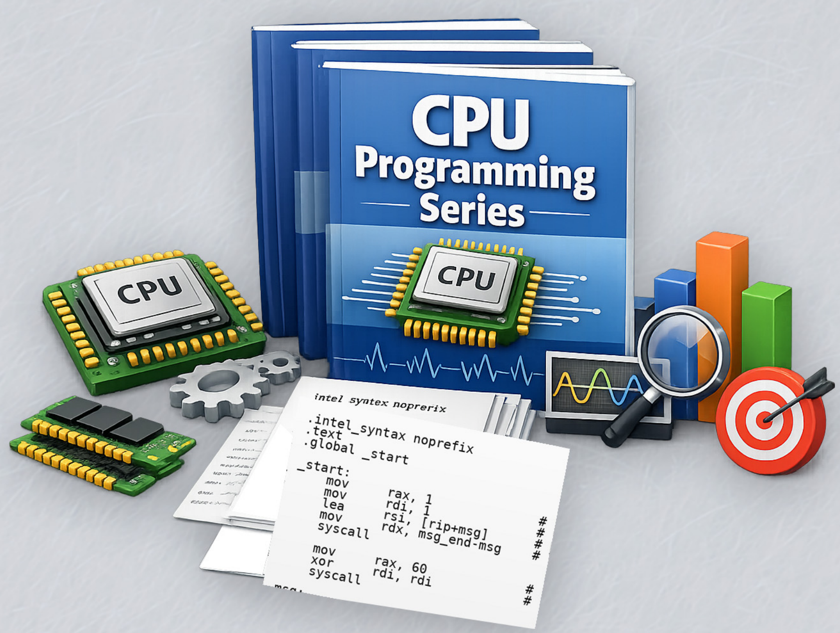
https://simplifycpp.org

CPU Programming Series

x86-64 Syscalls & Privilege Boundary

From User Code to the Kernel



```
intel syntex noprefix
.intel_syntax noprefix
.text
.global _start
_start:
    mov     rax, 1
    mov     rdi, 1
    lea     rsi, [rip+msg]        #
    mov     rdx, msg_end-msg      #
    syscall                       #
                                  #
    mov     rax, 60
    xor     rdi, rdi              #
    syscall                       #
```

10

Prepared by Ayman Alheraki

CPU Programming Series

86-64 Syscalls Privilege Boundary

From User Code to the Kernel

Prepared by Ayman Alheraki

simplifycpp.org

January 2026

Contents

Preface

Scope of This Booklet

This booklet focuses on the **x86-64 System V ABI** as it is implemented on modern UNIX-like systems, with a strict emphasis on **calling conventions**, **stack discipline**, and **compiler–ABI contracts**. It explains how function calls are actually realized at the machine level: how arguments are passed, how return values are delivered, how the stack frame is formed, aligned, and destroyed, and how registers are preserved across calls.
The scope is intentionally limited to **user-space execution** and **language-independent ABI rules**. Kernel entry mechanisms, syscalls, interrupts, and privilege transitions are explicitly out of scope and addressed in subsequent booklets.

Position Within the CPU Programming Series

This booklet occupies a critical position in the CPU Programming Series. It builds directly upon prior booklets covering registers, flags, data representation, memory layout, and stack fundamentals, and serves as the final mandatory step before entering operating system interaction and kernel boundaries.
Conceptually, it is the **bridge between pure CPU mechanics and real-world compiled programs**. Without a precise understanding of the System V ABI, it is impossible to correctly

analyze compiler-generated assembly, debug optimized binaries, reason about inter-language calls, or write reliable hand-crafted assembly that interoperates with C, C++, Rust, or other compiled languages.

What This Booklet Assumes (and Does Not)

This booklet assumes that the reader already understands:

- General-purpose and SIMD registers in x86-64
- Basic stack operation (`RSP`, push/pop, call/ret)
- Binary data representation and alignment concepts
- Reading basic assembly output from compilers

It does **not** assume:

- Prior knowledge of operating system internals
- Knowledge of syscalls, interrupts, or kernel code
- Familiarity with a specific high-level language

The focus is not on syntax memorization, but on **mechanical rules**, **invariants**, and **guarantees** enforced by the ABI.

Why Syscalls and Privilege Boundaries Matter

Although this booklet is strictly user-space focused, understanding the System V ABI is a prerequisite for understanding **syscalls and privilege transitions**. The ABI defines the last stable execution environment before control is transferred to the operating system.

For example, a normal function call preserves a well-defined subset of registers and uses a disciplined stack layout:

```
# System V function call discipline (conceptual)
# RDI, RSI, RDX, RCX, R8, R9  -> arguments
# RAX                         -> return value
# RBX, RBP, R12-R15           -> callee-saved

call my_function
```

In contrast, a syscall crosses a privilege boundary where:

- The calling convention changes
- The stack may switch
- Register preservation rules are different

Understanding where the ABI **ends** is essential to understanding where the kernel **begins**. This booklet therefore prepares the reader to correctly reason about the boundary between user code and the operating system, which is explored in the next stage of the series.

Chapter 1

Privilege Levels in x86-64

1.1 User Mode vs Kernel Mode

Modern x86-64 processors operate with multiple privilege levels, known as **rings**. In practice, only two are actively used by contemporary operating systems:

- **User Mode (Ring 3)**: where applications execute
- **Kernel Mode (Ring 0)**: where the operating system kernel executes

User mode code is deliberately restricted. It cannot directly access hardware devices, privileged CPU instructions, kernel memory, or critical control registers. Any attempt to do so results in a fault or exception.
Kernel mode code, by contrast, has unrestricted access to the processor and system resources. This asymmetry is fundamental to system stability and security.

```
# User-space instruction (allowed)
mov rax, rbx
```

```
# Privileged instruction (not allowed in user mode)
cli                 # Clear interrupt flag -> causes fault in Ring 3
```

The processor enforces these rules automatically; software cannot bypass them.

1.2 Why Privilege Separation Exists

Privilege separation exists to protect the system from:

- Application bugs
- Malicious code
- Accidental corruption of shared resources

Without privilege separation, a single faulty application could overwrite kernel memory, reprogram hardware, or halt the entire machine. By confining applications to user mode, the operating system ensures that failures are isolated and recoverable.
From a design perspective, this separation also defines a clean **contract**:

- Applications request services
- The kernel validates and performs them

This contract is enforced through controlled entry points, not trust.

1.3 Hardware-Enforced Boundaries (Conceptual)

The boundary between user mode and kernel mode is enforced by the CPU itself, not by convention. Several hardware mechanisms participate in this enforcement:

- Current Privilege Level (CPL)
- Page table permission bits
- Instruction privilege checks
- Model-Specific Registers controlling transitions

When executing in user mode, the CPL is set to the least-privileged level. Memory accesses are checked against page permissions, and privileged instructions are rejected.
Transitions across the boundary can occur only through specific mechanisms designed for this purpose, such as system calls. Arbitrary jumps into kernel code are impossible.

```
# Illegal attempt to jump into kernel space
jmp 0xffffffff81000000    # Kernel address -> page fault
```

This enforcement is entirely automatic and occurs on every instruction and memory access.

1.4 Transition Costs and Security Implications

Crossing the privilege boundary is intentionally expensive compared to a normal function call. A transition requires:

- CPU mode switch
- Register state changes
- Stack context changes
- Strict validation of inputs

These costs discourage frequent boundary crossings and encourage batching of kernel requests. Performance-sensitive software must account for this overhead.
From a security standpoint, every transition represents a potential attack surface. For this reason:

- Entry paths are minimal and tightly controlled
- Inputs are validated aggressively
- Execution context is carefully reconstructed on return

Understanding these costs and risks is essential before studying system calls themselves. This chapter establishes the conceptual foundation required to analyze how user code safely and efficiently requests kernel services, which is the focus of the next chapter.

Chapter 2

From User Code to the Kernel

2.1 What Happens When User Code Needs the Kernel

User code routinely needs services that it is not allowed to perform directly, such as:

- creating or managing processes and threads
- opening files and performing I/O
- allocating memory with kernel involvement (e.g., mapping pages)
- networking, timers, and device access

Because user mode is restricted, the only correct path is to **request** the kernel to perform the operation. Conceptually, this is a controlled call across the privilege boundary:

1. User code prepares a **syscall number** and its arguments.
2. A dedicated instruction transfers control to a privileged entry point.
3. The kernel validates the request, performs it, and returns a result or an error.

4. Control returns to user mode at the instruction following the transition.

A syscall is therefore closer to **an architectural boundary crossing** than to a normal function call. A function call assumes mutual trust and shared address space rules; a syscall assumes the opposite and is designed to survive hostile or corrupted inputs.

```
# Concept-only shape of a Linux x86-64 system call (details later)
# rax = syscall number
# rdi, rsi, rdx, r10, r8, r9 = args (Linux syscall ABI)

mov rax, 60           # __NR_exit
xor rdi, rdi          # status = 0
syscall               # enter kernel, never returns
```

Even when the syscall does return, it may also be affected by signals and restart rules (covered later in the booklet).

2.2 Historical Overview: INT 0x80 vs SYSCALL

On 32-bit Linux, system calls were traditionally issued with a software interrupt:

```
# IA-32 Linux (historical pattern)
mov eax, 1            # __NR_exit (IA-32)
xor ebx, ebx          # status = 0
int 0x80              # software interrupt -> kernel
```

This mechanism works, but it routes through the interrupt/exception machinery and typically involves more overhead than a dedicated fast syscall path.
On x86-64 (and late x86 generations), CPUs introduced dedicated instructions for fast transitions:

- **SYSCALL/SYSRET** for 64-bit mode
- **SYSENTER/SYSEXIT** historically for some 32-bit fast paths

`syscall` is designed specifically to reduce transition cost by using a dedicated entry mechanism configured by model-specific registers, rather than the more general interrupt gate path.

```
# x86-64 Linux preferred mechanism
mov rax, 60          # __NR_exit
xor rdi, rdi
syscall
```

Practical takeaway: modern x86-64 user space uses `syscall`, while `int 0x80` remains primarily a legacy interface for compatibility (and is not the normal path for 64-bit processes).

2.3 Controlled Entry Points into the Kernel

The kernel is not entered by arbitrary branching. The CPU enforces that ring transitions occur only through:

- system call entry instructions (e.g., `syscall`)
- exceptions and faults (e.g., page fault, general protection fault)
- interrupts (e.g., timer interrupt, device interrupts)

These entry paths are **controlled** because they land on specific kernel entry stubs that:

- establish a trusted execution context (stack and CPU state)
- save the minimum required registers

- validate the user request and pointers
- dispatch to the appropriate kernel handler

An important distinction:

- **Syscall entry** is **intentional**: user code requests a kernel service.
- **Exception entry** is **reactive**: the CPU detects illegal behavior or a fault.

Example: trying to execute a privileged instruction from user mode triggers an exception entry path, not a syscall.

```
# User-mode attempt to modify interrupt state is illegal
cli                  # triggers a fault -> kernel exception handler
```

Both syscalls and exceptions cross the same privilege boundary, but they exist for different reasons and follow different conventions.

2.4 Fast Path vs Slow Path Transitions

Even within syscalls, not every entry/exit is equal. Kernels generally implement:

- **Fast paths**: common operations optimized for minimal overhead
- **Slow paths**: rare, complex, or exceptional cases requiring extra work

Fast paths aim to:

- minimize register saving
- avoid expensive checks when not needed (while preserving safety)
- reduce branching and avoid heavy kernel subsystems

Slow paths appear when additional machinery is required, for example:

- blocking I/O (may sleep and reschedule)
- page faults during copy to/from user space
- signal delivery or interruption of a syscall
- tracing, auditing, or security policy checks
- uncommon error cases and retries

Conceptual illustration: a syscall that must copy user memory cannot blindly trust the pointer. It must validate and handle faults safely, which may force a slow path.

```
# Concept: user provides a pointer in RSI
# Kernel must treat this as untrusted input:
# - verify accessibility
# - handle page faults during copy
# - return -EFAULT on invalid memory
```

Practical takeaway: understanding syscall performance requires distinguishing:

- the fixed transition overhead of entering/exiting the kernel
- the variable cost of what the kernel must do (fast vs slow path)

This sets the stage for the next chapter, where we formalize the `syscall` instruction behavior and the Linux x86-64 syscall ABI in detail.

Chapter 3

The `syscall` Instruction

3.1 Purpose and Design Goals of `syscall`

The `syscall` instruction is a hardware-defined mechanism introduced to provide a **fast, minimal, and strictly controlled transition** from user mode to kernel mode on x86-64 processors. It replaces older, more general mechanisms such as software interrupts for the common case of system calls.
Its primary design goals are:

- **Explicit privilege transition**: enforce a Ring 3 to Ring 0 switch under full CPU control.
- **Low latency**: avoid the overhead of the full interrupt/exception dispatch machinery.
- **Predictable execution model**: always enter the kernel at a single, preconfigured entry point.
- **Minimal architectural side effects**: preserve only what is required for a correct return.

Unlike a normal function call, `syscall` does not assume trust, shared conventions, or a shared stack. It is designed for an adversarial boundary where user code may be buggy or malicious.

```
# Linux x86-64 syscall invocation (conceptual)
# rax = syscall number
# rdi, rsi, rdx, r10, r8, r9 = arguments

mov rax, 39          # __NR_getpid
syscall
# rax contains return value or negative error
```

3.2 CPU State Before and After `syscall`

Before executing `syscall`, the processor is executing in user mode with the following relevant state:

- Current Privilege Level (CPL) = 3
- `RIP` points to the `syscall` instruction
- `RSP` refers to the user stack
- General-purpose registers hold syscall number and arguments

When `syscall` is executed, the CPU performs an architectural transition with well-defined effects:

- CPL is set to 0 (kernel mode)
- `RIP` is loaded from a kernel entry address

- The return instruction pointer is saved into `RCX`
- The user `RFLAGS` value is saved into `R11`
- Selected flags in `RFLAGS` are cleared according to configuration

Crucially, no return address is pushed onto the stack. The return context is carried entirely in registers.

```
# Architectural effects of SYSCALL (conceptual)
# RCX <- user RIP (next instruction)
# R11 <- user RFLAGS
# RIP <- kernel entry RIP
# CPL <- 0
```

Once inside the kernel, early entry code typically switches to a kernel-controlled stack, since the user stack cannot be trusted for privileged execution.

3.3 MSRs Involved in Syscall Handling

The `syscall`/`sysret` mechanism is configured through a small set of model-specific registers (MSRs). These registers define the execution environment of the transition and are programmed by the operating system during boot.
Conceptually, the MSRs control:

- The kernel entry instruction pointer for `syscall`
- The code segment selectors used in kernel and user modes
- Which flags are masked on entry to the kernel

Key roles (names omitted here intentionally):

- An MSR that provides the kernel entry `RIP`
- An MSR that encodes kernel and user code segment selectors
- An MSR that defines which bits in `RFLAGS` are cleared on entry

These registers ensure that user code cannot influence where execution enters the kernel or how privilege state is established. Once configured, every `syscall` instruction uses the same trusted entry path.

3.4 SYSCALL vs SYSRET (Conceptual Flow)

`syscall` and `sysret` form a paired mechanism:

- `syscall`: transitions from user mode to kernel mode
- `sysret`: returns from kernel mode back to user mode

Conceptually, the flow is:

1. User code executes `syscall`
2. CPU saves return context into `RCX` and `R11`
3. Kernel executes syscall handler logic
4. Kernel prepares return value in `RAX`
5. Kernel executes `sysret` to resume user execution

```
# Conceptual syscall lifecycle
# User:
mov rax, 1
```

```
syscall

# Kernel:
# handle request
# prepare return value
sysret
```

`sysret` restores execution to user mode using the saved `RCX` (instruction pointer) and `R11` (flags). The CPU reestablishes CPL=3 and resumes execution immediately after the original `syscall` instruction.

This paired design minimizes overhead while maintaining strict control over privilege transitions, making `syscall`/`sysret` the fundamental mechanism for user–kernel interaction on modern x86-64 systems.

Chapter 4

Linux x86-64 Syscall ABI

4.1 Syscall Calling Convention Overview

On Linux x86-64, a system call is invoked with the `syscall` instruction and follows a calling convention that is **distinct from the System V function-call ABI**. The syscall ABI is designed for a privileged boundary crossing where the kernel must treat all user inputs as untrusted.

The syscall interface consists of:

- A **syscall number** identifying the requested service
- Up to **six arguments** passed in registers
- A **return value** (or error indicator) delivered in `RAX`

High-level model:

1. Place syscall number in `RAX`
2. Place arguments in designated registers

3. Execute `syscall`

4. Read return value from `RAX`

```
# Minimal syscall pattern (Linux x86-64)
# rax = syscall number
# rdi, rsi, rdx, r10, r8, r9 = args
# return in rax

mov rax, 39          # __NR_getpid
syscall
# rax = pid (>= 0) on success
```

This convention is stable for user space and is what assembly code, runtimes, and libc ultimately rely on when issuing syscalls.

4.2 Register Usage and Argument Passing

Linux x86-64 uses the following register mapping for syscall arguments:

- **Syscall number:** `RAX`
- **Arg1:** `RDI`
- **Arg2:** `RSI`
- **Arg3:** `RDX`
- **Arg4:** `R10`
- **Arg5:** `R8`
- **Arg6:** `R9`

The use of R10 for the 4th argument (instead of RCX) is not arbitrary: RCX is overwritten by the syscall instruction (it receives the user return RIP), so it cannot safely carry an argument across the transition.

Example: write(fd, buf, count)

```
# ssize_t write(int fd, const void* buf, size_t count)
# __NR_write = 1 on Linux x86-64

mov rax, 1             # __NR_write
mov rdi, 1             # fd = 1 (stdout)
lea rsi, [rel msg]     # buf
mov rdx, msg_len       # count
syscall

# rax = bytes written (>= 0) or -errno (< 0)

msg:
    .ascii "Hello from syscall\n"
msg_len = . - msg
```

Example with 4 arguments: openat(dirfd, pathname, flags, mode) (conceptual argument placement only)

```
# long openat(int dirfd, const char* pathname, int flags, mode_t
↪  mode)
# rdi = dirfd
# rsi = pathname
# rdx = flags
# r10 = mode   (4th arg)
```

```
mov rax, 257           # __NR_openat (number shown as an example)
mov rdi, -100          # AT_FDCWD (conceptual)
lea rsi, [rel path]
mov rdx, 0             # flags (conceptual)
mov r10, 0             # mode
syscall

path:
    .ascii "/etc/hostname\0"
```

The kernel treats all pointers (`RSI` in the examples) as untrusted user addresses and must validate access during copy.

4.3 Return Values and Error Reporting

The syscall return value is delivered in `RAX`.

- On **success**, `RAX` contains a non-negative result (often a value, count, or file descriptor).
- On **failure**, Linux returns a **negative error number** in `RAX` (e.g., `-EFAULT`, `-EINVAL`).

This is different from the user-facing C library behavior: libc typically converts negative returns into `-1` and stores the positive error code in `errno`. Pure assembly callers must handle the kernel convention directly.
Example: handle success vs error (concept-only)

```
# After syscall:
# rax >= 0   => success
# rax < 0    => -errno
```

```
syscall
test rax, rax
jns .ok                    # jump if non-negative
neg rax                    # rax = errno (positive)
# handle error in rax
jmp .done

.ok:
# handle success (rax = result)

.done:
```

This rule is consistent across syscalls: the sign of `RAX` indicates whether an error occurred.

4.4 Clobbered Registers and ABI Guarantees

At the syscall boundary, you must assume the kernel may overwrite most registers while executing the syscall handler. However, there are two architectural clobbers that are always relevant at the user boundary:

- `RCX` is overwritten (holds user return `RIP`)
- `R11` is overwritten (holds saved user `RFLAGS`)

Therefore, user code must treat `RCX` and `R11` as **clobbered by `syscall`**.
In addition, because syscalls are privileged operations that may trigger scheduling, signal handling, and deep kernel code paths, correct user-space assembly should follow a conservative rule:

Assume that any register not explicitly part of the syscall interface is not preserved across the syscall, unless your calling environment guarantees otherwise.

In practice, when writing syscall wrappers in assembly:

- Save any values you must keep across the syscall
- Do not use `RCX` for arguments (use `R10` for Arg4)
- Expect flags to be modified (do not depend on condition codes after return)

Example: preserving a value across syscall

```
# Preserve RBX across syscall (example technique)
push rbx
mov rbx, 12345

mov rax, 39          # getpid
syscall

# rax = pid
pop rbx
```

4.5 Comparison with Function Call ABI

The Linux x86-64 syscall ABI differs from the System V AMD64 function-call ABI in critical ways:

- **Entry mechanism:** function call uses `call`; syscall uses `syscall`
- **Privilege level:** function call stays in user mode; syscall switches to kernel mode

- **Argument registers:**
  - Function call: `RDI, RSI, RDX, RCX, R8, R9`
  - Syscall: `RDI, RSI, RDX, R10, R8, R9` (note `R10` instead of `RCX`)
- **Clobbers:**
  - Function call clobbers depend on caller-saved rules
  - Syscall always clobbers `RCX` and `R11` at the boundary
- **Error reporting:**
  - Function call typically uses return value conventions and library-level error handling
  - Syscall returns negative `-errno` in `RAX` on failure

Concrete side-by-side illustration (argument registers only):

```
# Function call (SysV AMD64 ABI):
# arg1=rdi arg2=rsi arg3=rdx arg4=rcx arg5=r8 arg6=r9
call some_function

# Syscall (Linux x86-64 ABI):
# arg1=rdi arg2=rsi arg3=rdx arg4=r10 arg5=r8 arg6=r9
# rax=syscall_number
syscall
```

Practical takeaway: do not mix these two conventions. A correct syscall wrapper must follow the syscall ABI exactly, and a correct function call must follow the System V function ABI. Confusing `RCX` vs `R10` is one of the most common sources of subtle bugs in hand-written syscall code.

Chapter 5

Kernel Entry and Exit Mechanics

5.1 Stack Switching and Kernel Stacks

When user code enters the kernel (via `syscall`, an exception, or an interrupt), the kernel must immediately execute with a **trusted stack**. The user stack pointer (`RSP`) is under user control and cannot be relied upon for privileged execution.
Two core reasons the kernel does not run on the user stack:

- **Trust**: user `RSP` may point to invalid/unmapped memory or memory crafted to trigger faults.
- **Isolation**: kernel stack must not be writable by user processes.

On x86-64, modern kernels use per-thread or per-CPU kernel stacks. On entry, the kernel quickly switches to its kernel stack and only then performs heavier work.
Conceptual flow:

1. CPU transfers control to a kernel entry point (privileged `RIP`).
2. Entry code switches to the kernel stack.

3. Minimal state is saved and a kernel “frame” is built.
4. The syscall/exception/interrupt is dispatched.

Concept-only illustration (not a real kernel stub):

```
# Concept: early entry code switches stacks
# user_rsp is untrusted
# kernel_rsp is trusted (obtained from kernel structures)

mov r12, rsp          # save user rsp temporarily (concept)
mov rsp, r13          # switch to kernel rsp (concept)
# now safe to push/save state
```

Important note: for `syscall` specifically, the instruction itself does not push a return address. The kernel must build its own entry frame and preserve return context using the architectural conventions of the `syscall/sysret` mechanism.

5.2 Context Saving Responsibilities

The kernel must preserve enough CPU state to:

- resume the interrupted user code correctly
- protect kernel correctness during nested events
- potentially perform scheduling (switch to a different task)

A useful way to think about kernel entry state is to split it into layers:

- **Architectural return context**: what is strictly required to return to user space.
- **Kernel working context**: additional registers/state used while executing kernel code.

- **Scheduling context**: the state needed to suspend the current task and later resume it.

For a `syscall` entry on x86-64, the CPU already provides two key pieces of the user return context:

- `RCX` contains the user return instruction pointer
- `R11` contains the saved user `RFLAGS`

Therefore, correct kernel entry code must treat `RCX` and `R11` as part of the return context and preserve them across the syscall handling path.
Concept-only illustration:

```
# Concept: preserve syscall return context early
push rcx                # user RIP for SYSRET
push r11                # user RFLAGS for SYSRET
# save other registers as required by kernel convention
```

In addition, kernel code must be prepared for asynchronous events (interrupts) and synchronous faults (page faults) that can occur while it is servicing a syscall, and it must preserve state in a way that supports nested handling safely.

5.3 Returning Safely to User Space

Returning to user space is not a simple inverse of entry. The kernel must re-establish:

- **User privilege level** (Ring 3)
- **User instruction pointer** (resume point)
- **User flags** (with enforced constraints)
- **User stack pointer** (restored user `RSP`)

For syscalls, the conceptual return mechanism is `sysret`, which returns to the user instruction pointer held in `RCX` and restores flags from `R11` (subject to architectural masking rules).

Concept-only return sequence:

```
# Concept: prepare return value and restore context
# rax already holds return value
pop r11                 # restore user RFLAGS (concept)
pop rcx                 # restore user RIP (concept)
mov rsp, r12            # restore user RSP (concept)
sysret                  # return to user mode
```

The kernel must ensure that the return context is **safe** and **canonical** for x86-64 user space. Returning with invalid addresses or inconsistent state must be prevented, even if the user input attempted to provoke it.

Practical rule for user-space assembly writers:

- Do not assume any flags are preserved across a syscall.
- Do not assume `RCX` and `R11` survive a syscall.
- Always treat the syscall as a boundary that may schedule, fault, or be interrupted.

5.4 Common Failure and Security Scenarios

Kernel entry/exit paths are among the most security-critical code in an operating system because they operate on untrusted inputs at the highest privilege level. Common failure patterns fall into a few categories:

5.4.1 Invalid Pointers and Faults During Copy

Syscalls often accept user pointers (buffers, strings, structures). These pointers may be:

- unmapped
- mapped but not readable/writable as required
- mapped but crossing into invalid pages

A correct kernel must handle faults during copy and return an error rather than crashing. Concept-only user-side scenario:

```
# write(fd=1, buf=0x1, count=16) -> invalid pointer
mov rax, 1
mov rdi, 1
mov rsi, 1            # invalid user pointer
mov rdx, 16
syscall
# rax < 0  (e.g., -EFAULT) expected on failure
```

5.4.2 Signals Interrupting Syscalls

A syscall may be interrupted by a signal, producing an early return with an error indicating interruption. The exact restart behavior depends on kernel policy and user-space signal handling configuration. The key mechanical point is that syscall completion is not guaranteed to be atomic with respect to signal delivery.

```
# Concept: syscall can return early due to signal
# user code must handle "interrupted" outcomes correctly
syscall
test rax, rax
```

```
jns .ok
# error path (may include interruption)
```

5.4.3 Privilege Confusion and Entry Point Integrity

The kernel must guarantee that:

- user code cannot choose the kernel entry address
- user code cannot execute privileged instructions directly
- return to user mode cannot be redirected to kernel addresses

These are enforced by CPU privilege checks and the kernel's strict validation of return state.

5.4.4 Register Convention Mismatches

A frequent correctness bug in low-level user code is mixing:

- the System V function-call ABI
- the Linux syscall ABI

The most common pitfall is the 4th argument register:

- function ABI uses `RCX`
- syscall ABI uses `R10`

```
# Wrong for syscalls (RCX is not arg4)
# rcx will be clobbered by SYSCALL
mov rcx, 123          # WRONG for arg4

# Correct for syscalls
mov r10, 123          # arg4
```

5.4.5 Hidden Costs: Scheduling and Slow Paths

Even when the transition itself is fast, the kernel may:

- block the calling thread
- schedule another thread
- fault and resolve memory mappings

Therefore, “syscall cost” must be understood as:

- fixed entry/exit overhead
- plus variable work (fast vs slow path)

Correct systems programming treats syscalls as boundary crossings with both performance and security consequences, not as ordinary calls.

Chapter 6

Signals and Exceptions (Conceptual)

6.1 Difference Between Signals and Syscalls

A **syscall** is an intentional request from user space to the kernel:

- It is initiated by user code (explicitly).
- It asks the kernel to perform a privileged service (I/O, process control, memory mapping, etc.).
- It follows a defined ABI: syscall number + arguments in registers + return value in `RAX`.

A **signal** is a kernel-delivered notification to user space:

- It is delivered by the kernel to a process/thread.
- It represents an event (fault, timer, external request, child status change, etc.).
- It may interrupt normal user execution and run a user-registered handler.

A practical mental model:

- Syscall: **user** asks the **kernel** for service.
- Signal: **kernel** interrupts the **user** to notify an event.

Example: user performs a syscall intentionally (write), while a signal can arrive at any instruction boundary.

```
# Intentional: syscall
mov rax, 39          # __NR_getpid
syscall              # user -> kernel -> user

# Not intentional: a signal may interrupt execution here
nop
nop
```

This chapter is conceptual: it explains the mechanics that matter for syscall correctness and privilege-boundary reasoning, without going into full POSIX signal programming.

6.2 Synchronous vs Asynchronous Events

In low-level execution terms, events that cross into the kernel can be categorized by whether they are caused directly by the currently executing instruction.

6.2.1 Synchronous events

A synchronous event is directly triggered by the current instruction. Examples include:

- **Syscall entry**: `syscall` instruction explicitly transfers control.
- **Faults and exceptions**: illegal instruction, divide error, general protection fault, page fault.

If the same instruction is re-executed under the same conditions, it will typically reproduce the same synchronous event.
Conceptual example: executing a privileged instruction in user mode triggers a synchronous fault.

```
# User mode cannot execute this
cli                   # synchronous fault -> kernel exception path
```

6.2.2 Asynchronous events

An asynchronous event is not caused by the current user instruction. It arrives due to external or concurrent conditions. Examples include:

- Hardware interrupts (timer tick, device completion)
- Signal delivery triggered by another thread/process (e.g., termination request)
- Child process state changes (conceptually delivered as a signal)

Asynchronous events can arrive between instructions, and their timing is not controlled by user code.
The important connection to syscalls: asynchronous events (signals) may **interrupt** a thread that is inside a syscall or about to return from one.

6.3 Signal Delivery from Kernel to User Space

Signal delivery is a controlled kernel-to-user transition. Conceptually, when the kernel decides to deliver a signal to a thread, it must:

1. Choose a delivery point (typically when returning to user mode or at a safe interruption boundary).

2. Save the interrupted user context (register state, instruction pointer, flags, stack state).
3. Arrange for user code to execute a signal handler (if installed), or apply a default action.
4. Provide a way to resume the interrupted execution after the handler completes.

From the user’s perspective, signal delivery looks like an **unexpected call** that happens “by itself”:

- it can interrupt normal flow
- it runs on the user stack (unless an alternate signal stack is configured)
- it eventually returns to the interrupted instruction stream

Conceptual user-space view:

```
# User execution (conceptual timeline)
# ...
# instruction A
# instruction B
# [signal arrives]
# -> handler runs
# -> returns
# instruction C continues
```

Key idea for systems programmers: signal delivery requires the kernel to construct a user-mode execution frame that is safe, correctly aligned, and consistent with the ABI expectations of user-space code.

6.4 Interaction Between Signals and Syscalls

Signals and syscalls interact in several important ways that affect correctness and robustness of low-level code.

6.4.1 Syscalls may be interrupted

A blocking syscall (or even a fast syscall under certain circumstances) may be interrupted by a signal, causing the syscall to return early with an error indication. User code must be prepared for this.

Concept-only pattern for robust syscall wrappers:

- check `RAX` for negative error
- handle interruption by retrying when appropriate

```
# Concept: syscall wrapper loop (error handling shape only)
.retry:
syscall
test rax, rax
jns .ok                # success (rax >= 0)

# rax is negative -errno
# A robust wrapper may retry on specific interruption cases (policy
 ↪  dependent)
# Here we just expose the error
neg rax                # rax = errno (positive)
jmp .done

.ok:
# rax = result
.done:
```

6.4.2 Syscall restart rules

Some syscalls are restartable depending on how signal handlers are configured and the kernel's restart policy. The important conceptual point is:

> A syscall boundary is not guaranteed to be a simple single-shot request/response; it may return early and require user-space policy to decide whether to retry.

Therefore, assembly-level syscall code must treat "interrupted syscall" as a normal, expected outcome.

6.4.3 Signals can arrive right before or right after syscall return

Signals are often delivered when the kernel is about to return to user space. This means:

- the thread can appear to "return from syscall into handler" rather than into the next user instruction
- user code must not assume immediate continuation after syscall return

6.4.4 Exception-like signals (faults)

Some signals correspond to synchronous faults (e.g., invalid memory access). Conceptually:

- a user-space fault triggers a kernel exception path
- the kernel converts it into a user-visible signal (or terminates the process)

This is why it is useful to consider "signals and exceptions" together: many signals are the user-visible consequence of low-level exceptions, and the delivery path itself is another controlled kernel-to-user transition.
Practical takeaway: to write correct low-level code around syscalls, you must assume:

- syscalls can fail with negative errors
- syscalls can be interrupted
- control flow may jump to a handler between instructions
- after a signal handler, execution resumes as if an unexpected call occurred

Chapter 7

Context Save and Restore Concepts

7.1 What "Context" Really Means

In low-level execution, a **context** is the complete set of architectural and logical state required to resume execution **as if no interruption occurred**. It is not a single structure, but a collection of CPU-visible and kernel-managed state.
At minimum, execution context includes:

- Instruction pointer (`RIP`)
- Stack pointer (`RSP`)
- Flags register (`RFLAGS`)
- General-purpose registers

In practice, a full execution context may also include:

- SIMD and floating-point registers

- Control registers and thread-local state
- Memory mapping context (address space identifier)
- Kernel scheduling metadata

The defining property of a correctly saved context is **transparency**: after restore, the code continues exactly where it left off, with no observable difference except for effects explicitly caused by the event (syscall result, signal handler execution, etc.).
Conceptual illustration:

```
# Context must preserve the illusion:
# Before interruption:
add rax, rbx

# After interruption and restore:
# Execution resumes here exactly as expected
add rax, rbx
```

Any omission or corruption of context breaks this illusion and results in incorrect behavior.

7.2 Minimal vs Full Context Saving

Not every kernel entry requires saving the same amount of state. A key performance principle is to save **only what is necessary**.

7.2.1 Minimal context saving

Minimal context saving is used when:

- the kernel knows it will return quickly

- no task switch will occur
- no complex kernel subsystems will be invoked

In this case, the kernel preserves only:

- architectural return state (e.g., `RCX`, `R11` for `syscall`)
- registers it actively uses

Concept-only example:

```
# Minimal save (conceptual)
push rcx              # user RIP
push r11              # user RFLAGS
# handle fast syscall
pop r11
pop rcx
```

This approach minimizes memory traffic and latency.

7.2.2 Full context saving

Full context saving is required when:

- the kernel may block or sleep
- a context switch to another task may occur
- signals or interrupts require deep kernel paths

A full save typically includes all general-purpose registers and, when necessary, extended register state.
Concept-only illustration:

```
# Full save (conceptual shape)
push rax
push rbx
push rcx
push rdx
push rsi
push rdi
# plus additional registers as required
```

The kernel must ensure that the saved context is sufficient to resume the task at any later time, possibly on a different CPU core.

7.3 Performance Impact of Context Switching

Saving and restoring context is expensive relative to normal instruction execution because it:

- touches memory (often multiple cache lines)
- disrupts cache locality
- may require saving large register sets

A full context switch between tasks typically costs orders of magnitude more than a simple function call. The cost includes:

- saving the outgoing task state
- restoring the incoming task state
- switching address spaces
- reloading CPU execution context

Conceptual contrast:

```
# Function call (cheap)
call f
ret

# Context switch (expensive)
# save task A
# select task B
# restore task B
```

Because of this cost, operating systems are designed to:

- avoid unnecessary context switches
- keep fast paths short
- batch work inside the kernel when possible

This performance reality explains why syscalls are designed to be fast but not free, and why excessive syscall usage can dominate runtime in low-level code.

7.4 Where Context Switches Fit in the Syscall Path

A syscall does **not** automatically imply a context switch. Many syscalls complete entirely within the context of the calling thread.
Typical syscall path options:

- **Fast path**: syscall handled, return to user without switching tasks.
- **Slow path**: syscall blocks or triggers scheduling, causing a context switch.

Examples of syscalls that often stay on the fast path:

- simple queries (process ID, time read)
- operations satisfied from cache

Examples of syscalls that may cause a context switch:

- blocking I/O
- waiting on synchronization primitives
- resource contention

Conceptual flow:

```
# Syscall path (conceptual)
syscall
# kernel entry
# if fast path:
#   handle and return
# else:
#   save full context
#   schedule another task
#   later restore context
# return to user
```

For user-space developers, the key takeaway is that a syscall is a **potential suspension point**. Correct low-level code must assume that:

- execution may pause for an unbounded time
- another thread may run in between
- the CPU core may change before resumption

Understanding where and why context is saved and restored is essential for reasoning about syscall performance, signal interaction, and concurrency at the privilege boundary.

Chapter 8

Putting It All Together

8.1 Full Syscall Lifecycle Walkthrough

This section ties the previous chapters into a single end-to-end model of what happens when user code performs a Linux x86-64 system call.
A syscall lifecycle can be understood as four phases:

1. **User preparation**: place syscall number and arguments into the syscall ABI registers.
2. **Boundary transition**: execute `syscall`; CPU enters kernel mode and transfers control to the kernel entry point.
3. **Kernel service**: kernel validates inputs, performs the requested operation, and produces a result or error.
4. **Return to user**: kernel returns to user mode and resumes execution after the `syscall` instruction.

Example: `getpid()` in pure assembly (conceptual correctness: register protocol and return handling)

```
# long getpid(void) -> pid in rax
mov rax, 39           # __NR_getpid
syscall               # enter kernel, return to user
# rax = pid (>= 0) or -errno (< 0, rare for getpid)
```

Example: `write(1, msg, len)` with minimal error handling

```
# ssize_t write(int fd, const void* buf, size_t count)
# rax=__NR_write(1), rdi=fd, rsi=buf, rdx=count

lea rsi, [rel msg]
mov rdx, msg_len
mov rdi, 1
mov rax, 1            # __NR_write
syscall

# Success: rax = bytes written (>= 0)
# Failure: rax = -errno (< 0)
test rax, rax
jns .ok
neg rax               # rax = errno (positive)
# error handling here (rax holds errno)
.ok:

msg:
    .ascii "Syscall lifecycle: write()\n"
msg_len = . - msg
```

Key correctness rules reinforced by the walkthrough:

- `RAX` selects the service and receives the return value.

- Argument registers are `RDI, RSI, RDX, R10, R8, R9`.
- `RCX` and `R11` are clobbered by `syscall`.
- On failure, Linux returns `-errno` in `RAX`.

8.2 User → Kernel → User Execution Timeline

A syscall is a precise timeline of control and state changes. The most useful way to visualize it is as a sequence of checkpoints.

8.2.1 Timeline checkpoints (conceptual)

1. **T0 (User mode)**: user code is running normally.
2. **T1 (Prepare)**: user places syscall number and arguments in registers.
3. **T2 (Execute `syscall`)**: CPU switches to kernel mode and jumps to kernel entry.
4. **T3 (Entry)**: kernel establishes a trusted stack and saves return context.
5. **T4 (Dispatch)**: kernel identifies the requested syscall and validates inputs.
6. **T5 (Service)**: kernel performs the operation (fast or slow path).
7. **T6 (Return prep)**: kernel sets `RAX` to result or `-errno`.
8. **T7 (Return)**: kernel returns to user mode, resuming after `syscall`.

8.2.2 Timeline with fast vs slow path

```
# Fast path shape (conceptual):
# user -> syscall -> kernel handles quickly -> return

# Slow path shape (conceptual):
# user -> syscall -> kernel blocks or faults -> scheduler may run
↪  others
# later -> original thread resumes -> return to user
```

Why this matters:

- Even if the transition is fast, the work may not be.
- A syscall is a potential suspension point; the thread may sleep and resume later.
- Signals can be delivered near syscall boundaries and change the observed control flow.

8.3 Common Misconceptions About Syscalls

8.3.1 Misconception: "A syscall is just a function call"

Reality: a syscall crosses a privilege boundary. It is not a normal call:

- it enters Ring 0
- it uses different argument registers than the function-call ABI (Arg4 is `R10`, not `RCX`)
- it clobbers `RCX` and `R11`

```
# Common bug: using RCX as arg4 (wrong for syscalls)
mov rcx, 123          # WRONG: syscall arg4 is r10
mov r10, 123          # Correct
```

8.3.2 Misconception: “The syscall cannot be interrupted”

Reality: syscalls can be interrupted by signals, and blocking syscalls may sleep. User code must treat syscalls as boundary crossings with complex control flow possibilities.

8.3.3 Misconception: “Return value is always meaningful without error checks”

Reality: on Linux, errors are returned as negative `-errno`. Assembly must check sign.

```
syscall
test rax, rax
jns .ok
neg rax                # errno (positive)
# handle error
.ok:
```

8.3.4 Misconception: “The kernel trusts user pointers”

Reality: user pointers are untrusted inputs. The kernel validates and may fault during copy, returning errors (commonly manifested as negative `-errno`).

8.4 Why Understanding Syscalls Matters

Understanding syscalls is not optional for serious low-level work. It directly affects correctness, security reasoning, and performance analysis.

8.4.1 Correctness

- Correct register usage prevents silent corruption.

- Correct error handling prevents subtle failures.
- Correct assumptions about interruption avoid rare, hard-to-debug bugs.

8.4.2 Security

- Syscalls are privilege boundary crossings.
- All user inputs are treated as hostile by design.
- Many security failures originate from misunderstanding boundary invariants.

8.4.3 Performance

- Excess syscalls can dominate runtime due to entry/exit overhead and kernel work.
- “Fast syscall” does not mean “free syscall”.
- Understanding fast vs slow paths explains real-world latency variability.

8.4.4 Tooling and debugging

Reading traces, debugging crashes, and interpreting compiler output often requires knowing:

- when user code actually enters the kernel
- which registers carry arguments and results
- why the observed control flow differs from source-level expectations

Mastering syscalls and privilege boundaries completes the mental model of how user programs truly interact with the operating system on x86-64.

Appendices

Appendix A — Conceptual Syscall Flow Diagrams

User-to-Kernel Transition Overview

This appendix provides conceptual diagrams for the Linux x86-64 syscall path. The goal is to visualize the boundary crossing and the minimum architectural state transitions, without depending on any specific kernel implementation details.

Conceptual Syscall Flow (High-Level)

```
User Mode (Ring 3)
  |
  | 1) Prepare syscall ABI
  |    - RAX = syscall number
  |    - RDI, RSI, RDX, R10, R8, R9 = args
  |
  v
Execute SYSCALL
  |
  | 2) CPU-enforced transition
  |    - CPL: 3 -> 0
  |    - RIP: user -> kernel entry
  |    - RCX <- user RIP (return address)
  |    - R11 <- user RFLAGS
  |
  v
Kernel Entry Stub (Ring 0)
  |
  | 3) Establish trusted context
  |    - switch to kernel stack
  |    - save required registers/state
  |
  v
Syscall Dispatch + Handler
  |
  | 4) Validate + execute service
  |    - validate arguments, pointers
  |    - perform operation (fast/slow path)
  |    - set RAX = result or -errno
  |
  v
Kernel Return Path
  |
  | 5) Restore return context
  |    - restore user RSP (as needed)
  |    - prepare SYSRET using RCX/R11
  |
  v
User Mode (Ring 3)
  |
  | 6) Resume after SYSCALL
  |    - RAX holds return value
```

Fast Path vs Slow Path (Conceptual)

```
Fast path:
  SYSCALL -> quick validate -> execute -> SYSRET

Slow path examples:
  - blocking I/O (thread sleeps, scheduler runs others)
  - page fault during copy to/from user (fault handling)
  - signal interruption (early return / restart policy)
```

Register and Stack State Changes

This subsection focuses on the key architectural state changes that matter to user-space low-level code.

Register Roles at the Boundary (Linux x86-64 Syscall ABI)

```
On entry (user prepares):
  RAX = syscall number
  RDI = arg1
  RSI = arg2
  RDX = arg3
  R10 = arg4
  R8  = arg5
  R9  = arg6

Architectural effects of SYSCALL:
  RCX = user RIP of next instruction (return address)
  R11 = saved user RFLAGS
  RIP = kernel entry RIP (configured)
  CPL = 0 (kernel mode)
```

Clobbers Visible to User Code From the perspective of user-space assembly, two registers are always treated as clobbered by the `syscall` instruction itself:

```
Always clobbered by SYSCALL:
  RCX, R11
```

Therefore, user code must not assume these registers survive across a syscall.

Stack State Change (Conceptual) A syscall does not push a return address on the user stack. The return context is carried in registers.

```
Normal function call:
  call f
    - pushes return RIP onto current stack

Syscall:
  syscall
    - does NOT push return RIP onto user stack
    - stores return RIP in RCX instead
```

Inside the kernel, the entry stub typically switches to a kernel stack early. Conceptually:

```
Before SYSCALL:
  RSP -> user stack (untrusted)

After entering kernel:
  RSP -> kernel stack (trusted)
  (user RSP is saved in kernel state for later restore)
```

Concrete User-Side Demonstration (Register Hazards) This example demonstrates why `RCX` cannot be used as the 4th syscall argument and why `R10` is required.

```
# Incorrect syscall arg4 placement (RCX will be overwritten)
mov rax, 0              # syscall number (example only)
mov rcx, 123            # WRONG: syscall overwrites RCX
syscall                 # RCX clobbered
```

```
# Correct placement for arg4
mov rax, 0              # syscall number (example only)
mov r10, 123            # Correct: arg4 in R10
syscall
```

Return Value and Error Sign Convention Linux syscalls return:

- **Success:** `RAX` is non-negative
- **Failure:** `RAX` is negative (`-errno`)

```
# Generic syscall return check (conceptual)
syscall
test rax, rax
jns .ok
neg rax                 # rax = errno (positive)
# handle error in rax
.ok:
# success in rax
```

Appendix B — Common Errors and Dangerous Assumptions

This appendix lists the most common failure patterns seen when programmers first write syscall-level code or reason about user–kernel boundaries. Each item includes a concrete failure mode, why it happens, and a minimal correction pattern.

Assuming Syscalls Behave Like Functions

Error: treating `syscall` like `call` A function call is a user-space convention: it assumes shared ABI rules, no privilege switch, and it pushes a return address on the current stack. A syscall is a hardware-enforced boundary transition and does **not** behave like a normal call.

- **Function call:** pushes return address to stack, uses SysV function ABI registers.
- **Syscall:** saves return address in `RCX`, saves flags in `R11`, enters Ring 0, may switch stacks, may block, may be interrupted.

Symptom patterns

- using function-call argument registers for syscalls (especially Arg4)
- assuming caller/callee-saved rules apply across `syscall`
- assuming control returns immediately after `syscall` with no interruption

Concrete pitfall: Arg4 register mismatch

```
# WRONG: using RCX as the 4th argument like a normal function call
# SysV function ABI uses RCX for arg4, but SYSCALL overwrites RCX.
mov rax, 0              # syscall number (example only)
mov rdi, 1              # arg1
mov rsi, 2              # arg2
mov rdx, 3              # arg3
mov rcx, 4              # WRONG: RCX is clobbered by SYSCALL
syscall

# RIGHT: syscall ABI uses R10 for arg4
mov rax, 0
mov rdi, 1
mov rsi, 2
mov rdx, 3
mov r10, 4              # Correct arg4 register for syscalls
syscall
```

Correction checklist

- Use syscall ABI registers: `RDI, RSI, RDX, R10, R8, R9`.
- Treat `RCX` and `R11` as clobbered by `syscall`.
- Treat syscall as a potential suspension point (may block or be interrupted).

Error: ignoring Linux syscall error convention Linux syscalls return errors as **negative `-errno`** in `RAX`. This differs from libc wrappers that return `-1` and set `errno`.

```
# Correct sign-check for a raw syscall return
syscall
test rax, rax
jns .ok
neg rax                # rax = errno (positive)
# handle error (rax holds errno)
.ok:
# handle success (rax holds result)
```

Misunderstanding Privilege Boundaries

Error: assuming user code can "do kernel things" directly In user mode, privileged instructions and privileged state are blocked by the CPU. Attempting to execute privileged operations triggers faults, not "partial success".

```
# Privileged instruction in user mode -> fault (exception path)
cli                      # cannot be executed in Ring 3
```

Error: assuming kernel uses the user stack On entry, the kernel must run on a trusted stack. User `RSP` is untrusted and may be invalid or maliciously crafted. Correct kernel entry paths switch to a kernel stack early.
User-space implication:

- do not assume anything about kernel stack layout from user space
- do not assume `RSP` is meaningful to the kernel beyond being saved/validated

Error: assuming pointers are trusted User pointers passed to syscalls are untrusted. The kernel validates access and may fail with an error (commonly a negative `-errno`) if the pointer is invalid or faults during copy.

```
# Example of an invalid pointer passed to write
mov rax, 1          # __NR_write
mov rdi, 1          # fd = stdout
mov rsi, 1          # invalid user pointer (likely unmapped)
mov rdx, 16         # count
syscall
# Expect rax < 0 on failure (e.g., -EFAULT)
```

Security consequence The user–kernel boundary is a major attack surface. Correct reasoning requires assuming:

- user data is hostile until validated
- boundary transitions are controlled and audited
- unexpected events (signals/interrupts) can occur at boundaries

Ignoring ABI Register Rules

Error: mixing SysV function ABI and Linux syscall ABI These conventions are similar but **not identical**. Mixing them creates silent corruption.
Key differences that frequently break code:

- SysV function ABI arg4 is `RCX`; syscall arg4 is `R10`.
- `syscall` clobbers `RCX` and `R11`.
- Syscalls return `-errno` in `RAX` on failure.

Error: assuming caller-saved/callee-saved rules apply across syscalls Function-call ABIs define caller-saved/callee-saved contracts. A syscall crosses into kernel code and should be treated as:

- clobbering at least `RCX` and `R11` by definition
- potentially modifying other registers indirectly depending on calling environment
- always modifying flags in ways user code must not depend on

Robust syscall wrapper pattern (save what you need) If your user-space assembly needs to preserve values across syscalls, save them explicitly.

```
# Preserve a value across syscall (example technique)
push rbx                      # save caller state
mov rbx, 12345                # value you need later

mov rax, 39                   # __NR_getpid
syscall                       # RCX and R11 clobbered; do not depend on flags
```

```
# rax holds pid or -errno
pop rbx                  # restore saved state
```

Error: depending on flags after syscall Because the boundary transition modifies flags (and the kernel work itself is arbitrary), do not use condition codes from before `syscall` to make decisions after return.

```
# WRONG idea (do not do this):
# cmp rdi, rsi
# syscall
# jg .greater            # condition codes are not preserved
 ↪  meaningfully

# RIGHT idea:
syscall
test rax, rax            # set flags based on return value you actually
 ↪  need
jns .ok
```

Minimal ABI sanity checklist

- Load syscall number into `RAX`.
- Load args into `RDI, RSI, RDX, R10, R8, R9`.
- Assume `RCX` and `R11` are clobbered.
- Check `RAX` for negative `-errno`.
- Do not depend on flags across the syscall boundary.

Appendix C — Preparation for Next Booklets

This appendix defines the minimum readiness checklist to move from "syscalls and privilege boundaries" into deeper operating-system mechanics. The goal is not to teach the next topics fully, but to ensure the reader has the correct mental model, vocabulary, and low-level habits required to avoid common misconceptions.

Readiness for Interrupts and Exceptions

To be ready for interrupts and exceptions, you must be able to distinguish **why** control entered the kernel and what that implies about program correctness.

Core distinctions to master

- **Syscall** is intentional: user requests a service with `syscall`.
- **Exception** is synchronous: triggered by the current instruction (fault/trap).
- **Interrupt** is asynchronous: arrives independent of the current instruction.

Minimal mental model

- Exceptions have an immediate cause tied to an instruction (e.g., invalid opcode, divide error, page fault).
- Interrupts arrive between instructions and typically represent external/time events.
- Both lead to kernel entry with strict state saving and return rules.

Practical readiness exercises

1. Identify whether an event is synchronous or asynchronous from a short trace.
2. Explain what state must be preserved to resume user execution safely.
3. Explain why a user-mode privileged instruction must trap instead of “partially working”.

Concept-only trigger examples

```
# Synchronous exception example (user mode)
ud2                   # guaranteed invalid instruction -> exception
 ↪  path

# Synchronous exception example (user mode)
cli                   # privileged -> general-protection style fault
 ↪  path
```

```
# Asynchronous interrupt example (conceptual)
# Timer interrupt can arrive between any two instructions:
nop
nop
nop
```

Readiness checklist

- You can explain trap vs fault at a conceptual level (resume semantics differ).
- You can explain why page faults are “part of normal execution” in demand-paged systems.
- You can explain why returning to user mode requires validated instruction pointers and flags.

Readiness for Scheduling and Preemption

Scheduling and preemption require a correct understanding of **when a thread can stop running** and what the kernel must preserve so it can resume later.

Core concepts to master

- **Blocking vs non-blocking**: a syscall may block (sleep) and resume later.
- **Preemption**: the kernel may involuntarily stop a running thread to run another.
- **Context switch**: saving one task's context and restoring another's.

Key mental model A syscall boundary is a **potential scheduling point**. Even if the syscall itself is "fast", the kernel may need to wait for resources, causing the calling thread to sleep and another thread to run.

Conceptual syscall-to-scheduler flow

```
User:     syscall
Kernel:   validate -> attempt operation
          if resource not ready:
              save full context
              mark task sleeping
              pick next runnable task
          later:
              wake task
              restore context
              return to user
```

Practical readiness exercises

1. Explain why a blocking read can cause a context switch.

2. Explain why "time spent in syscall" may include time when the thread was not running.
3. Explain why user code must assume it might resume on a different CPU core.

Minimal assembly habit Assume any syscall can:

- be interrupted by signals
- block and later resume
- run after other threads have executed

```
# Robust syscall mindset (conceptual):
# treat syscall as a boundary with possible delays/interruptions
syscall
test rax, rax
jns .ok
neg rax                # errno
# handle error / retry policy
.ok:
```

Readiness for Kernel Internals

Kernel internals require understanding the core invariants of privileged code:

- kernel operates on untrusted user inputs
- kernel maintains per-task and per-CPU state
- kernel must preserve return context precisely
- kernel must not trust user stack or user pointers

Minimum internal structures to be ready to discuss (conceptual)

- **Task/thread structure**: holds saved registers, scheduling state, and kernel stack reference.
- **Kernel stack**: per-task (or per-CPU) trusted stack used during privileged execution.
- **Syscall dispatch table**: mapping syscall numbers to kernel handlers.
- **User memory access helpers**: mechanisms to validate/copy user buffers safely.

Critical invariants

- User pointers are validated and may fault during copy.
- Kernel return paths must restore user mode safely and prevent privilege confusion.
- The syscall ABI is stable at the boundary even if the kernel implementation evolves.

Readiness exercises

1. Explain why `RCX` and `R11` are special at the syscall boundary.
2. Explain why syscall Arg4 is `R10` (and what would break if you used `RCX`).
3. Explain why the kernel switches stacks and how this protects the system.

One-page readiness summary If you can correctly reason about:

- the difference between syscalls, exceptions, and interrupts
- the conditions that trigger a context switch
- the user–kernel trust boundary (stacks, pointers, privilege)

- the syscall ABI register and return conventions

then you are ready to move into deeper kernel entry paths, interrupt/exception handling, and scheduling internals in the next booklets.

References

x86-64 Architecture and Privilege Model Manuals

This booklet relies on the architectural definition of privilege levels, control transfers, and execution state as defined by the x86-64 architecture. Core concepts derived from these manuals include:

- privilege rings and Current Privilege Level (CPL)
- architectural effects of `SYSCALL` and `SYSRET`
- control registers, flags masking, and canonical addressing rules
- separation of architectural guarantees from microarchitectural behavior

These manuals define what the CPU *must* do during a privilege transition, independent of any operating system.

Linux Kernel Documentation (Syscalls & Entry Paths)

Linux kernel documentation provides the authoritative description of how the kernel uses the architectural mechanisms:

- syscall entry stubs and dispatch logic

- validation of user pointers and arguments
- kernel stack usage and per-task context handling
- fast path vs slow path behavior

The emphasis is on kernel-visible behavior rather than user-space abstractions. This includes how syscalls interact with scheduling, signals, and memory management.

System Call ABI and Calling Convention Specifications

The Linux x86-64 syscall ABI defines the stable user–kernel interface:

- syscall number placement in `RAX`
- argument registers `RDI, RSI, RDX, R10, R8, R9`
- return values and negative `-errno` convention
- architectural clobbers (`RCX`, `R11`)

These rules are contractual: user-space code must obey them exactly, regardless of kernel version or internal implementation changes.

Signals, Exceptions, and Process Control References

This booklet's treatment of signals and exceptions is grounded in the execution model defined by:

- synchronous exceptions triggered by instructions
- asynchronous events delivered independently of instruction flow

- conversion of faults into user-visible signals
- interaction between signal delivery and syscall return paths

The focus is on control flow effects and context preservation, not high-level signal APIs.

Compiler-Generated Syscall Code Behavior

Modern compilers generate syscalls indirectly through runtime libraries or inline sequences. This booklet references:

- register setup patterns for syscall wrappers
- error handling transformations performed by standard libraries
- differences between raw syscalls and library-mediated calls

Understanding compiler-generated behavior is essential for correctly reading disassembly and debugging optimized binaries.

Academic and Professional OS Architecture Materials

Concepts such as context switching, scheduling, and privilege separation are grounded in established operating system theory:

- process and thread models
- kernel/user isolation principles
- cost models for context switches and boundary crossings

These materials provide the theoretical framework that explains why modern kernels are structured around strict privilege boundaries.

Cross-References to Other Booklets in This Series

This booklet builds directly on earlier titles in the CPU Programming Series and prepares for subsequent ones:

- earlier booklets: registers, flags, memory, stack discipline, and calling conventions
- this booklet: completes the user-space execution model up to the kernel boundary
- upcoming booklets: interrupts, exceptions, scheduling, and kernel internals

Readers are expected to use this booklet as the transition point between pure user-space execution and full operating-system internals.