel-defined default action
- `SIGKILL` and `SIGSTOP` cannot be intercepted
- Signals originate from users, hardware, or kernel scheduling

6.2 Installing Signal Handlers

Processes may override default signal behavior by installing custom handlers using `rt_sigaction`. Signal handling is not a normal function call; it involves kernel-controlled stack manipulation and state restoration.

6.2.1 `rt_sigaction` Interface

`SYS_rt_sigaction = 13`

Registers

- `RDI` — signal number
- `RSI` — pointer to `struct sigaction`
- `RDX` — pointer to old action or NULL
- `R10` — size of signal mask (8 bytes on x86-64)

6.2.2 Minimal Signal Handler Example

```
.intel_syntax noprefix

.section .rodata
msg:
    .ascii "Interrupt received\n"
msg_len = . - msg

.section .bss
    .align 8
act:
```

```
    .skip 32          # struct sigaction (kernel ABI)

.section .text
.globl _start

sig_handler:
    mov eax, 1        # SYS_write
    mov edi, 1
    lea rsi, [rip + msg]
    mov edx, msg_len
    syscall

    mov eax, 15       # SYS_rt_sigreturn
    syscall

_start:
    lea rax, [rip + sig_handler]
    mov [act + 0], rax        # sa_handler
    mov qword ptr [act + 8], 0    # sa_flags
    mov qword ptr [act + 16], 0   # sa_mask (low)
    mov qword ptr [act + 24], 0   # sa_mask (high)

    mov eax, 13               # SYS_rt_sigaction
    mov edi, 2                # SIGINT
    lea rsi, [rip + act]
    xor edx, edx
    mov r10d, 8
    syscall

.wait:
    pause
    jmp .wait
```

6.2.3 Signal Delivery Mechanics

1. Kernel builds a signal frame on the user stack
2. Execution is redirected to the handler
3. Handler executes in user mode
4. Handler exits via `rt_sigreturn`
5. Kernel restores original register state

Returning with `ret` is invalid.

6.2.4 Safety Constraints

- Handlers are asynchronous
- Only async-signal-safe operations are allowed
- Direct syscalls are safest
- Non-reentrant library calls are dangerous

6.2.5 Summary

- Signal handlers override default behavior
- Kernel controls stack and state restoration
- Handlers must be minimal and safe

6.3 `kill` and Self-Signaling

The `kill` system call delivers a signal to a process or process group. It does not imply termination.

6.3.1 PID Semantics

- `pid > 0`: send to specific process
- `pid = 0`: send to current process group
- `pid = -1`: send to all permitted processes

6.3.2 Self-Signaling Example

```
.intel_syntax noprefix

.section .text
.globl _start
_start:
    mov eax, 62          # SYS_kill
    mov edi, 0           # current process group
    mov esi, 10          # SIGUSR1
    syscall

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

6.3.3 Thread-Targeted Signals

For multithreaded programs, `tgkill` targets individual threads.

```
SYS_tgkill = 234
```

6.3.4 Summary

- `kill` delivers signals, not termination
- Signals may be sent to self, groups, or threads
- Self-signaling enables controlled event handling
- Proper signal usage is essential for daemons and supervisors

Chapter 7

Environment and Arguments

7.1 Accessing `argc`, `argv`, and `envp`

When a program is loaded by the kernel via `execve`, the initial user-mode stack is constructed according to a strict ABI-defined layout. Before execution begins at the entry point (`_start`), the kernel places the argument count, argument vector, and environment vector onto the stack.
These values are **not passed in registers**. The entry point must retrieve them directly from memory. Normally, the C runtime performs this extraction and then calls `main(argc, argv, envp)`. In low-level programs or freestanding binaries, this work must be done manually.

7.1.1 Initial Stack Layout

At process entry, the stack has the following structure:

```
rsp -> argc
        argv[0]
```

```
        argv[1]
        ...
        argv[argc - 1]
        NULL
        envp[0]
        envp[1]
        ...
        envp[n]
        NULL
        auxiliary vector entries...
```

Each pointer is 8 bytes on x86-64. Both `argv` and `envp` are NULL-terminated arrays of pointers to NUL-terminated strings.

7.1.2 Retrieving `argc`, `argv`, and `envp`

The following example demonstrates how to extract all three values at `_start` using pure assembly.

```
.intel_syntax noprefix

.section .text
.globl _start
_start:
    mov rbx, rsp            # preserve initial stack pointer

    mov rax, [rbx]          # rax = argc
    lea rdi, [rbx + 8]      # rdi = &argv[0]

    # Compute envp:
    # envp = argv + (argc + 1)
    mov rcx, rax
    add rcx, 1              # include NULL terminator
```

```
    shl rcx, 3              # (argc + 1) * sizeof(void*)
    lea rsi, [rdi + rcx]    # rsi = envp

    # rax = argc
    # rdi = argv
    # rsi = envp
```

This establishes a calling context equivalent to:

```
main(argc, argv, envp)
```

7.1.3 Iterating Over Command-Line Arguments

The following example prints each command-line argument to standard output using direct system calls.

```
.intel_syntax noprefix

.section .rodata
newline:
    .byte 10

.section .text
.globl _start
_start:
    mov rbx, rsp
    mov rcx, [rbx]          # argc
    lea rdi, [rbx + 8]      # argv pointer

.arg_loop:
    test rcx, rcx
    je .done
```

```
    mov rsi, [rdi]          # pointer to argv[i]

    # compute string length
    xor rdx, rdx
.len_loop:
    cmp byte ptr [rsi + rdx], 0
    je .write
    inc rdx
    jmp .len_loop

.write:
    mov eax, 1              # SYS_write
    mov edi, 1              # stdout
    syscall

    # write newline
    mov eax, 1
    mov edi, 1
    lea rsi, [rip + newline]
    mov edx, 1
    syscall

    add rdi, 8              # next argv pointer
    dec rcx
    jmp .arg_loop

.done:
    xor edi, edi
    mov eax, 60             # SYS_exit
    syscall
```

This code performs raw traversal of the argument vector without relying on libc.

7.1.4 Accessing the Environment Vector

The environment vector is structurally identical to `argv`: a NULL-terminated array of pointers to strings.

```
.intel_syntax noprefix

# rsi already contains envp from earlier extraction
mov rbx, rsi                # rbx = envp
```

From this point, environment traversal follows the same logic as argument traversal.

7.1.5 Summary

- The kernel constructs `argc`, `argv`, and `envp` on the initial user stack.
- No registers are used for argument passing at `_start`.
- Both `argv` and `envp` are NULL-terminated arrays of pointers to NUL-terminated strings.
- Manual extraction is required in freestanding or libc-free programs.

7.2 Modifying the Environment

The environment is a purely user-space data structure. The kernel provides **no system calls** to modify it after process startup. All changes occur through user-space memory manipulation or library helpers.

7.2.1 Providing a Custom Environment at `execve`

The recommended method for controlling a process environment is to supply a custom `envp` vector during `execve`.

```
.intel_syntax noprefix

.section .rodata
path:
    .ascii "/usr/bin/env\0"
arg0:
    .ascii "env\0"

env1:
    .ascii "FOO=one\0"
env2:
    .ascii "BAR=two\0"

argv:
    .quad arg0
    .quad 0

envp:
    .quad env1
    .quad env2
    .quad 0
```

```
.section .text
.globl _start
_start:
    mov eax, 59              # SYS_execve
    lea rdi, [rip + path]
    lea rsi, [rip + argv]
    lea rdx, [rip + envp]
    syscall

    mov edi, 1
    mov eax, 60
    syscall
```

7.2.2 Using libc Helpers

Functions such as `setenv` and `unsetenv` are implemented entirely in user space. They manage internal bookkeeping and memory allocation safely and should be preferred when linking against libc.

```
.intel_syntax noprefix

.extern setenv

.section .rodata
key:
    .ascii "MYKEY\0"
val:
    .ascii "VALUE\0"

.section .text
.globl _start
_start:
```

```
    mov rdi, key
    mov rsi, val
    mov edx, 1              # overwrite
    call setenv

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

7.2.3 In-Place Modification (Advanced and Risky)

Direct mutation of the initial `envp` array is possible but dangerous. libc may copy or relocate environment data, and overwriting strings risks memory corruption.

```
.intel_syntax noprefix

.section .rodata
new_env:
    .ascii "LANG=xx_XX.UTF-8\0"

.section .text
.globl _start
_start:
    mov rbx, rsp
    mov rcx, [rbx]          # argc
    lea rdi, [rbx + 8]      # argv
    mov rax, rcx
    add rax, 1
    shl rax, 3
    lea rsi, [rdi + rax]    # envp

    lea rdx, [rip + new_env]
    mov [rsi], rdx          # replace envp[0]
```

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

7.2.4 Inheritance Semantics

- `fork` inherits the environment via Copy-on-Write memory.
- `execve` constructs a fresh environment from the supplied `envp`.
- The kernel never modifies environment variables after process startup.

7.2.5 Summary

- Environment handling is entirely a user-space responsibility.
- Prefer explicit `envp` construction at `execve`.
- Use libc helpers for in-process modification.
- Avoid manual pointer or string mutation unless you fully control runtime behavior.

7.3 Argument Passing to Executed Programs

Arguments supplied to `execve` define the new program's command line. The kernel copies the provided strings and constructs a new stack layout for the replacement process.

7.3.1 Rules for `argv`

- `argv` must be a NULL-terminated array of pointers.
- Each pointer must reference a valid NUL-terminated string.
- `argv[0]` conventionally names the program.
- The kernel performs minimal validation.

7.3.2 Minimal Example

```
.intel_syntax noprefix

.section .rodata
path:
    .ascii "/bin/echo\0"
arg0:
    .ascii "echo\0"
arg1:
    .ascii "Hello from execve\0"

argv:
    .quad arg0
    .quad arg1
    .quad 0

envp:
```

```
    .quad 0

.section .text
.globl _start
_start:
    mov eax, 59              # SYS_execve
    lea rdi, [rip + path]
    lea rsi, [rip + argv]
    lea rdx, [rip + envp]
    syscall

    mov edi, 1
    mov eax, 60
    syscall
```

7.3.3 Kernel Stack Reconstruction

During `execve`, the kernel:

- Allocates a fresh user stack
- Copies argument and environment strings
- Builds `argc`, `argv`, and `envp`
- Transfers execution to the new ELF entry point

7.3.4 Summary

- The caller fully controls arguments passed to a new program.
- The kernel reconstructs the initial stack from supplied vectors.
- Proper NULL termination is mandatory.

- This mechanism underlies all UNIX process execution models.

Chapter 8

Simple Command Loop

8.1 Reading User Commands

Reading user commands reliably at the syscall boundary requires handling several realities of Unix I/O:

- Syscalls are register-based: `read` uses `rax=0`, `rdi=fd`, `rsi=buf`, `rdx=len`.
- Partial reads are normal; a complete line may arrive in fragments.
- EOF (Ctrl-D) is indicated by `read` returning 0.
- Line endings may be `\n` or `\r\n`.
- Buffers must be bounded to avoid overflow.
- `read` may be interrupted by signals (`-EINTR`) and should be retried.

The following implementation prints a prompt, reads a single line into a fixed buffer, strips newline and carriage return, and returns a NUL-terminated string suitable for tokenization.

8.1.1 Assembly: Reliable Line Reader and Loop

```
.intel_syntax noprefix

.section .rodata
prompt:
    .ascii "$ "
prompt_len = . - prompt

eof_msg:
    .ascii "EOF, exiting.\n"
eof_len = . - eof_msg

err_msg:
    .ascii "Read error, exiting.\n"
err_len = . - err_msg

.section .bss
line_buf:
    .skip 1024

.section .text
.globl _start

# syscall numbers
SYS_READ  = 0
SYS_WRITE = 1
SYS_EXIT  = 60

STDIN  = 0
STDOUT = 1

_start:
```

```
main_loop:
    # print prompt
    mov eax, SYS_WRITE
    mov edi, STDOUT
    lea rsi, [rip + prompt]
    mov edx, prompt_len
    syscall

    # read line into buffer
    lea rdi, [rip + line_buf]
    mov esi, 1024
    call read_line

    test rax, rax
    je handle_eof
    js handle_error

    # echo line back
    mov edx, eax
    lea rsi, [rip + line_buf]
    mov eax, SYS_WRITE
    mov edi, STDOUT
    syscall

    jmp main_loop

handle_eof:
    mov eax, SYS_WRITE
    mov edi, STDOUT
    lea rsi, [rip + eof_msg]
    mov edx, eof_len
    syscall
```

```
    xor edi, edi
    mov eax, SYS_EXIT
    syscall

handle_error:
    mov eax, SYS_WRITE
    mov edi, STDOUT
    lea rsi, [rip + err_msg]
    mov edx, err_len
    syscall

    mov edi, 1
    mov eax, SYS_EXIT
    syscall

# ------------------------------------------------
# read_line(buffer, size)
# rdi = buffer pointer
# rsi = buffer size
# returns:
#   rax > 0 : length
#   rax = 0 : EOF
#   rax < 0 : error
# ------------------------------------------------
read_line:
    push rbx
    push r12
    push r13

    mov r12, rdi        # buffer base
    mov r13, rsi        # capacity
    xor rbx, rbx        # bytes accumulated
```

```
.read_more:
    mov eax, SYS_READ
    mov edi, STDIN
    lea rsi, [r12 + rbx]
    mov rdx, r13
    syscall

    cmp rax, 0
    jl .error
    je .eof

    mov rcx, rax
    mov rsi, r12
    add rsi, rbx

.scan:
    test rcx, rcx
    je .no_nl
    mov al, byte ptr [rsi]
    cmp al, 10
    je .found_nl
    inc rsi
    dec rcx
    jmp .scan

.found_nl:
    mov byte ptr [rsi], 0
    cmp rsi, r12
    je .done
    cmp byte ptr [rsi-1], 13
    jne .done
    mov byte ptr [rsi-1], 0
```

```
.done:
    mov rax, rsi
    sub rax, r12
    jmp .ret

.no_nl:
    add rbx, rax
    sub r13, rax
    cmp r13, 1
    jg .read_more

    lea rsi, [r12 + rbx - 1]
    mov byte ptr [rsi], 0
    mov rax, rbx
    jmp .ret

.eof:
    test rbx, rbx
    je .ret_zero
    lea rsi, [r12 + rbx]
    mov byte ptr [rsi], 0
    mov rax, rbx
    jmp .ret

.ret_zero:
    xor rax, rax
    jmp .ret

.error:
    # rax already holds -errno
.ret:
    pop r13
    pop r12
```

```
    pop rbx
    ret
```

8.1.2 Tokenizing Input

Tokenization replaces whitespace with NUL bytes and builds an `argv` pointer array.

```
.intel_syntax noprefix

# tokenize_line(buffer, argv, max)
# rdi = buffer
# rsi = argv array
# rdx = max args
# returns rax = argc

tokenize_line:
    xor rcx, rcx
    mov r8, rdi

.skip:
    mov al, byte ptr [r8]
    test al, al
    je .finish
    cmp al, ' '
    je .next
    cmp al, 9
    je .next
    jmp .start

.next:
    inc r8
    jmp .skip
```

```
.start:
    cmp rcx, rdx
    jae .finish
    mov [rsi + rcx*8], r8
    inc rcx

.scan:
    mov al, byte ptr [r8]
    test al, al
    je .finish
    cmp al, ' '
    je .term
    cmp al, 9
    je .term
    inc r8
    jmp .scan

.term:
    mov byte ptr [r8], 0
    inc r8
    jmp .skip

.finish:
    mov qword ptr [rsi + rcx*8], 0
    mov rax, rcx
    ret
```

8.1.3 Running Commands

```
.intel_syntax noprefix

.section .bss
child_status:
```

```
    .skip 4

.section .text

# run_command(argv, argc)
# rdi = argv
# rax = argc

run_command:
    test rax, rax
    je .ret

    mov rsi, rdi

    mov eax, 57             # fork
    syscall

    test rax, rax
    je .child

.parent:
    mov edi, -1
    lea rsi, [rip + child_status]
    xor edx, edx
    xor r10d, r10d
    mov eax, 61             # wait4
    syscall
    ret

.child:
    mov rdi, [rsi]          # argv[0]
    xor rdx, rdx            # envp = inherit
    mov eax, 59             # execve
```

```
    syscall

    mov edi, 1
    mov eax, 60
    syscall

.ret:
    ret
```

8.1.4 Summary

- Input handling must tolerate partial reads, EOF, and signals.
- Tokenization is performed in-place with no allocations.
- Command execution follows the classic `fork → exec → wait` model.
- This forms the minimal, correct foundation of a Unix shell.

Chapter 9

Building a Mini Shell

9.1 Command Execution

In a minimal shell, *command execution* means taking a parsed `argv[]` vector and launching the corresponding program in a **separate process**. The shell itself must survive execution, so commands are always run using the classic:

`fork → execve → wait`

This section assumes that:

- `argv[]` is a NULL-terminated vector of pointers.
- `argv[0]` is the command name or path.
- `envp[]` is already available to the shell.

9.1.1 Execution Without `PATH` Searching

Initially, the shell executes only **explicit paths**:

- Absolute paths (`/bin/ls`)
- Relative paths (`./a.out`)

This keeps the execution model minimal and unambiguous.

—

9.1.2 Minimal Executor (No PATH Search)

```
.intel_syntax noprefix

.section .bss
child_status:
    .skip 4

.section .text

# execute_command(argv, argc, envp)
# rdi = argv
# rax = argc
# rsi = envp

execute_command:
    test    rax, rax
    je      .ret

    mov     rbx, rdi        # save argv
    mov     r12, rsi        # save envp

    mov     eax, 57         # sys_fork
    syscall

    test    rax, rax
```

```
    je      .child

.parent:
    mov     edi, -1
    lea     rsi, [rip + child_status]
    xor     edx, edx
    xor     r10d, r10d
    mov     eax, 61         # sys_wait4
    syscall
    ret

.child:
    mov     rdi, [rbx]      # argv[0]
    mov     rsi, rbx        # argv
    mov     rdx, r12        # envp
    mov     eax, 59         # sys_execve
    syscall

    # execve failed → exit immediately
    mov     edi, 127
    mov     eax, 60
    syscall

.ret:
    ret
```

—

9.2 Path Resolution

The kernel does **not** search `$PATH`. If `argv[0]` contains no slash, the shell must search manually.

9.2.1 Finding `PATH` in `envp`

```
.intel_syntax noprefix

# find_path(envp)
# rdi = envp
# returns rax = pointer to PATH value or 0

find_path:
    mov     rbx, rdi

.next:
    mov     rax, [rbx]
    test    rax, rax
    je      .not_found

    cmp     dword ptr [rax], 0x54414850   # "PATH"
    jne     .advance
    cmp     byte ptr [rax+4], '='
    jne     .advance

    lea     rax, [rax+5]
    ret

.advance:
    add     rbx, 8
    jmp     .next

.not_found:
    xor     rax, rax
    ret
```

—

9.2.2 PATH-Based Execution

```
.intel_syntax noprefix

.section .bss
path_buf:
    .skip 512

.section .text

# try_exec_path(argv, envp)
# rdi = argv
# rsi = envp
# never returns on success

try_exec_path:
    mov     rbx, rdi        # argv
    mov     r12, rsi        # envp

    mov     rdi, rsi
    call    find_path
    test    rax, rax
    je      .fail

    mov     r13, rax        # PATH pointer
    mov     r14, [rbx]      # command name

.next_dir:
    mov     al, byte ptr [r13]
    test    al, al
    je      .fail

    lea     rdi, [rip + path_buf]
```

```
    mov     rsi, r13

.copy_dir:
    mov     al, byte ptr [rsi]
    cmp     al, ':'
    je      .dir_done
    test    al, al
    je      .dir_done
    mov     byte ptr [rdi], al
    inc     rsi
    inc     rdi
    jmp     .copy_dir

.dir_done:
    mov     byte ptr [rdi], '/'
    inc     rdi

.copy_cmd:
    mov     al, byte ptr [r14]
    mov     byte ptr [rdi], al
    inc     rdi
    inc     r14
    test    al, al
    jne     .copy_cmd

    # execve(candidate, argv, envp)
    lea     rdi, [rip + path_buf]
    mov     rsi, rbx
    mov     rdx, r12
    mov     eax, 59
    syscall

    # exec failed → advance PATH
```

```
    mov     r14, [rbx]
    mov     r13, rsi
    cmp     byte ptr [r13], ':'
    jne     .next_dir
    inc     r13
    jmp     .next_dir

.fail:
    ret
```

—

9.3 Error Handling

9.3.1 Child Failure Is Terminal

If `execve` returns, it has failed. The child must **never** return to shell logic.

```
.intel_syntax noprefix

exec_failed_exit:
    mov     edi, 127
    mov     eax, 60
    syscall
```

—

9.3.2 PATH-Aware Executor

```
.intel_syntax noprefix

# execute_shell_command(argv, argc, envp)
```

```
# rdi = argv
# rax = argc
# rsi = envp

execute_shell_command:
    test    rax, rax
    je      .ret

    mov     rbx, rdi
    mov     r12, rsi

    mov     eax, 57
    syscall

    test    rax, rax
    je      .child

.parent:
    mov     edi, -1
    lea     rsi, [rip + child_status]
    xor     edx, edx
    xor     r10d, r10d
    mov     eax, 61
    syscall
    ret

.child:
    mov     rdi, [rbx]

.scan:
    mov     al, byte ptr [rdi]
    test    al, al
    je      .search
```

```
    cmp     al, '/'
    je      .direct
    inc     rdi
    jmp     .scan

.direct:
    mov     rdi, [rbx]
    mov     rsi, rbx
    mov     rdx, r12
    mov     eax, 59
    syscall
    jmp     exec_failed_exit

.search:
    mov     rdi, rbx
    mov     rsi, r12
    call    try_exec_path
    jmp     exec_failed_exit

.ret:
    ret
```

—

9.4 Summary

- The shell must never call `execve` directly.
- All execution occurs in forked children.
- `PATH` searching is purely user-space logic.
- Failed `execve` must lead to immediate `_exit`.

- The parent always waits to avoid zombie processes.

This completes the **correct execution core** of a minimal POSIX shell.

Chapter 10

Final Project

10.1 A Fully Functional Minimal Shell (Expanded)

A shell is an interactive program that provides a **human-controlled entry point** into the operating system. Unlike fixed-function executables, a shell must:

1. **Read** text instructions from the user.
2. **Interpret** the request and form a runnable program invocation.
3. **Create a child process** to execute the requested program.
4. **Replace the child's memory image** using `execve`.
5. **Wait** for the child to terminate before reading the next command.

These steps are implemented entirely using **Linux system calls**, without reliance on libc or intermediate runtimes.
The minimal executable behavior of a shell reflects the fundamental user-process interaction model in Unix-like systems:

```
User  Terminal  Shell  Kernel  Executed Programs
```

10.1.1 Interactive Operation and Terminal Contract

The shell receives input via STDIN (file descriptor 0), typically a terminal or pseudoterminal. When interactive:

- The kernel handles line editing and canonical mode input under the terminal driver unless raw mode is explicitly enabled.
- The shell must respond *prompt-first*, otherwise the user receives no visible feedback.

10.1.2 Maintaining Process Identity and Control

It is critical that the shell **does not replace itself** when executing commands.
If it used `execve` directly:

```
shell → execve → new program replaces shell
```

The shell would disappear, losing the input loop.
Instead, the shell **forks**, duplicating the process, and only the **child** executes the external program:

```
shell
  fork()
      parent → wait → continue loop
      child → execve → external program
```

This allows persistent shell state, multiple commands, and controlled process cleanup.

10.1.3 PATH Resolution: Why It Is User-Space Responsibility

The Linux kernel executes only **explicit paths**. It does not search directories. The shell must resolve `$PATH` in user space.

10.1.4 Complete Minimal Shell Implementation

```
.intel_syntax noprefix

.equ SYS_READ,   0
.equ SYS_WRITE,  1
.equ SYS_FORK,   57
.equ SYS_EXECVE, 59
.equ SYS_WAIT4,  61
.equ SYS_EXIT,   60

.equ STDIN,  0
.equ STDOUT, 1

.section .data
prompt:
    .asciz "$ "
prompt_len = . - prompt

nf_msg:
    .ascii "command not found\n"
nf_len = . - nf_msg

.section .bss
line_buf:    .skip 1024
argv_buf:    .skip 64*8
child_stat:  .skip 4
path_tmp:    .skip 512

.section .text
.globl _start

read_line:
```

```
    mov     eax, SYS_WRITE
    mov     edi, STDOUT
    lea     rsi, [rip+prompt]
    mov     edx, prompt_len
    syscall

    mov     eax, SYS_READ
    mov     edi, STDIN
    lea     rsi, [rip+line_buf]
    mov     edx, 1024
    syscall
    test    rax, rax
    je      exit_shell
    dec     rax
    mov     byte ptr [line_buf+rax], 0
    ret

tokenize:
    xor     rcx, rcx
    mov     r8, rdi
.skip:
    mov     al, [r8]
    test    al, al
    je      .done
    cmp     al, ' '
    je      .adv
    cmp     al, 9
    je      .adv
    jmp     .tok
.adv:
    inc     r8
    jmp     .skip
.tok:
```

```
    mov     [rsi+rcx*8], r8
    inc     rcx
.scan:
    mov     al, [r8]
    test    al, al
    je      .done
    cmp     al, ' '
    je      .null
    cmp     al, 9
    je      .null
    inc     r8
    jmp     .scan
.null:
    mov     byte ptr [r8], 0
    inc     r8
    jmp     .skip
.done:
    mov     qword ptr [rsi+rcx*8], 0
    mov     rax, rcx
    ret

get_envp:
    mov     rax, [rsp]
    lea     rdi, [rsp+8]
.skip_argv:
    test    rax, rax
    je      .done
    add     rdi, 8
    dec     rax
    jmp     .skip_argv
.done:
    add     rdi, 8
    mov     rax, rdi
```

```
    ret

find_path:
.next:
    mov     rax, [rdi]
    test    rax, rax
    je      .none
    cmp     dword ptr [rax], 0x48544150
    jne     .cont
    cmp     byte ptr [rax+4], '='
    jne     .cont
    lea     rax, [rax+5]
    ret
.cont:
    add     rdi, 8
    jmp     .next
.none:
    xor     rax, rax
    ret

try_exec_path:
    push    rbx
    push    r12
    mov     rbx, rdi
    mov     r12, rsi
    mov     rdi, rsi
    call    find_path
    test    rax, rax
    je      .fail
    mov     r8, rax
    mov     r9, [rbx]
.next_dir:
    mov     al, [r8]
```

```
    test    al, al
    je      .fail
    lea     rdi, [rip+path_tmp]
    mov     rsi, r8
.copy:
    mov     al, [rsi]
    cmp     al, ':'
    je      .slash
    test    al, al
    je      .slash
    mov     [rdi], al
    inc     rsi
    inc     rdi
    jmp     .copy
.slash:
    mov     byte ptr [rdi], '/'
    inc     rdi
    mov     rsi, r9
.copyc:
    mov     al, [rsi]
    mov     [rdi], al
    inc     rsi
    inc     rdi
    test    al, al
    jne     .copyc
    lea     rdi, [rip+path_tmp]
    mov     rsi, rbx
    mov     rdx, r12
    mov     eax, SYS_EXECVE
    syscall
    mov     r8, rsi
    cmp     byte ptr [r8], ':'
    jne     .next_dir
```

```
    inc     r8
    jmp     .next_dir
.fail:
    pop     r12
    pop     rbx
    ret

run_cmd:
    test    rax, rax
    je      .ret
    push    rbx
    push    r12
    mov     rbx, rdi
    mov     r12, rsi
    mov     eax, SYS_FORK
    syscall
    test    rax, rax
    je      .child
.parent:
    mov     edi, -1
    lea     rsi, [rip+child_stat]
    xor     edx, edx
    xor     r10d, r10d
    mov     eax, SYS_WAIT4
    syscall
.ret:
    pop     r12
    pop     rbx
    ret
.child:
    mov     rdi, [rbx]
.scan:
    mov     al, [rdi]
```

```
    test    al, al
    je      .path
    cmp     al, '/'
    je      .direct
    inc     rdi
    jmp     .scan
.direct:
    mov     rdi, [rbx]
    mov     rsi, rbx
    mov     rdx, r12
    mov     eax, SYS_EXECVE
    syscall
.path:
    mov     rdi, rbx
    mov     rsi, r12
    call    try_exec_path
    mov     eax, SYS_WRITE
    mov     edi, STDOUT
    lea     rsi, [rip+nf_msg]
    mov     edx, nf_len
    syscall
    mov     eax, SYS_EXIT
    mov     edi, 127
    syscall

_start:
    mov     rbx, rsp
.loop:
    call    read_line
    lea     rdi, [rip+line_buf]
    lea     rsi, [rip+argv_buf]
    mov     rdx, 64
    call    tokenize
```

```
    mov     rdi, argv_buf
    mov     rsi, rbx
    call    get_envp
    mov     rsi, rax
    call    run_cmd
    jmp     .loop

exit_shell:
    mov     eax, SYS_EXIT
    xor     edi, edi
    syscall
```

Conclusion

This booklet has examined the interaction between user space and the Linux kernel from a low-level, system-call–oriented perspective, demonstrating how process control, file operations, environment management, and signal handling form the operational foundation of all Unix-like systems. By intentionally avoiding high-level abstractions and external runtime libraries, we exposed the direct mechanisms through which software requests services from the kernel and how the kernel enforces memory protection, privilege separation, and controlled execution.

Beginning with privilege levels and the user–kernel boundary, we established the architectural and security assumptions that make system stability possible. We then progressed through register-based system call conventions, return values, and error handling. This enabled practical work with core facilities such as file input/output, descriptor management, process creation, program replacement using execve, and synchronization through wait and signals.

The development of the minimal shell in the final chapters demonstrated how these mechanisms compose into a functioning software environment. The shell does not rely on implicit behavior or hidden layers; instead, it explicitly performs each step required to create child processes, search execution paths, launch programs, manage environment inheritance, and synchronize control flow. By constructing the shell from first principles, we confirmed that complex user-facing tools ultimately reduce to a small and rigorously defined set of system calls.

Although the programs built in this booklet are intentionally minimal, the concepts underlying

them scale to modern implementations of shells, service supervisors, container runtimes, init systems, and low-level debugging infrastructures. Any software that creates or manages processes must adhere to the same execution model illustrated here.

The mastery of Linux system calls is essential not because it encourages programming at a permanently low level, but because it clarifies the ground truth upon which higher-level abstractions depend. Understanding this layer of execution enables the developer to reason accurately about performance, reliability, security, debugging, memory behavior, and runtime control. It also provides the knowledge required to evaluate toolchains, languages, and frameworks with precision, selecting them according to the actual needs of the system rather than convention or assumption.

This booklet concludes with a complete, correct, and extensible model of user–kernel interaction. In subsequent work, these foundations can be extended to advanced topics including asynchronous I/O, inter-process communication, namespace isolation, cgroups, file descriptor passing, job control, dynamic loader behavior, relocatable binaries, and kernel module interactions.

The objective has not been to memorize system call numbers or replicate historical patterns, but to cultivate a structured mental model of how Linux operates at its lowest software-visible level. With that model established, the reader is now equipped to reason independently, implement tools with confidence, and advance into deeper areas of systems programming with clarity and rigor.

References

1. **The Linux Kernel Documentation**
   Linux Kernel Project.
   Detailed official descriptions of kernel internals, process handling, memory management, and system call interfaces. The documentation provides authoritative reference material regarding system behavior and kernel implementation details.

2. **Linux System Call Table (x86-64 ABI Specification)**
   System V Application Binary Interface — AMD64 Architecture Processor Supplement.
   Defines register usage conventions, calling rules, stack handling, data types, and execution transition behavior between user mode and kernel mode. Serves as the formal specification for system call invocation.

3. **Intel 64 and IA-32 Architectures Software Developer's Manual**
   Intel Corporation.
   Provides architectural descriptions of privilege levels, memory protection, paging structures, interrupt handling, and CPU execution semantics required to understand ring transitions and system call mechanisms.

4. **POSIX.1 Base Specifications — Process and Signal Interfaces**
   IEEE Standards Association.
   Defines portable standards for process creation, execution, environment handling, file

operations, descriptors, and signal semantics. Establishes the behavioral model adopted across Unix and Unix-like platforms.

5. **The GNU Assembler (GAS) Documentation — x86-64 Intel Syntax Mode**
   Free Software Foundation.
   Specifies instruction syntax, directive rules, symbol resolution, relocation behavior, and object file generation under the GNU assembler toolchain.

6. **Linux Programmer's Manual (man Sections 2 and 7)**
   Core system interfaces including:

   - Section 2: System Calls (e.g., `read`, `write`, `open`, `execve`, `fork`, `wait`, `kill`)
   - Section 7: Operating System Overview, capabilities, and environment variable behavior

   These serve as direct behavioral reference for how Linux exposes functionality to user space.

7. **ELF File Format Specification**
   Tool Interface Standards (TIS) Committee.
   Describes binary format layout, program headers, interpreter paths, symbol resolution basics, and runtime memory loading process that underpin `execve` behavior when replacing a process image.

8. **The Linux Process Model — Kernel Scheduling and Task Management**
   Contemporary Linux systems post-2020.
   Includes improved scheduler behavior, thread group semantics, namespace consistency, and refined `wait4` process state transitions relevant to interactive shell execution and child process supervision.