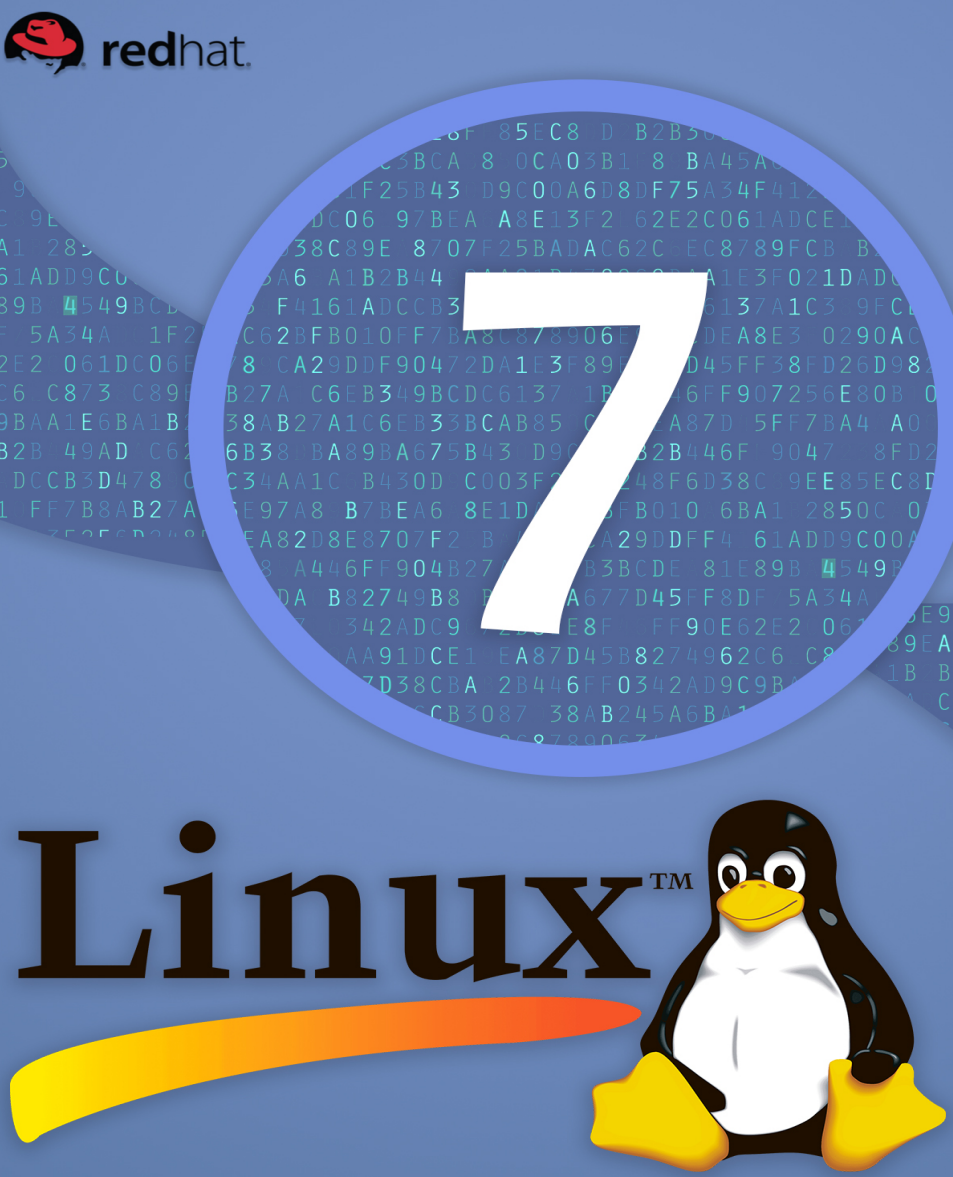
Low-Level Programming on Linux (x86-64)

Linux Kernel Module Development

redhat

debian

7



Prepared by Ayman Alheraki

https://simplifycpp.org

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

Linux Kernel Module Development

Prepared by Ayman Alheraki

simplifycpp.org

November 2025

Contents

Author's Introduction

Modern software systems are built atop layers of abstraction so deep that it is now possible to create complex applications without ever interacting directly with the processor, the memory subsystem, or the kernel that ultimately governs execution. Yet these underlying layers— CPU architecture, instruction execution, memory ordering, system calls, linkers, loaders, and kernel runtime mechanisms—are precisely where performance, correctness, reliability, and security are defined.

This series, Low-Level Programming on Linux (x86-64), is founded on the conviction that genuine mastery of systems programming requires a clear and unambiguous understanding of those foundations. As hardware grows more complex and the Linux kernel continues to evolve rapidly, the only stable path to competence is direct engagement with the architecture itself: the processor's execution model, the kernel's subsystems, the binary formats and runtimes that bind them together, and the calling conventions and ABI contracts that govern interaction between C, C++, and assembly. Everything presented across these volumes is grounded in verified, post-2020 Linux behavior rather than historical assumptions or obsolete conventions.

The series begins by establishing the x86-64 architecture as it is actually used by Linux: register files, addressing modes, segmentation remnants, page tables, interrupts, and the modern System V AMD64 calling convention. Assembly language is introduced from a strictly Linux-centric perspective, with every example written in Intel syntax and aligned with the expectations of contemporary compilers and the kernel itself. As

the volumes progress, each layer of the Linux execution environment is examined in its concrete form: GNU assembler mechanics and relocations, system calls, process creation, virtual memory, allocators, the ELF binary format, kernel modules, low-level cryptographic primitives, and finally minimal runtime construction without reliance on the standard C library.

Throughout the series, the emphasis is consistently placed on precision, correctness, and architectural clarity. Every assembly fragment adheres to the conventions enforced by modern toolchains and the Linux kernel. Every memory operation respects the Linux memory model. Every subsystem is described not as a conceptual abstraction, but in terms of its concrete data structures, execution contexts, and low-level rules. No mechanism is introduced without being anchored in the architectural or binary reality that makes it function.

This approach is intentional. Low-level programming cannot be learned through surface explanations or informal descriptions. It demands a mental model aligned with how interrupts are dispatched, how the scheduler manages preemption, how loaders resolve relocations, how virtual memory is constructed, how the ELF dynamic linker arranges segments, and how the stack behaves under ABI constraints. It requires the ability to read assembly with the same fluency as high-level code. It requires awareness of race conditions at the instruction level and an understanding of the cost and consequence of each operation, down to the microarchitectural effects of a single memory barrier.

These booklets are not intended to replace official documentation, nor to serve as simplified introductions. Instead, they provide a structured and technically disciplined path that brings the reader into direct contact with every layer on which modern Linux systems depend. Each topic connects naturally to the next: assembly to ELF, ELF to loaders, loaders to the kernel, the kernel to modules, modules to interrupts, interrupts to memory management, and memory management to runtime construction. This continuity is the defining characteristic of low-level engineering.

To ensure practical relevance, all examples are tested on contemporary distributions such as Debian 13 and RHEL 10 using current GNU assembler and toolchain revisions. Code is written without dependence on user-space libraries, emphasizing syscall-driven execution, standalone assembly routines, strict stack alignment, and exact register semantics. As a result, these volumes function not only as conceptual references but as executable blueprints for building reliable, architecture-aware systems on Linux.
Ultimately, this series is written for the programmer who refuses to treat the system as a black box—for the engineer who seeks to understand how a process begins executing, how an interrupt is routed, how an ELF binary becomes a running program, how the kernel allocates memory, how instructions flow through pipelines and caches, how a module integrates into kernel space, and how a runtime can be constructed from nothing more than raw system calls and assembly.
The path is demanding. Yet mastery of low-level programming forms the foundation upon which the most advanced systems—compilers, kernels, hypervisors, interpreters, real-time platforms, and high-performance engines—are built. Through these volumes, the reader acquires not only knowledge of Linux and x86-64 internals, but also the discipline, architectural awareness, and technical confidence required to engineer systems at the lowest levels with precision and intent.
This series is an invitation to see Linux not merely as an operating system to rely upon, but as an execution environment to understand, to analyze, and to deliberately control.

Stay Connected For more discussions and valuable content about Linux Kernel Module Development, 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

Kernel Development Model

1.1 Kernel Space Access Rules

Kernel development on Linux requires a precise understanding of the execution environment that governs memory visibility, privilege boundaries, and access permissions. All kernel code executes in a privileged processor mode and therefore carries the responsibility of enforcing strict rules to protect system integrity. Violating these rules often leads to immediate faults, silent data corruption, or a compromised machine. This section outlines the modern (post-2020) access constraints the x86-64 Linux kernel imposes and demonstrates their operation with minimal Intel-syntax GNU assembler examples.

1.1.1 Execution Privilege Levels

The x86-64 architecture provides four privilege rings, but Linux uses only two:

- Ring 0: Kernel

- Ring 3: User space

Transitions between privilege levels occur strictly through kernel- controlled entry points such as system calls, interrupts, traps, and exceptions.
A kernel module executes at CPL=0, which allows privileged instructions such as lgdt, wrmsr, or hlt. However, Linux policy and execution context prohibit most direct use of these instructions outside tightly controlled kernel subsystems.
Minimal example (illegal in most kernel contexts):

```
.intel_syntax noprefix
.globl test_priv

test_priv:
    hlt             # Halts the processor (fatal outside idle loop)
    ret
```

Executing this instruction in a module causes a system hang. Kernel access rules exist precisely to prevent such failures.

1.1.2 Kernel Address Space Visibility

On x86-64, kernel and user space share a single virtual address map, with the kernel occupying the upper canonical half. While Ring 0 code can technically dereference user-space addresses, Linux explicitly forbids direct access to user memory.
Linux enforces pointer safety using copy_from_user, copy_to_user, and controlled accessors. Modern kernels also enable hardware enforcement through SMEP and SMAP:

- SMAP prevents kernel-mode data access to user pages
- SMEP prevents kernel-mode execution of user-space code

Unsafe user pointer dereference:

```
.intel_syntax noprefix
.globl unsafe_read

unsafe_read:
    mov rax, rdi          # rdi contains a user pointer
    mov rbx, [rax]        # Faults when SMAP is enabled
    ret
```

This produces a page fault or general protection exception.
Controlled SMAP window (illustrative only):

```
.intel_syntax noprefix
.globl controlled_load

controlled_load:
    stac                  # Enable supervisor access to user memory
    mov rax, [rdi]        # Valid only after pointer validation
    clac                  # Restore SMAP protection
    ret
```

Modern kernels restrict such sequences to audited core code only.

1.1.3 Kernel Memory Safety Boundaries

Kernel space has no internal memory isolation. Any incorrect pointer dereference can corrupt unrelated kernel structures.
Rules enforced by kernel engineering practice:

- Never trust external pointers
- Never perform unchecked pointer arithmetic

- Never use freed kernel memory
- Never dereference physical addresses directly

Dangerous pointer arithmetic:

```
.intel_syntax noprefix
.globl write_offset

write_offset:
    mov rax, rdi            # Kernel pointer
    add rax, rsi            # Unchecked offset
    mov qword ptr [rax], rdx
    ret
```

If the offset exceeds bounds, unrelated kernel state is overwritten.

1.1.4 Access to Model-Specific Registers (MSRs)

Ring 0 allows MSR access, but Linux strictly regulates it.

- Only documented MSRs may be accessed
- Many MSRs are vendor-specific
- Incorrect writes may cause permanent lockups

Safe MSR read (Time Stamp Counter):

```
.intel_syntax noprefix
.globl read_tsc

read_tsc:
    mov ecx, 0x10           # MSR_TSC
```

```
    rdmsr
    shl rdx, 32
    or  rax, rdx
    ret
```

Modules must never modify power, frequency, or APIC MSRs directly.

1.1.5 Kernel Stack Access Rules

The kernel stack is:

- Per-CPU
- Fixed-size (commonly 16 KB)
- Non-growable
- Unsuitable for large allocations

Unsafe stack usage:

```
.intel_syntax noprefix
.globl bad_stack

bad_stack:
    sub rsp, 8192           # Consumes half the kernel stack
    mov qword ptr [rsp], rax
    add rsp, 8192
    ret
```

Kernel stacks must remain shallow and predictable.

1.1.6 Access Rules for Control Registers

Control registers are managed exclusively by architecture-specific kernel code.

Forbidden operation:

```
.intel_syntax noprefix
    mov cr0, rax            # Never allowed in modules
```

Allowed read of CR2:

```
.intel_syntax noprefix
.globl read_cr2

read_cr2:
    mov rax, cr2
    ret
```

1.1.7 Memory Barriers and Ordering Rules

Despite x86-64 TSO guarantees, the kernel requires explicit ordering in many contexts.

- mfence
- lfence
- sfence
- Locked atomic operations

Example write barrier:

```
.intel_syntax noprefix
.globl write_barrier

write_barrier:
    mfence
    ret
```

1.1.8 Device-Memory Access Constraints

MMIO access requires mapping via ioremap and strict ordering.
Unsafe MMIO read:

```
.intel_syntax noprefix
    mov rax, [rdi]          # Violates device ordering
```

Correct ordered access:

```
.intel_syntax noprefix
.globl read_mmio64

read_mmio64:
    mov rax, qword ptr [rdi]
    lfence
    ret
```

1.1.9 Conclusion

Kernel access rules exist to preserve correctness in an environment where a single instruction can compromise the entire system. Hardware features such as SMEP, SMAP, and architectural barriers assist, but the primary responsibility remains with the kernel developer.

1.2 GPL Licensing Notes

Linux kernel modules are governed by strict GPL-based licensing rules. Although modules are compiled separately, they execute as privileged kernel components and are subject to the same architectural and legal constraints.

1.2.1 Exported Symbols and GPL Enforcement

Linux exposes symbols via:

- EXPORT_SYMBOL()
- EXPORT_SYMBOL_GPL()

Example GPL-only symbol reference:

```
.intel_syntax noprefix
.globl my_call
.extern gpl_only_function

my_call:
    call gpl_only_function
    ret
```

Non-GPL modules attempting this are rejected.

1.2.2 Module Metadata and Licensing Declarations

Assembly-level GPL declaration:

```
.intel_syntax noprefix
.section .modinfo
.asciz "license=GPL"
```

Without this, the module is treated as proprietary and taints the kernel.

1.2.3 Kernel Tainting

Non-GPL modules taint the kernel:

```
.intel_syntax noprefix
.section .modinfo
.asciz "license=Proprietary"

.text
.globl module_entry
module_entry:
    xor rax, rax
    ret
```

1.2.4 Inline Assembly and Licensing

Licensing depends on behavior and linkage, not source language. Pure assembly modules are not exempt from GPL requirements.

1.2.5 Practical Summary

A kernel module is an extension of the kernel itself. Because it runs in Ring 0 and interacts with internal subsystems, GPL compliance is both a legal and architectural requirement.

Chapter 2

Module Structure

2.1 module_init and module_exit

The entry and exit points of a Linux kernel module define the module's lifecycle within the kernel image. They determine when the module becomes operational, when its subsystems are registered, and how it safely disengages from the kernel environment. In the Linux kernel development model, these two functions form the structural skeleton that all modules must follow, regardless of whether they are implemented in C or in low-level assembly.
This section explains their semantics, execution environment, and constraints, and presents minimal Intel-syntax GNU assembler equivalents suitable for low-level module development.

2.1.1 The Role of the Initialization Function

module_init associates a function with the module load event.
When the module is inserted using insmod or modprobe, the kernel invokes the

initialization routine in process context with full privileges and strict expectations:

- It must complete quickly
- It must not block indefinitely
- It must not leak resources
- It must return 0 on success or a negative error code on failure
- On failure, all previously acquired resources must be released manually

Canonical C form:

```
static int ___init mod_start(void)
{
    return 0;
}
module_init(mod_start);
```

At the binary level, the kernel records the initialization function pointer in an .initcall ELF section.

2.1.2 The Exit Function and Graceful Teardown

module_exit registers a cleanup routine executed when the module is removed using rmmod.
The exit routine must:

- Unregister all callbacks
- Release all allocated memory
- Cancel timers and workqueues

- Dismantle device or filesystem entries
- Guarantee no references remain to module code or data

Failure to unwind correctly leads to use-after-free errors or immediate kernel faults.
Canonical C form:

```
static void __exit mod_end(void)
{
    /* cleanup */
}
module_exit(mod_end);
```

If a module is marked permanent, the exit handler is omitted and unload is disallowed.

2.1.3 Binary-Level Structure of Init and Exit Sections

At the ELF level, the kernel recognizes:

- .init.text
- .exit.text

Depending on kernel configuration, .init.text may be discarded immediately after module initialization completes.
Assembly developers must place code in the correct sections so the kernel loader can correctly interpret its lifecycle role.

2.1.4 Implementing Init and Exit Functions in Assembly

Assembly-based modules must declare init and exit routines explicitly and place them in recognized sections.
Initialization Function

```
.intel_syntax noprefix

.section .init.text
.globl mod_start
mod_start:
    xor eax, eax          # return 0 (success)
    ret
```

The return value follows the C ABI: a negative value aborts loading.
Exit Function

```
.intel_syntax noprefix

.section .exit.text
.globl mod_end
mod_end:
    ret
```

2.1.5 Connecting Init and Exit Functions to Kernel Metadata

The module_init and module_exit macros expand to ELF entries placed in .initcall and .exitcall sections.
Assembly equivalent:

```
.intel_syntax noprefix

.section .initcall6.init
.quad mod_start

.section .exitcall.exit
.quad mod_end
```

Initcall level 6 corresponds to the default module initialization stage.

2.1.6 Execution Context of Init and Exit Functions

Initialization Context

- Executes in process context
- May sleep
- May allocate memory with GFP_KERNEL
- Must ensure synchronization for shared state
- Must return promptly

Exit Context

- Executes in process context
- Must reverse all initialization actions
- Must ensure no CPU is executing module code
- Must prevent races with interrupts and worker threads

2.1.7 Failure Handling in mod_start

If initialization fails:

- All allocated resources must be released
- A negative errno value must be returned
- The kernel aborts loading and never calls mod_end

Example:

```
.intel_syntax noprefix
.globl mod_start

mod_start:
    mov eax, -22          # -EINVAL
    ret
```

2.1.8 Minimal Complete Assembly Module Skeleton

```
.intel_syntax noprefix

.section .modinfo
.asciz "license=GPL"

.section .init.text
.globl mod_start
mod_start:
    xor eax, eax
    ret

.section .exit.text
.globl mod_end
mod_end:
    ret

.section .initcall6.init
.quad mod_start

.section .exitcall.exit
.quad mod_end
```

2.1.9 Essential Safety Principles for Init and Exit Routines

1. Initialization must be bounded and predictable
2. Exit routines must fully dismantle initialized resources
3. Init code must not rely on exit code unless loading succeeds
4. Assembly must obey the System V AMD64 ABI
5. Stack usage must remain minimal and deterministic

2.2 Logging with printk

Kernel logging is performed through printk(), which writes to the kernel log ring buffer and operates under strict execution and ordering constraints.

2.2.1 Calling printk from Assembly

```
.intel_syntax noprefix
.extern printk

.section .text
.globl log_message
log_message:
    lea rdi, [rip + msg]
    xor eax, eax          # no vector arguments
    call printk
    ret

.section .rodata
msg:
    .asciz "Kernel module says hello.\n"
```

2.2.2 Logging Dynamic Values

```
.intel_syntax noprefix
.extern printk

.section .text
.globl log_reg
log_reg:
    lea rdi, [rip + fmt]
    mov rsi, rax
    xor eax, eax
    call printk
    ret

.section .rodata
fmt:
    .asciz "Value in RAX = %llx\n"
```

2.2.3 Complete Assembly Module Using printk

```
.intel_syntax noprefix

.section .modinfo
.asciz "license=GPL"

.extern printk

.section .init.text
.globl mod_start
mod_start:
    lea rdi, [rip + init_msg]
    xor eax, eax
    call printk
```

```
    xor eax, eax
    ret

.section .exit.text
.globl mod_end
mod_end:
    lea rdi, [rip + exit_msg]
    xor eax, eax
    call printk
    ret

.section .initcall6.init
.quad mod_start

.section .exitcall.exit
.quad mod_end

.section .rodata
init_msg:
    .asciz "Module loaded.\n"

exit_msg:
    .asciz "Module unloaded.\n"
```

2.3 Symbol Exporting

Symbol exporting extends the kernel's global symbol table and is enforced at module load time.

2.3.1 Exporting Symbols in Assembly

```
.intel_syntax noprefix
```

```
.section .text
.globl add64
add64:
    add rdi, rsi
    mov rax, rdi
    ret

.section .export_symbol
.asciz "add64"
```

2.3.2 Exporting GPL-Only Symbols

```
.intel_syntax noprefix

.section .text
.globl fast_mul
fast_mul:
    imul rdi, rsi
    mov rax, rdi
    ret

.section .export_symbol_gpl
.asciz "fast_mul"
```

2.3.3 Exporting Data Objects

```
.intel_syntax noprefix

.section .data
.globl kernel_counter
kernel_counter:
    .quad 0
```

```
.section .export_symbol
.asciz "kernel_counter"
```

2.3.4 Complete Module with Exported Symbol

```
.intel_syntax noprefix

.section .modinfo
.asciz "license=GPL"

.section .text
.globl k_add
k_add:
    add rdi, rsi
    mov rax, rdi
    ret

.section .export_symbol
.asciz "k_add"

.section .init.text
.globl mod_start
mod_start:
    xor eax, eax
    ret

.section .exit.text
.globl mod_end
mod_end:
    ret

.section .initcall6.init
```

```
.quad mod_start

.section .exitcall.exit
.quad mod_end
```

2.3.5 Conclusion

Module structure, logging, and symbol exporting define the contractual boundary between a kernel module and the rest of the system. Correct section placement, ABI compliance, disciplined logging, and strict export rules are not optional conveniences—they are architectural requirements. In kernel space, structural correctness is a prerequisite for stability.

Chapter 3

Memory Allocation in Kernel

3.1 kmalloc and kfree

Dynamic memory allocation in the Linux kernel differs fundamentally from user-space allocation. Kernel allocators operate under strict latency constraints, limited stack and heap space, and explicit execution context rules. Memory must often be physically contiguous, and allocations may be forbidden from sleeping depending on context. The primary interface for general-purpose dynamic memory in kernel modules is the slab/SLUB allocator, exposed through kmalloc and kfree.

3.1.1 Architectural Foundation of kmalloc

kmalloc allocates physically contiguous memory from kernel-managed slab caches. Key differences from user-space allocation:

- Memory is allocated from SLUB caches
- Pages are permanently resident and unswappable

- Allocation behavior depends on GFP flags
- Zeroing is optional (___GFP_ZERO)
- Allocation failure must be handled explicitly

Internally, large allocations may fall back to the page allocator.

3.1.2 GFP Flags and Allocation Context

GFP flags describe how memory may be obtained:

- GFP_KERNEL — sleepable, reclaim allowed
- GFP_ATOMIC — non-sleeping, interrupt-safe
- GFP_NOWAIT — no reclaim, immediate failure
- ___GFP_ZERO — zero-initialize memory

Using sleepable flags in atomic context is a fatal error.

3.1.3 Using kmalloc and kfree in C

```
char *buf = kmalloc(128, GFP_KERNEL);
if (!buf)
    return -ENOMEM;

kfree(buf);
```

3.1.4 Calling kmalloc and kfree from Assembly

Kernel allocation functions follow the System V AMD64 ABI.

Allocation

```
.intel_syntax noprefix
.extern kmalloc

.section .text
.globl alloc_block
alloc_block:
    mov rdi, 256        # size
    mov rsi, 0x20       # GFP_KERNEL
    call kmalloc
    ret
```

On failure, rax = 0.

Freeing

```
.intel_syntax noprefix
.extern kfree

.section .text
.globl free_block
free_block:
    mov rdi, rax        # pointer to free
    call kfree
    ret
```

Only memory returned by kmalloc may be passed to kfree.

3.1.5 Correct Allocation Context

Sleepable Context

- Process context
- Reclaim allowed
- Use GFP_KERNEL

Atomic Context

- Interrupts, spinlocks, preemption disabled
- No sleeping
- Use GFP_ATOMIC

Unsafe example:

```
mov rdi, 64
mov rsi, 0x20      # GFP_KERNEL
call kmalloc       # illegal in interrupt context
```

Correct atomic allocation:

```
mov rdi, 64
mov rsi, 0x40      # GFP_ATOMIC
call kmalloc
```

3.1.6 Memory Alignment Guarantees

kmalloc guarantees:

- Natural alignment for all kernel types
- Physical contiguity
- Compatibility with most DMA (subject to device constraints)

Requested sizes are rounded to slab cache buckets (32, 64, 128, 256…).

3.1.7 Zeroed Allocation

```
.intel_syntax noprefix
.extern kmalloc

.section .text
.globl alloc_zero
alloc_zero:
    mov rdi, 128
    mov rsi, 0x21          # GFP_KERNEL | __GFP_ZERO
    call kmalloc
    ret
```

3.1.8 Failure Handling

kmalloc may fail under any condition.

```
test rax, rax
jz .fail
# safe to use memory
```

All prior allocations must be unwound manually.

3.1.9 Rules for kfree

1. Pointer must originate from kmalloc or related allocators
2. Double-free is forbidden
3. Memory must not be in use by another CPU
4. Freeing NULL is allowed
5. Memory must not outlive the module

Correct pattern:

```
test rdi, rdi
jz .skip
call kfree
.skip:
```

3.1.10 Complete Allocation Example

```
.intel_syntax noprefix
.extern kmalloc
.extern kfree

.section .text
.globl alloc_use_free
alloc_use_free:
    mov rdi, 64
    mov rsi, 0x20
    call kmalloc
    test rax, rax
    jz .fail

    mov qword ptr [rax], 0x1122334455667788

    mov rdi, rax
    call kfree

    xor eax, eax
    ret

.fail:
    mov eax, -12          # -ENOMEM
    ret
```

3.1.11 Safety Principles for kmalloc/kfree

1. Match GFP flags to execution context
2. Always check for failure
3. Avoid high-order allocations
4. Free exactly once
5. Never free memory used by other CPUs
6. Zero memory only when required
7. Treat kernel memory as scarce

3.2 GFP Allocation Flags

GFP flags encode the kernel's allocation policy and context constraints.

3.2.1 Core GFP Classes

GFP_KERNEL

```
mov rsi, 0x20
call kmalloc
```

GFP_ATOMIC

```
mov rsi, 0x40
call kmalloc
```

___GFP_ZERO

```
mov rsi, 0x21
call kmalloc
```

3.2.2 Context-Aware Allocation Example

```
.intel_syntax noprefix
.extern kmalloc

.section .text
.globl choose_alloc
choose_alloc:
    # rdi = size
    # rsi = 0 sleepable, 1 atomic

    test rsi, rsi
    jnz .atomic

.sleepable:
    mov rsi, 0x20
    call kmalloc
    ret

.atomic:
    mov rsi, 0x40
    call kmalloc
    ret
```

3.3 Slab Cache Basics

SLUB is the dominant allocator in modern kernels, optimized for scalability and cache locality.

3.3.1 Why Slab Caches Exist

- Fixed-size objects

- Deterministic allocation
- Per-CPU fast paths
- Minimal fragmentation
- Cache-line alignment

3.3.2 Creating a Slab Cache (C)

```
struct kmem_cache *cache;

cache = kmem_cache_create("my_cache",
                          sizeof(struct obj),
                          0, 0, NULL);
```

3.3.3 Allocating and Freeing from a Cache

```
ptr = kmem_cache_alloc(cache, GFP_KERNEL);
kmem_cache_free(cache, ptr);
```

3.3.4 Slab Allocation in Assembly

Allocate

```
.intel_syntax noprefix
.extern kmem_cache_alloc

.section .text
.globl cache_alloc
cache_alloc:
    mov rdi, rbx          # kmem_cache *
    mov rsi, 0x20
```

```
    call kmem_cache_alloc
    ret
```

Free

```
.intel_syntax noprefix
.extern kmem_cache_free

.section .text
.globl cache_free
cache_free:
    mov rdi, rbx          # kmem_cache *
    mov rsi, rax          # object
    call kmem_cache_free
    ret
```

3.3.5 Complete Slab Cache Module Example

```
.intel_syntax noprefix

.extern kmem_cache_create
.extern kmem_cache_destroy
.extern printk

.section .bss
.align 8
cache_ptr:
    .quad 0

.section .rodata
cache_name:
    .asciz "my_cache"
msg_init:
```

```
    .asciz "Cache created\n"
msg_exit:
    .asciz "Cache destroyed\n"

.section .init.text
.globl mod_start
mod_start:
    lea rdi, [rip + cache_name]
    mov rsi, 32
    xor rdx, rdx
    xor rcx, rcx
    xor r8,  r8
    call kmem_cache_create

    mov [rip + cache_ptr], rax

    lea rdi, [rip + msg_init]
    xor eax, eax
    call printk

    xor eax, eax
    ret

.section .exit.text
.globl mod_end
mod_end:
    mov rdi, [rip + cache_ptr]
    call kmem_cache_destroy

    lea rdi, [rip + msg_exit]
    xor eax, eax
    call printk
    ret
```

```
.section .initcall6.init
.quad mod_start

.section .exitcall.exit
.quad mod_end
```

3.3.6 Safety Principles for Slab Usage

1. Create caches only during init
2. Destroy caches only after freeing all objects
3. Never free into a destroyed cache
4. Prefer slab caches for frequent allocations
5. Do not mix slab and kmalloc memory
6. Respect GFP constraints

3.3.7 Conclusion

Kernel memory allocation is governed by strict architectural and contextual rules. Correct usage of kmalloc, GFP flags, and slab caches is mandatory for system stability. Misuse results not in graceful failure, but in deadlocks, corruption, or immediate system failure.

Chapter 4

/proc and /sys Interfaces

4.1 Creating a /proc Entry

The /proc filesystem provides a dynamic, in-memory interface through which kernel components expose internal state to user space. Unlike traditional filesystems, /proc entries do not represent data stored on disk; instead, they are runtime views into kernel structures. For kernel modules, /proc represents a controlled mechanism for exporting debugging information, module status, or runtime statistics while maintaining strict synchronization and safety guarantees.
This section explains the creation, registration, and removal of /proc entries in modern Linux kernels (post-5.10), including low-level considerations, concurrency behavior, and integration with Intel-syntax GNU assembly.

4.1.1 The /proc Filesystem Model

/proc exposes kernel information via virtual files that:

- Do not implement full POSIX semantics

- Are backed by the procfs infrastructure
- Invoke callbacks on read and write
- Have no persistent storage
- Require strict validation and locking

Kernel modules create entries through the procfs API.

4.1.2 Modern Procfs API

Modern kernels use:

- proc_create()
- proc_create_data()
- proc_remove()

Each entry is backed by a proc_ops structure and optional private data.

4.1.3 Minimal C Glue (Mandatory)

Structural glue must be written in C.

```
static int myproc_show(struct seq_file *m, void *v)
{
    seq_printf(m, "Hello from kernel module.\n");
    return 0;
}

static int myproc_open(struct inode *inode, struct file *file)
{
```

```
    return single_open(file, myproc_show, NULL);
}

static const struct proc_ops myproc_ops = {
    .proc_open    = myproc_open,
    .proc_read    = seq_read,
    .proc_lseek   = seq_lseek,
    .proc_release = single_release,
};
```

4.1.4 Assembly Integration for /proc Callbacks

Assembly is ideal for low-level data retrieval.

Example: Read CR2 in Assembly

```
.intel_syntax noprefix

.section .text
.globl read_cr2
read_cr2:
    mov rax, cr2        # faulting linear address
    ret
```

Used from C:

```
extern unsigned long read_cr2(void);

static int myproc_show(struct seq_file *m, void *v)
{
    seq_printf(m, "cr2 = 0x%lx\n", read_cr2());
    return 0;
}
```

4.1.5 Read Path Execution Model

When user space reads /proc/myproc:

1. .proc_open executes
2. single_open() binds a seq_file
3. seq_read() streams output
4. show() generates content
5. Data is copied to user space

The seq_file layer guarantees safe pagination and concurrency.

4.1.6 Assembly Example: Fast Hardware Read

Reading the TSC

```
.intel_syntax noprefix

.section .text
.globl read_tsc64
read_tsc64:
    rdtsc
    shl rdx, 32
    or  rax, rdx
    ret
```

Used in C:

```
extern u64 read_tsc64(void);

static int myproc_show(struct seq_file *m, void *v)
{
    seq_printf(m, "TSC = %llu\n", read_tsc64());
    return 0;
}
```

4.1.7 Safety Rules for /proc

1. Validate all user input
2. Keep callbacks short and deterministic
3. Avoid allocation in read paths
4. Never sleep inside seq_file iteration
5. Always remove entries on module exit
6. Use assembly only for fast, safe operations

4.2 Using seq_file (Advanced)

4.2.1 Iterative seq_file Model

The full iterator pipeline:

1. start()
2. show()
3. next()

4. stop()

This model supports large, ordered datasets efficiently.

4.2.2 Assembly Helper Used by Iterators

```
.intel_syntax noprefix

.section .text
.globl read_tsc64
read_tsc64:
    rdtsc
    shl rdx, 32
    or  rax, rdx
    ret
```

4.2.3 Advanced Assembly Integration

Reading CR3

```
.intel_syntax noprefix

.section .text
.globl read_cr3
read_cr3:
    mov rax, cr3
    ret
```

Used safely from seq_file callbacks.

4.2.4 Safety Principles for seq_file

1. Never expose kernel pointers

2. Keep callbacks constant-time
3. Avoid sleeping locks
4. Assembly helpers must be deterministic
5. Always complete iteration

4.3 sysfs Attributes

sysfs exposes kernel objects and attributes in a structured, object-oriented manner. Unlike /proc, sysfs represents persistent kernel objects and configuration state.

4.3.1 sysfs Architecture

sysfs is built on:

- kobject
- kset
- attribute

All sysfs entries are text-based and strictly validated.

4.3.2 Using Assembly in sysfs Attributes

Assembly is suitable for reading hardware registers.

```
.intel_syntax noprefix

.section .text
.globl read_cr2
```

```
read_cr2:
    mov rax, cr2
    ret
```

Used from C:

```
extern unsigned long read_cr2(void);

static ssize_t cr2_show(struct kobject *kobj,
                        struct kobj_attribute *attr,
                        char *buf)
{
    return scnprintf(buf, PAGE_SIZE,
                     "cr2 = 0x%lx\n", read_cr2());
}
```

4.3.3 sysfs Safety Rules

1. Always validate user input
2. Never expose raw pointers
3. Keep operations short
4. Use locking for shared state
5. Ensure full cleanup on exit

4.3.4 Conclusion

The /proc and /sys interfaces provide complementary mechanisms for exposing kernel state. /proc is optimized for diagnostics and runtime inspection, while sysfs represents structured kernel objects and configuration. By combining disciplined C interfaces with

small, deterministic Intel-syntax assembly helpers, kernel modules can expose hardware-level data safely, efficiently, and in full compliance with modern kernel constraints.

Chapter 5

DebugFS

5.1 Creating DebugFS Nodes

debugfs is a specialized virtual filesystem designed exclusively for kernel debugging and instrumentation. Unlike /proc and /sys, which enforce strict structural and semantic contracts, debugfs offers a flexible, developer-oriented interface with minimal policy constraints. It is intended for diagnostics, tracing, performance testing, and rapid prototyping during kernel development, not for production interfaces.
Originally introduced in Linux 2.6 and significantly refined in modern kernels (post-5.10), debugfs provides:

- Arbitrary hierarchical file creation
- Direct read and write callbacks
- Binary and text data exposure
- Access to low-level hardware state

- A low-friction environment without ABI guarantees

Because debugfs bypasses normal kernel stability rules, all safety responsibility lies with the module developer.

5.1.1 Architectural Purpose of DebugFS

The core purpose of debugfs is:
Expose kernel state rapidly for developers without creating stable interfaces or affecting kernel ABI guarantees.
Key properties:

- Typically mounted at /sys/kernel/debug
- Fully virtual and ephemeral
- No enforced object model
- No restriction to text-only data
- Writable nodes permitted

Unlike sysfs, DebugFS:

- Has no kobject hierarchy
- Imposes no lifetime rules
- Allows binary output
- Is not suitable for distribution

This flexibility makes DebugFS powerful—but inherently dangerous.

5.1.2 Creating a Basic DebugFS File

A basic DebugFS file is created using:

```
debugfs_create_file("myfile", 0644, parent, data, &fops);
```

Where:

- "myfile" is the file name
- Permissions use standard UNIX modes
- parent is a directory dentry or NULL
- data is private per-file data
- fops defines file behavior

5.1.3 Defining File Operations

Most DebugFS nodes implement .read and optionally .write.
Minimal Read Handler

```
static ssize_t dbg_read(struct file *filp,
                        char __user *buf,
                        size_t len,
                        loff_t *ppos)
{
    return simple_read_from_buffer(buf, len, ppos,
                                   "debug\n", 6);
}
```

Optional Write Handler

```
static ssize_t dbg_write(struct file *filp,
                         const char __user *buf,
                         size_t len,
                         loff_t *ppos)
{
    /* validate input */
    return len;
}
```

File Operations Structure

```
static const struct file_operations dbg_fops = {
    .owner = THIS_MODULE,
    .read  = dbg_read,
    .write = dbg_write,
};
```

5.1.4 Creating Directories and Files

```
struct dentry *dir, *file;

dir  = debugfs_create_dir("mydir", NULL);
file = debugfs_create_file("regs", 0444, dir, NULL, &dbg_fops);
```

This produces:

```
/sys/kernel/debug/mydir/regs
```

5.1.5 Integrating Low-Level Assembly

DebugFS glue must be written in C, but low-level data acquisition is ideal for assembly.
Assembly: Read CR3

```
.intel_syntax noprefix

.section .text
.globl read_cr3
read_cr3:
    mov rax, cr3
    ret
```

Using CR3 in a DebugFS Read

```
extern unsigned long read_cr3(void);

static ssize_t dbg_read(struct file *filp,
                        char __user *buf,
                        size_t len,
                        loff_t *ppos)
{
    char tmp[64];
    int n = scnprintf(tmp, sizeof(tmp),
                      "cr3 = 0x%lx\n", read_cr3());

    return simple_read_from_buffer(buf, len, ppos,
                                   tmp, n);
}
```

5.1.6 Binary Output Through DebugFS

Unlike sysfs or procfs, DebugFS supports raw binary output.

```
static ssize_t dbg_read_bin(struct file *filp,
                            char __user *buf,
                            size_t len,
                            loff_t *ppos)
{
```

```
    u64 val = read_cr3();
    return simple_read_from_buffer(buf, len, ppos,
                                   &val, sizeof(val));
}
```

Binary DebugFS nodes are ideal for:

- Hardware counters
- Register snapshots
- Timing samples
- Internal state dumps

5.1.7 Module Initialization and Cleanup

Initialization

```
static struct dentry *root;

static int ___init mod_start(void)
{
    root = debugfs_create_dir("module_dbg", NULL);
    if (!root)
        return -ENOMEM;

    debugfs_create_file("info", 0444, root,
                        NULL, &dbg_fops);
    return 0;
}
```

Cleanup

```
static void __exit mod_end(void)
{
    debugfs_remove_recursive(root);
}
```

5.1.8 Advanced Example: Multi-Register Read

Assembly: TSC and CR2

```
.intel_syntax noprefix

.section .text
.globl read_tsc64
read_tsc64:
    rdtsc
    shl rdx, 32
    or  rax, rdx
    ret

.globl read_cr2
read_cr2:
    mov rax, cr2
    ret
```

DebugFS Read

```
extern u64 read_tsc64(void);
extern unsigned long read_cr2(void);

static ssize_t dbg_read(struct file *f,
                        char __user *buf,
                        size_t len,
                        loff_t *ppos)
{
```

```
    char tmp[128];
    int n = scnprintf(tmp, sizeof(tmp),
                      "TSC = %llu\nCR2 = 0x%lx\n",
                      read_tsc64(), read_cr2());

    return simple_read_from_buffer(buf, len, ppos,
                                   tmp, n);
}
```

5.1.9 Safety Principles for DebugFS

1. Never use DebugFS for production interfaces
2. Validate all writable input
3. Avoid exposing kernel pointers
4. Keep callbacks fast and deterministic
5. Never sleep in read/write paths unless explicitly safe
6. Assembly helpers must not block or allocate
7. Always remove DebugFS nodes on unload

5.1.10 Conclusion

DebugFS provides an unrestricted environment for exposing kernel module internals during development. Its flexibility allows low-level Intel-syntax assembly routines to surface real hardware state directly to user space. When used with discipline—atomic access, bounded output, and deterministic execution—DebugFS becomes an indispensable tool for kernel diagnostics, performance analysis, and runtime tracing on modern x86-64 Linux systems.

Chapter 6

Interrupt Handling (Intro)

6.1 IRQ Request and Release

Interrupts are the fundamental asynchronous signaling mechanism used by hardware devices to notify the kernel of external events. In modern Linux systems, interrupt handling follows a layered architecture: the CPU's local APIC receives interrupt vectors, the kernel's IRQ subsystem dispatches them, and device drivers register handlers using request_irq().
For kernel modules, correct IRQ acquisition and release is critical. Improper handling results in interrupt storms, undefined hardware behavior, and system instability.
This section introduces the modern IRQ request model, handler constraints, and safe teardown practices on x86-64 Linux systems.

6.1.1 Kernel IRQ Architecture Overview

On x86-64 systems:

- Hardware signals are routed through the I/O APIC or local APIC.

- APICs map IRQ lines to interrupt vectors (0–255).
- The kernel's IRQ core dispatches the interrupt to registered ISRs.
- Device drivers register handlers via request_irq().

The kernel serializes execution per IRQ line, but handlers must remain atomic, deterministic, and extremely fast.

6.1.2 Requesting an IRQ Line

The standard IRQ registration API is:

```
int request_irq(unsigned int irq,
                irq_handler_t handler,
                unsigned long flags,
                const char *name,
                void *dev_id);
```

Constraints

- Handler executes in hard IRQ context:
  - No sleeping
  - No blocking locks
  - No GFP_KERNEL allocation
  - No heavy computation
- dev_id must uniquely identify the handler on shared IRQs
- flags control trigger mode and sharing

Minimal IRQ Handler

```
static irqreturn_t my_irq_handler(int irq, void *dev_id)
{
    trace_record(irq);    /* atomic, lockless */
    return IRQ_HANDLED;
}
```

6.1.3 Using Intel-Syntax Assembly in an IRQ Handler

IRQ handlers must be declared in C, but may call tiny assembly helpers for constant-time register capture.

Assembly: Fast CPU State Snapshot

```
.intel_syntax noprefix

.section .text
.globl read_cpu_state
read_cpu_state:
    pushfq
    pop rax          # RFLAGS
    mov rdx, cr2     # faulting address
    ret
```

Usage in IRQ Handler

```
extern unsigned long read_cpu_state(unsigned long *cr2);

static irqreturn_t my_irq_handler(int irq, void *dev_id)
{
    unsigned long flags, cr2;
    flags = read_cpu_state(&cr2);

    this_cpu_write(last_irq_flags, flags);
    this_cpu_write(last_irq_cr2, cr2);
```

```
    return IRQ_HANDLED;
}
```

All operations must remain constant-time and non-sleeping.

6.1.4 Registering the IRQ in Module Initialization

```
static int irq_line = 19;

static int ___init mod_start(void)
{
    return request_irq(irq_line,
                       my_irq_handler,
                       IRQF_SHARED,
                       "my_irq_dev",
                       &irq_line);
}
```

Notes

- IRQF_SHARED allows multiple handlers per line
- dev_id must match during release

6.1.5 Releasing an IRQ Line

IRQ handlers must be explicitly released:

```
free_irq(irq_line, &irq_line);
```

Rules:

- irq and dev_id must match registration

- Release must occur before memory teardown
- Device interrupts must be disabled first

Failure causes use-after-free or orphaned handlers.

6.1.6 Ensuring IRQ Safety

Interrupt handlers must not:

- Sleep or block
- Allocate memory with GFP_KERNEL
- Traverse large shared structures
- Perform long loops
- Emit large logs or DebugFS output
- Call blocking I/O APIs

Interrupt handlers should:

- Capture minimal state
- Acknowledge hardware quickly
- Schedule deferred processing
- Use atomic or per-CPU data

6.1.7 Assembly-Based Bottom-Half Signaling

Assembly: Atomic Counter Increment

```
.intel_syntax noprefix

.section .text
.globl irq_atomic_inc
irq_atomic_inc:
    lock inc qword ptr [rdi]
    ret
```

Usage in ISR

```
extern void irq_atomic_inc(u64 *p);

static irqreturn_t my_irq_handler(int irq, void *dev)
{
    irq_atomic_inc(&irq_event_count);
    return IRQ_WAKE_THREAD;
}
```

6.1.8 Threaded IRQ Handlers

Modern kernels support threaded IRQs:

```
request_threaded_irq(irq_line,
                     my_irq_handler,
                     my_thread_fn,
                     0,
                     "my_irq_dev",
                     dev);
```

Benefits

- Hard IRQ remains minimal
- Thread handler may sleep
- Complex logic moved safely

Thread Function

```
static irqreturn_t my_thread_fn(int irq, void *dev)
{
    msleep(5);
    heavy_analysis();
    return IRQ_HANDLED;
}
```

6.1.9 Correct Teardown Order

1. Disable device interrupts
2. Ensure no further IRQs fire
3. Call free_irq()
4. Remove DebugFS / sysfs nodes
5. Free memory

6.1.10 Complete Minimal IRQ Module

Assembly: Read TSC

```
.intel_syntax noprefix

.section .text
```

```
.globl read_tsc64
read_tsc64:
    rdtsc
    shl rdx, 32
    or  rax, rdx
    ret
```

C Driver Skeleton

```
extern u64 read_tsc64(void);
static int irq_line = 11;

static irqreturn_t isr(int irq, void *dev)
{
    this_cpu_write(last_irq_tsc, read_tsc64());
    return IRQ_HANDLED;
}

static int __init mod_start(void)
{
    return request_irq(irq_line, isr, 0, "irq_demo", &irq_line);
}

static void __exit mod_end(void)
{
    free_irq(irq_line, &irq_line);
}
```

6.1.11 Conclusion

IRQ request and release form the foundation of kernel-level interrupt handling. Hard IRQ handlers must be minimal, atomic, and deterministic, while all heavy processing must be deferred. By combining Linux's IRQ subsystem with disciplined Intel-syntax

assembly helpers, kernel modules can capture precise CPU state with minimal latency. Correct teardown ensures stability and prevents catastrophic race conditions. This discipline is essential for robust low-level driver development on modern x86-64 Linux systems.

Chapter 7

User–Kernel Communication

7.1 IOCTL

ioctl (Input/Output Control) provides a bidirectional, command-driven communication mechanism between user space and kernel modules. Unlike standard read/write operations, which transfer raw byte streams, ioctl enables structured, command-specific operations for controlling device behavior, configuring parameters, or exchanging well-defined data structures.
Within Linux kernel development, ioctl remains a central interface for character devices, drivers, and specialized subsystems requiring precise control semantics. This section introduces the modern IOCTL interface, security and race-safety considerations, and safe integration of x86-64 assembly routines.

7.1.1 Architectural Role of IOCTL

ioctl acts as the command dispatch interface of a device. It is appropriate when:

- Byte-stream I/O is insufficient

- One-shot or control-oriented commands are required
- Structured data must cross the user–kernel boundary
- User tools must manipulate kernel-side state explicitly

Key properties:

- Commands identified by integer values
- Direction encoded (read, write, or both)
- Explicit size information
- Arbitrary data structures allowed

The kernel imposes no semantic meaning on commands; API correctness is entirely the responsibility of the module author.

7.1.2 Modern IOCTL Encoding

Modern kernels require IOCTL commands to be defined using encoding macros:

- _IO(type, nr) — no data transfer
- _IOR(type, nr, data_type) — kernel to user
- _IOW(type, nr, data_type) — user to kernel
- _IOWR(type, nr, data_type) — bidirectional

Example Definitions

```
#define DEV_IOC_MAGIC   'K'

#define DEV_GET_FLAG    _IOR(DEV_IOC_MAGIC, 0x01, int)
#define DEV_SET_FLAG    _IOW(DEV_IOC_MAGIC, 0x02, int)
#define DEV_EXCH_STATE  _IOWR(DEV_IOC_MAGIC, 0x03, struct dev_state)
```

These encodings preserve ABI stability and ensure correct interpretation by tracing and diagnostic tools.

7.1.3 IOCTL File Operation

IOCTL dispatch occurs through .unlocked_ioctl.

File Operations

```
static const struct file_operations fops = {
    .owner          = THIS_MODULE,
    .unlocked_ioctl = dev_ioctl,
};
```

Handler Skeleton

```
static long dev_ioctl(struct file *file,
                      unsigned int cmd,
                      unsigned long arg)
{
    switch (cmd) {

    case DEV_GET_FLAG:
        return put_user(dev_flag, (int __user *)arg);

    case DEV_SET_FLAG:
        if (get_user(dev_flag, (int __user *)arg))
            return -EFAULT;
```

```
        return 0;

    case DEV_EXCH_STATE: {
        struct dev_state kstate;

        if (copy_from_user(&kstate,
                           (void ___user *)arg,
                           sizeof(kstate)))
            return -EFAULT;

        kstate.seq = read_tsc64();

        if (copy_to_user((void ___user *)arg,
                         &kstate,
                         sizeof(kstate)))
            return -EFAULT;
        return 0;
    }

    default:
        return -ENOTTY;
    }
}
```

7.1.4 Assembly Integration in IOCTL Paths

IOCTL handlers execute in process context, making them suitable for short, deterministic assembly helpers.
Assembly routines must be:

- Constant-time
- Non-blocking

- Free of side effects

Assembly: Read TSC

```
.intel_syntax noprefix

.section .text
.globl read_tsc64
read_tsc64:
    rdtsc
    shl rdx, 32
    or  rax, rdx
    ret
```

Used directly inside IOCTL handlers to provide precise timing information.

7.1.5 Structured Data Transfer

Structures exchanged via IOCTL must be:

- Properly aligned
- Fully validated
- Size-checked
- Versioned if long-lived

Example Structure

```
struct dev_state {
    u64 seq;
    u32 mode;
    u32 status;
};
```

Safe transfer requires explicit copying and fault handling.

7.1.6 Race Safety in IOCTL

IOCTL executes in process context and may:

- Sleep
- Acquire mutexes
- Allocate memory

However, it may race with:

- IRQ handlers
- Workqueues
- Other IOCTL invocations

Recommended protections:

- Mutexes for configuration state
- Spinlocks for IRQ-shared data
- Atomic types for counters and flags
- Memory barriers for structured state

7.1.7 Secure IOCTL Design

Correct IOCTL design requires:

1. Strict command whitelisting
2. Validation of all user pointers

3. Explicit bounds checking
4. No kernel pointer leakage
5. No unvalidated hardware access
6. No control-flow decisions from user data

Many historical kernel vulnerabilities originate from unsafe IOCTL implementations.

7.1.8 Complete IOCTL Example

Assembly: Read CR2

```
.intel_syntax noprefix

.section .text
.globl read_cr2
read_cr2:
    mov rax, cr2
    ret
```

Kernel Handler

```
extern unsigned long read_cr2(void);

#define DEV_GET_CR2 _IOR(DEV_IOC_MAGIC, 0x10, unsigned long)

static long dev_ioctl(struct file *file,
                      unsigned int cmd,
                      unsigned long arg)
{
    if (cmd == DEV_GET_CR2) {
        unsigned long v = read_cr2();
```

```
        return put_user(v,
                        (unsigned long ___user *)arg);
    }
    return -ENOTTY;
}
```

7.1.9 Conclusion

IOCTL provides a command-level, synchronous communication channel between user space and kernel modules. When designed carefully—with strict validation, atomic state handling, and safe data copying—it enables powerful and secure device control. Assembly-based helpers may be used safely for precise CPU state or timing capture when kept deterministic and side-effect free.

7.2 Netlink

Netlink is Linux's asynchronous, message-oriented communication mechanism between kernel and user space. Unlike IOCTL, Netlink is socket based, packet oriented, and naturally supports broadcasts and multi-client communication.
It underpins major subsystems such as routing, wireless configuration, audit, SELinux, and modern monitoring tools.

7.2.1 Architectural Overview

Netlink sockets are:

- Message-based
- Connectionless
- Asynchronous

- Multicast-capable
- Multi-client friendly

Kernel modules act as protocol endpoints, exchanging discrete messages described by struct nlmsghdr.

7.2.2 Kernel Netlink Socket Creation

```
static struct sock *nl_sock;

static const struct netlink_kernel_cfg cfg = {
    .input = nl_recv_fn,
};

nl_sock = netlink_kernel_create(&init_net,
                                NETLINK_USERSOCK,
                                &cfg);
```

The input callback executes in softirq context and must not sleep or block.

7.2.3 Assembly in Netlink Paths

Assembly: Fast TSC Read

```
.intel_syntax noprefix

.section .text
.globl nl_read_tsc
nl_read_tsc:
    rdtsc
    shl rdx, 32
    or  rax, rdx
    ret
```

Used for constant-time event timestamping.

7.2.4 Sending Kernel Replies

```
static int send_reply(struct sk_buff *req,
                      u32 value)
{
    struct sk_buff *skb;
    struct nlmsghdr *nlh;

    skb = nlmsg_new(sizeof(value), GFP_KERNEL);
    if (!skb)
        return -ENOMEM;

    nlh = nlmsg_put(skb,
                    NETLINK_CB(req).portid,
                    0,
                    NLMSG_DONE,
                    sizeof(value),
                    0);

    *(u32 *)nlmsg_data(nlh) = value;

    return nlmsg_unicast(nl_sock,
                         skb,
                         NETLINK_CB(req).portid);
}
```

7.2.5 Conclusion

Netlink provides scalable, asynchronous, structured communication between kernel and user space. Combined with atomic state handling, careful synchronization, and

assembly-based fast paths, it forms the foundation of modern kernel control and monitoring interfaces.

7.3 Shared Memory Approaches

Shared memory offers the highest-performance user–kernel communication mechanism. After initial mapping, no system calls are required, enabling continuous, zero-copy data exchange.

7.3.1 Kernel Mapping Model

The kernel allocates memory and maps it into user space via mmap. Both sides access the same physical pages.

7.3.2 Example mmap Implementation

```
static void *shared_area;

static int dev_mmap(struct file *f,
                    struct vm_area_struct *vma)
{
    unsigned long pfn =
        virt_to_phys(shared_area) >> PAGE_SHIFT;

    return remap_pfn_range(vma,
                           vma->vm_start,
                           pfn,
                           PAGE_SIZE,
                           vma->vm_page_prot);
}
```

7.3.3 Synchronization Rules

Shared memory provides no implicit ordering. Correctness requires:

- Atomic variables
- Memory barriers
- Explicit producer/consumer protocols
- Careful structure design

7.3.4 Assembly-Accelerated Updates

Atomic Increment

```
.intel_syntax noprefix

.section .text
.globl shm_atomic_inc
shm_atomic_inc:
    lock inc qword ptr [rdi]
    ret
```

7.3.5 Conclusion

Shared memory enables the lowest-latency user–kernel communication. Combined with explicit synchronization, atomic operations, and assembly-based fast paths, it forms the backbone of high-performance, real-time kernel instrumentation and control systems on x86-64 Linux.

Chapter 8

Final Project

8.1 "Live System Information" Kernel Module

A live system-information kernel module acts as an embedded instrumentation layer within the running kernel. Unlike traditional debugging interfaces such as procfs, debugfs, or snapshot-based tracing, this module continuously captures real-time CPU and kernel execution state and exposes it through a minimal, race-safe, zero-copy interface.
This final project synthesizes all concepts introduced throughout the book: allocation rules, SMP synchronization, interrupt-context semantics, IOCTL and Netlink signaling, shared-memory mapping, and low-level x86-64 instrumentation. The resulting design is optimized for determinism, latency, and correctness on modern x86-64 Linux kernels.

8.1.1 Project Motivation

Traditional kernel observability tools suffer from inherent limitations:

- High latency introduced by tracing and sampling

- Significant configuration or instrumentation complexity
- Inability to capture instantaneous architectural state
- Repeated copy overhead on each query
- Frequent system calls and context switches

This module avoids those limitations by:

- Maintaining continuous, high-frequency telemetry
- Mapping kernel memory directly into user space
- Using sequence-based synchronization for consistency
- Employing Intel-syntax assembly for exact architectural state
- Operating entirely on preallocated memory

It forms a minimal yet powerful foundation for performance analysis, fault investigation, real-time monitoring, and kernel instrumentation education.

8.1.2 Telemetry Data Architecture

At the core of the module lies the per-CPU telemetry block.
Kernel-Resident Telemetry Structure

```
struct live_info {
    u64 seq;        /* sequence counter */
    u64 tsc;        /* timestamp counter */
    u64 rflags;     /* CPU flags register */
    u64 cr2;        /* last fault address */
    u64 cpu_id;     /* logical CPU id */
    u64 events;     /* update counter */
};
```

Design properties:

- Fits within a single cache line
- Contains only architecturally critical state
- Avoids padding and false sharing
- Easily extensible

Each CPU owns its own instance:

```
DEFINE_PER_CPU(struct live_info, cpu_info);
```

This eliminates cross-CPU contention entirely.

8.1.3 Shared Memory Allocation and Mapping

A single page of memory is allocated at module load time and mapped directly into user space.
Allocation

```
static void *shared_area;

static int ___init live_init(void)
{
    shared_area = kzalloc(PAGE_SIZE, GFP_KERNEL);
    if (!shared_area)
        return -ENOMEM;
    return 0;
}
```

Allocation occurs only during initialization to guarantee that no memory allocation is required during runtime.
Mapping

```
static int live_mmap(struct file *f,
                     struct vm_area_struct *vma)
{
    unsigned long pfn =
        virt_to_phys(shared_area) >> PAGE_SHIFT;

    return remap_pfn_range(vma,
                           vma->vm_start,
                           pfn,
                           PAGE_SIZE,
                           vma->vm_page_prot);
}
```

After mapping, user space reads kernel data through ordinary memory accesses without additional system calls.

8.1.4 Assembly-Level CPU State Extractors

All architectural extractors are implemented using constant-time, Intel-syntax assembly.
Read Timestamp Counter

```
.intel_syntax noprefix

.section .text
.globl read_tsc64
read_tsc64:
    rdtsc
    shl rdx, 32
    or  rax, rdx
    ret
```

Read RFLAGS

```
.intel_syntax noprefix

.section .text
.globl read_rflags
read_rflags:
    pushfq
    pop rax
    ret
```

Read CR2

```
.intel_syntax noprefix

.section .text
.globl read_cr2
read_cr2:
    mov rax, cr2
    ret
```

These helpers are deterministic, non-blocking, and safe in all kernel contexts where they are used.

8.1.5 Telemetry Update Engine

Telemetry updates follow a seqlock-inspired protocol.

```
static void update_live_info(void)
{
    struct live_info *info = this_cpu_ptr(&cpu_info);
    u64 seq = info->seq + 1;

    info->seq = seq;                /* begin update */
```

```
    info->tsc    = read_tsc64();
    info->rflags = read_rflags();
    info->cr2    = read_cr2();
    info->cpu_id = smp_processor_id();
    info->events++;

    smp_wmb();                  /* publish data */
    info->seq = seq + 1;        /* end update */
}
```

Odd sequence values indicate an in-progress update; even values indicate a stable snapshot.

8.1.6 Update Triggers

Updates may be triggered via:

- High-resolution timers
- Workqueues
- IOCTL commands
- Netlink requests
- Deferred IRQ bottom halves

Heavy operations are always deferred outside hard interrupt context.

8.1.7 User-Space Access Protocol

User space polls telemetry safely using sequence validation:

```
do {
    seq1 = info->seq;
    barrier();
    local = *info;
    barrier();
    seq2 = info->seq;
} while (seq1 != seq2  (seq1 & 1));
```

This guarantees consistent snapshots without locks or syscalls.

8.1.8 Race-Safety Properties

The design avoids races through:

- Per-CPU data ownership
- Sequence-based consistency
- No shared global state
- Preallocated memory only

No spinlocks or mutexes are required in the fast path.

8.1.9 Optional Extensions

Possible extensions include:

- CR3 (page-table base) capture
- GS-base inspection
- IRQ entry/exit tracing
- Hardware performance counters via RDPMC

8.1.10 Minimal Module Skeleton

```
extern u64 read_tsc64(void);
extern u64 read_rflags(void);
extern u64 read_cr2(void);

DEFINE_PER_CPU(struct live_info, cpu_info);
static void *shared_area;

static int ___init live_init(void)
{
    shared_area = kzalloc(PAGE_SIZE, GFP_KERNEL);
    return shared_area ? 0 : -ENOMEM;
}

static void ___exit live_exit(void)
{
    kfree(shared_area);
}

module_init(live_init);
module_exit(live_exit);
```

8.1.11 Conclusion

The “Live System Information” kernel module represents the culmination of modern low-level Linux development practices. It demonstrates how precise architectural state can be captured using assembly, synchronized across CPUs using sequence discipline, and exposed to user space through zero-copy shared memory.
This project embodies the core philosophy of the entire work: precision, correctness, determinism, and mastery of the kernel execution environment.

Conclusion

Linux kernel module development remains one of the most demanding disciplines in systems programming. Unlike user-space development—where layers of abstraction isolate the programmer from hardware details, memory ordering, and concurrency hazards—the kernel environment exposes the full complexity of the processor architecture, the Linux memory model, and the evolving execution semantics of modern SMP systems. This volume established a rigorous foundation for writing safe, performant, and correct kernel modules on contemporary x86-64 Linux kernels.
Throughout the chapters, the book developed a progressive architectural understanding of the kernel module lifecycle: initialization and teardown, memory allocation semantics, interrupt handling, user–kernel communication, and low-level state extraction. Each topic was framed not as an isolated API reference, but as part of a coherent execution model governed by strict constraints on context, ordering, and lifetime. Central to this approach were the principles of latency correctness, race safety, structural isolation, and architectural determinism—all of which are non-negotiable in production-quality kernel code.
Three core themes emerged as the pillars of correct kernel development:

1. Execution-Context Discipline.
   Kernel correctness depends on precise awareness of the current execution context: hard IRQ, softirq, workqueue, threaded interrupt, or process context. Each

context carries explicit constraints regarding sleeping, locking, memory allocation, and scheduling. Violating these constraints by invoking blocking operations, allocating memory with inappropriate GFP flags, or acquiring incompatible locks can corrupt global system state or deadlock the kernel. Robust kernel code is therefore engineered with exact, context-specific intent at every execution path.

2. Explicit Resource and Lifetime Management.
   Kernel modules must manage all resources with deterministic ordering and strict symmetry.

   - Dynamic memory allocations, slab objects, and page mappings require precise pairing and bounded lifetime.
   - Interrupt handlers demand balanced request_irq() and free_irq() semantics.
   - DebugFS entries, sysfs attributes, Netlink sockets, timers, workqueues, and shared-memory mappings must be dismantled in a controlled teardown sequence.

   Resource leaks in kernel space are permanent until reboot. Unlike user-space applications, there is no process isolation to contain mistakes. Kernel development therefore requires a level of rigor that exceeds conventional systems programming.

3. Architectural Precision.
   Modern x86-64 kernel development demands an intimate understanding of CPU state, memory barriers, cache coherence, and the Linux memory-ordering model. Inline assembly—such as rdtsc, pushfq, mov cr2, and lock-prefixed instructions—was introduced not as an academic curiosity, but as a necessary tool for implementing deterministic fast paths and architectural instrumentation. When applied with constant-time discipline and clear intent, assembly integrates naturally and safely into kernel logic.

The final project, the “Live System Information” kernel module, unified the full scope of this volume into a single coherent system: a continuously updating telemetry engine that captures real CPU state via Intel-syntax assembly, stores it in per-CPU structures, synchronizes updates using lightweight sequence protocols, and exposes the results to user space through zero-copy memory mappings. This project demonstrated the complete pipeline of modern kernel-module engineering—from raw register access to stable user-space visibility—implemented with correctness, determinism, and minimal overhead.

Above all, Linux kernel development demands a mindset grounded in engineering conservatism. Every line of code must be justified, every memory access must be intentional, and every synchronization primitive must be chosen with a full understanding of its architectural implications. Errors in kernel space are never local; they propagate globally and unpredictably.

As the Linux kernel continues to evolve—with new APIs, refined memory models, and emerging architectural features—the fundamental principles presented in this volume remain stable:

> Understand your execution context, manage resources explicitly, and treat kernel space as an environment where correctness must always precede convenience.

By completing this volume, the reader has acquired not only the practical ability to write kernel modules, but also the engineering perspective required to move deeper into the Linux kernel itself—into device drivers, filesystems, networking stacks, memory management, and the core subsystems that define Linux as a modern operating system. Subsequent volumes will build directly upon this discipline, extending the same principles from modular extensions into the kernel’s internal architecture.

References

The material presented in this booklet is grounded in the modern Linux kernel execution model, its post-2020 subsystem behavior, and the architectural rules of the x86-64 instruction set architecture. While kernel development evolves continuously, the technical foundations used throughout this volume rely on stable, long-standing kernel interfaces combined with carefully documented refinements introduced after 2020. The reference categories below define the conceptual and architectural basis for all explanations, synchronization models, module interfaces, memory semantics, and assembly-level integrations presented in this work.

Linux Kernel Module Framework and Core Subsystems

This booklet treats the Linux kernel source itself as the authoritative specification for module behavior, lifecycle, and integration. All descriptions of module loading, symbol resolution, and initialization ordering derive from the kernel's internal implementation rather than secondary documentation.
The primary foundations include:

- The kernel module loader, including object layout, relocation handling, symbol export rules, and module state transitions.

- The exported interfaces defined in linux/module.h, linux/init.h, and linux/kernel.h, including post-2020 changes affecting module metadata, licensing enforcement, and namespace control.
- Internal infrastructure governing module_init, module_exit, per-CPU data allocation, and livepatch-safe initialization sequencing.
- The Linux Kernel Memory Model, which formally defines ordering rules for smp_load_acquire, smp_store_release, smp_rmb, smp_wmb, and lockless per-CPU operations.

These components constitute the definitive specification for kernel module behavior and are treated as normative throughout this booklet.

Kernel Memory Allocation and Slab / SLUB Architecture

All explanations of kernel-space memory allocation are derived directly from the kernel's internal allocator implementations.
This includes:

- The SLUB allocator design and the physical page lifecycle on x86-64 NUMA systems.
- Precise semantics of GFP masks under process context, interrupt context, atomic context, and preemption-disabled execution.
- Post-2020 allocator refinements, including hardened freelists, read-once/write-once semantics, and initialization guarantees.

These subsystems define the only correct model for dynamic memory management inside kernel modules.

Interrupt Handling, IRQ Subsystem, and Deferred Execution

All material related to interrupts, deferred work, and asynchronous execution is based on the canonical Linux IRQ architecture.
Foundational elements include:

- The Generic IRQ Layer, including interrupt descriptors, handler action lists, softirq dispatch, and APIC routing on x86-64 systems.
- Hard IRQ entry and exit paths, interrupt masking rules, and preemption behavior.
- Threaded interrupt support and its interaction with real-time and PREEMPT kernel configurations.
- Post-2020 semantics of tasklets, softirqs, workqueues, and the strict safety constraints governing their interaction with user–kernel interfaces.

These components form the authoritative model for safe and correct asynchronous event handling in the Linux kernel.

User–Kernel Communication Mechanisms

All user–kernel communication techniques described in this booklet rely on ABI-stable kernel interfaces.
This includes:

- The IOCTL subsystem, including command encoding macros (_IOR, _IOW, _IOWR) that ensure cross-architecture compatibility and strict data direction semantics.

- The Netlink socket family and the Generic Netlink architecture, which define message framing, multicast delivery, and asynchronous communication behavior.

- Memory-mapping primitives such as remap_pfn_range, vm_operations_struct, and page-fault handlers used to implement zero-copy shared memory between kernel and user space.

These mechanisms represent the sanctioned and production-grade methods for structured kernel communication.

Debugging and Introspection Subsystems

All chapters covering /proc, seq_file, sysfs, and DebugFS are grounded in the kernel's internal virtual filesystem frameworks.
Specifically:

- The procfs and seq_file implementations used for safe, deterministic, streaming output.

- The sysfs infrastructure based on kobjects, reference counting, and strict attribute lifetimes.

- The DebugFS subsystem intended for developer-facing instrumentation and controlled exposure of kernel internals.

These subsystems define the modern mechanisms for exporting diagnostic and introspective data without violating kernel safety constraints.

x86-64 Architectural Specification

All assembly-level routines and low-level state extraction paths depend on the architectural guarantees of the AMD64 (x86-64) ISA.
This includes:

- Canonical definitions of RFLAGS, segment bases, control registers (cr2, cr3), model-specific registers, and timestamp counters.
- The x86-64 memory-ordering model, including its strong ordering guarantees and its interaction with Linux's explicit memory-barrier semantics.
- The behavior of privileged instructions such as rdtsc, rdmsr, mov crX, and serialization instructions used in constant-time telemetry and synchronization paths.

These architectural rules form the foundation of all assembly examples presented in this volume.

Post-2020 Linux Kernel Evolution

The conceptual framework of this booklet aligns with the post-2020 evolution of the Linux kernel, including:

- Enhanced symbol namespace isolation and module-level namespace annotations.
- Explicit formalization of synchronization primitives and memory-model rules.
- Hardening improvements in the SLUB allocator and memory-safety tooling.
- Advances in IRQ threading, softirq isolation, and CPU scheduling.

- Expanded diagnostic infrastructure, including Kernel Memory Sanitizer (KMSAN), stack-protector hardening, and fault-analysis facilities.

These changes reflect the operational environment of modern production kernels.

Project-Level Foundations for the "Live System Information" Module

The final project integrates kernel subsystems and architectural mechanisms drawn from:

- The per-CPU subsystem and its architecture-specific guarantees.
- Shared-memory mapping semantics implemented in the core memory management subsystem.
- Kernel timekeeping and timestamp-counter synchronization.
- IRQ-safe and preemption-safe telemetry capture strategies.
- Inline assembly conventions consistent with the x86-64 System V ABI as implemented by the Linux kernel.

Together, these components define the authoritative model for building low-overhead instrumentation modules without external dependencies.

Closing Statement

The references presented here are not a bibliography of external texts, but a structured outline of the kernel-internal specifications, architectural rules, and subsystem

implementations that define Linux kernel behavior. Kernel development is guided by these primary sources, as they embody the actual execution model of Linux on modern x86-64 hardware.

Accordingly, all techniques, synchronization idioms, assembly routines, and module designs in this booklet are based on verified, post-2020, architecture-conscious kernel behavior—reflecting not historical conventions, but the contemporary, production-grade Linux kernel in active use today.