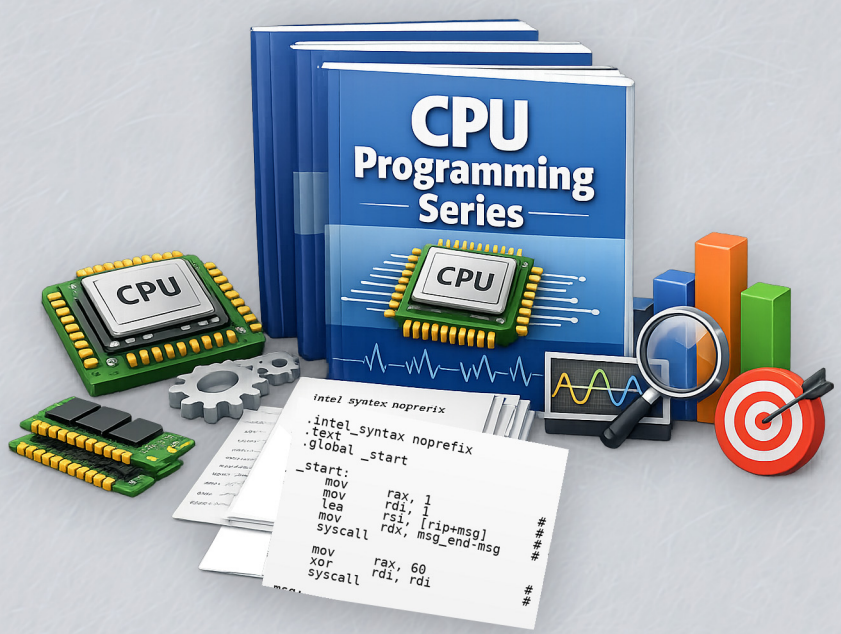
https://simplifycpp.org

CPU Programming Series

AArch64 Exceptions & Syscalls

EL0–EL3 Explained for Programmers



14

Prepared by Ayman Alheraki

CPU Programming Series

AArch64 Exceptions & Syscalls

EL0–EL3 Explained for Programmers

Prepared by Ayman Alheraki

simplifycpp.org

January 2026

Contents

Preface

Purpose of This Booklet

This booklet explains the AArch64 exception model as it is actually experienced by programmers: how control moves between **EL0 (user)** and **EL1 (kernel)** during syscalls, and why faults (page faults, permission faults, alignment faults, illegal instructions) show up as **synchronous exceptions**. It also gives a clear conceptual map of **EL2 (hypervisor)** and **EL3 (secure monitor)** so you can understand what they are, what they are not, and when your code interacts with them indirectly.
The goals are practical:

- Build a correct mental model of **exception entry/return**: vector selection, saved state, and return semantics.
- Teach you to **diagnose crashes and traps** using the key architectural state: `ESR_ELx`, `FAR_ELx`, `ELR_ELx`, `SPSR_ELx`.
- Show how a Linux syscall is **not a normal function call**: it is an exception path with ABI rules, privilege change, and controlled return.
- Explain where **virtualization (EL2)** and **secure world (EL3)** sit, without turning the booklet into a firmware manual.

Core promise. After finishing, you should be able to look at an AArch64 crash report, disassembly, or kernel trace and say: *which exception happened, at which EL, why it happened, what architectural state proves it, and what the correct fix is*.

Target Audience and Prerequisites

This booklet is for:

- C/C++ systems programmers doing Linux user-space work who need to understand syscalls, signals, and faults.
- Low-level engineers reading disassembly, debugging hard crashes, or doing performance-sensitive work near the kernel boundary.
- Embedded and platform engineers who must reason about EL1/EL2/EL3 even when application code runs at EL0.

Prerequisites (assumed knowledge):

- Basic AArch64 register literacy: `X0--X30`, `SP`, `PC`, and the idea of **PSTATE**.
- Comfort reading short assembly snippets and recognizing `BL`, `RET`, `MOV`, `LDR/STR`.
- A working concept of user/kernel separation (even if you have never read kernel code).

Not required: You do not need prior firmware experience, virtualization expertise, or a full Linux kernel tour. When such topics appear, they are introduced only to the extent that they affect the programmer's model.

How This Booklet Fits into the CPU Programming Series

This booklet is positioned where many programmers hit a wall: **the first time "normal code" becomes an exception path**. It connects the dots between architectural state, ABI discipline, and OS behavior.

- **Builds on**: AArch64 core registers and addressing discipline (the foundation needed to read exception entry/return code).
- **Extends**: calling convention knowledge into the syscall boundary (syscalls look like calls, but they are not).
- **Prepares you for**: virtualization-aware tracing (EL2) and secure monitor interfaces (EL3) in later booklets.

Series navigation. If you already know how AArch64 load/store and stack discipline work, you will move faster. If not, skim the earlier register and ABI booklets first; exceptions assume that baseline.

Conceptual vs Practical Scope

This booklet deliberately separates **architectural invariants** from **OS-specific implementation details**.

What is treated as conceptual (portable across systems)

- The definition of **exception levels** (EL0–EL3) and why transitions occur.
- The meaning of **synchronous exception** vs **IRQ/FIQ** vs **SError**.
- The role of **exception vectors** and the general entry/return mechanism.

- The meaning of the key system registers: `ESR_ELx`, `FAR_ELx`, `ELR_ELx`, `SPSR_ELx`.

What is treated as practical (Linux-centric, programmer-facing)

- The AArch64 syscall ABI as used from EL0 (register conventions, return values, and error handling).
- How Linux turns specific faults into **signals** and how that appears to user programs.
- How to **debug** an exception from user space and interpret evidence from disassembly and register state.

What is explicitly out of scope (kept minimal)

- Full EL3 firmware design, secure boot chains, and complete TrustZone runtime architecture.
- Hypervisor implementation details beyond the minimum needed to understand EL2's place in the model.
- Writing a full kernel exception subsystem from scratch.

Rule of thumb. If a detail does not change your ability to *reason about program behavior, debug faults, or understand syscall flow*, it is either summarized or deferred to later booklets.

Preview Examples: What You Will Be Able to Explain and Debug

Example 1 — Syscall: EL0 → EL1 → EL0 (Not a Normal Call)

A Linux syscall is invoked with `SVC #0`. Arguments are placed in registers and the kernel returns with a value in `X0`. This looks like a call/return from the programmer's point of view, but architecturally it is an exception entry/return path.

```c
/* Minimal syscall-style wrapper concept (ABI idea, not a full libc
↪  replacement). */
long my_getpid(void);
```

```
/* AArch64 GAS syntax (GNU as). Comments use C-style markers. */
.text
.global my_getpid
.type my_getpid, %function
my_getpid:
    /* Place syscall number in x8 (Linux AArch64 convention). */
    mov     x8, #172          /* __NR_getpid */
    /* SVC triggers a synchronous exception from EL0 to EL1. */
    svc     #0
    /* Kernel returns to EL0; result is in x0. */
    ret
```

What you will learn to answer:

- Why `svc #0` is classified as a synchronous exception.
- Which state is saved/restored on entry/return, and why `eret` is the architectural return mechanism.
- Why syscall conventions are strict (register usage, clobbers, error reporting).

Example 2 — Fault: Null Pointer Load ⇒ Synchronous Exception ⇒ Signal

A bad memory access in EL0 does not "just crash": it triggers a synchronous exception that the OS turns into a signal.

```c
/* Intentional fault: dereference a null pointer. */
static volatile int *p = (int*)0;
int main(void) { return *p; }
```

```asm
/* Representative pattern in generated code (illustrative). */
.text
.global _start
_start:
    /* x0 = 0; attempt to load from [x0] => translation fault in EL0.
     ↪  */
    mov     x0, #0
    ldr     w1, [x0]          /* triggers a synchronous exception */
    /* Unreached */
    mov     x8, #93           /* __NR_exit */
    svc     #0
```

What you will learn to answer:

- How the CPU classifies the fault and records evidence in `ESR_EL1` and `FAR_EL1`.
- Why the fault enters EL1 (kernel), and how the kernel decides to send `SIGSEGV`.
- How to distinguish a page fault from a permission fault, using the exception syndrome fields.

Example 3 — Illegal Instruction vs Privileged Instruction

Some failures are about **what** you executed, not **where** you accessed. Trying to execute a privileged operation at EL0 triggers a trap/fault that is diagnosed differently from a memory fault.

```
/* The exact instruction choice is illustrative: the point is
 ↪  privilege violation at EL0. */
.text
.global demo_priv_violation
.type demo_priv_violation, %function
demo_priv_violation:
    /* Attempt to access a privileged system register from EL0
     ↪  (conceptual example). */
    /* The architecture will trap and record syndrome in ESR_EL1. */
    mrs     x0, sctlr_el1      /* illegal at EL0: traps */
    ret
```

What you will learn to answer:

- How to interpret the syndrome to separate "undefined instruction" from "privileged access trap".
- Why the *same* instruction may be legal at EL1 but illegal at EL0.

Example 4 — Where EL2 and EL3 Appear in a Programmer's Life

Even if your application runs at EL0, you may still see EL2/EL3 indirectly:

- EL2 can affect timing, trapping behavior, and virtualization-related exits.
- EL3 governs secure monitor transitions (e.g., platform services), often invisible until debugging platform issues.

You will not implement a hypervisor or secure monitor here, but you will gain the correct map: *what belongs to EL1 vs EL2 vs EL3, and what evidence indicates their involvement.*

Chapter 1

AArch64 Exception Model Overview

1.1 What Is an Exception in ARM?

In AArch64, an **exception** is any architecturally-defined event that **diverts control flow** from the current instruction stream to a privileged handler at the **same or higher Exception Level (EL)**. The key idea is simple:

- Normal code flow: `PC` advances sequentially through instructions.
- Exception flow: hardware redirects execution to a **vector entry** and provides enough saved state for software to determine *why* the diversion happened and how to continue (or terminate).

Exceptions are the unifying mechanism behind:

- System calls (`SVC`)
- Debug traps / intentional breakpoints (`BRK`)
- Memory faults (instruction/data aborts, permission faults, translation faults)

- Interrupt delivery (IRQ/FIQ)
- Machine-check style errors (SError)
- Controlled traps used by virtualization (`HVC`) and secure monitor transitions (`SMC`)

Programmer's mental model

Treat an exception like a **hardware-enforced control transfer** with a **structured evidence package**: `ELR_ELx` (where it happened), `SPSR_ELx` (saved state), `ESR_ELx` (what kind of exception), and sometimes `FAR_ELx` (fault address).

1.2 Synchronous vs Asynchronous Exceptions

ARM separates exceptions into two practical timing classes that matter for debugging and design.

1.2.1 Synchronous exceptions (precise, instruction-related)

A **synchronous** exception is **caused by executing a specific instruction** or by an instruction's required memory access. It is architecturally **precise**: the exception is associated with a specific point in the instruction stream.
Common synchronous sources in EL0 programs:

- **Supervisor call**: `SVC #imm` (syscall entry)
- **Breakpoint**: `BRK #imm` (debug/intentional trap)
- **Instruction abort**: fetch fault (execute permission, translation fault, etc.)
- **Data abort**: load/store fault (translation, permission, alignment, etc.)

- **Illegal/undefined instruction** or **privileged instruction at EL0**
- **Trapped system register access** (when configured to trap)

1.2.2 Asynchronous exceptions (not tied to the current instruction)

An **asynchronous** exception is **not caused by the instruction currently executing**. It is delivered due to an external or deferred event. Two common families:

- **Interrupts**: **IRQ** and **FIQ** (external devices, timers, IPIs)
- **SError**: asynchronous system error signaling (implementation/platform dependent behavior)

Why this matters

- With synchronous exceptions, your first question is: *Which instruction caused it?*
- With interrupts, your first question is: *What event arrived, and why was it enabled/unmasked now?*
- With SError, your first question is: *What part of the system signaled an error, and how does the platform report it?*

1.3 Exceptions vs Interrupts vs Faults

The terminology is often overloaded, so this booklet uses a strict hierarchy:

- **Exception** (umbrella): any event that diverts control to an exception vector.
- **Interrupt**: an exception delivered from an external/async source (IRQ/FIQ).

- **Fault/Abort**: a synchronous exception triggered by instruction fetch or data access problems.

1.3.1 Faults (as programmers encounter them)

In AArch64 software practice, the most important fault families are:

- **Instruction abort**: fault when fetching an instruction (e.g., execute permission, translation failure).
- **Data abort**: fault when performing a load/store (e.g., unmapped page, permission fault).

A key observation for debugging:

- A syscall is an exception that you **asked for**.
- A fault is an exception that your program **provoked by violating a rule** (memory, privilege, alignment, etc.).
- An interrupt is an exception that the **outside world delivered to you**.

Example: three different reasons for leaving EL0

```
/* AArch64 GAS syntax; comments use C-style markers. */
.text
.global demo_three_exits
.type demo_three_exits, %function
demo_three_exits:
    /* (1) Intentional: syscall entry (synchronous exception) */
    mov     x8, #172          /* __NR_getpid (Linux AArch64) */
    svc     #0                /* EL0 -> EL1 -> EL0 */
```

```
    /* (2) Intentional: debug trap (synchronous exception) */
    brk     #0                  /* Debug exception if enabled/handled
     ↪  */

    /* (3) Unintentional: data abort (synchronous fault) */
    mov     x0, #0
    ldr     x1, [x0]            /* Likely translation fault -> EL1
     ↪  handler */

    ret
```

What changes across (1), (2), and (3) is not "did we take an exception?" (yes in all), but **what the syndrome reports** and **how the OS chooses to respond**.

1.4 Exception Entry and Return (High-Level View)

Exception entry/return is a hardware-defined protocol. Your code does not *invent* the rules; it follows them.

1.4.1 High-level entry sequence

When an exception is taken to a target EL (typically EL1 for OS kernels), hardware performs these conceptual steps:

1. **Select the vector base** for the target EL (conceptually: a table of entry points).
2. **Record the return address** into `ELR_ELx` (where execution should resume).
3. **Save the prior state** into `SPSR_ELx` (saved PSTATE, masks, mode bits, etc.).

4. **Populate syndrome info** (for many exceptions) into `ESR_ELx`; for address faults also set `FAR_ELx`.

5. **Update execution state** to the target EL (privilege level changes; interrupt masks may be applied).

6. **Branch to the vector entry** corresponding to the exception class (sync/IRQ/FIQ/SError and context).

The result: handler code starts executing at a privileged vector entry with architectural state that explains *why* it was entered.

1.4.2 High-level return sequence

Returning from an exception is performed via `ERET` (exception return):

- `ERET` restores PSTATE from `SPSR_ELx`
- `ERET` restores the next `PC` from `ELR_ELx`
- control resumes in the previous context (often EL0 user-space) *as if the exception was a controlled detour*

Conceptual sketch: syscall path

```
/* Conceptual flow only: illustrates the architectural "entry/return"
 ↪  contract. */
.text

/* EL0 user code */
.global user_syscall_example
```

```
.type user_syscall_example, %function
user_syscall_example:
    mov     x8, #64           /* __NR_write (example) */
    /* x0=fd, x1=buf, x2=len would be set by caller */
    svc     #0                /* synchronous exception to EL1 */
    ret

/* EL1 kernel-side vector handler (conceptual, not real Linux code)
 ↪  */
.global el1_sync_vector_stub
.type el1_sync_vector_stub, %function
el1_sync_vector_stub:
    /* On entry:
       - ELR_EL1 holds return address into EL0
       - SPSR_EL1 holds saved PSTATE from EL0
       - ESR_EL1 describes the exception class (e.g., SVC from EL0)
    */
    /* Save volatile registers to kernel stack (policy-defined) */
    /* Decode ESR_EL1 to route: syscall vs fault vs other */
    /* Execute service */
    /* Place return value in x0 (ABI policy for syscall return) */
    eret                    /* return to EL0 at ELR_EL1 with state
     ↪  from SPSR_EL1 */
```

1.4.3 Conceptual sketch: data abort evidence

A data abort handler typically relies on:

- `ESR_EL1`: tells you *what kind* of abort and key attributes.
- `FAR_EL1`: gives the *faulting virtual address* for address-related faults.

- `ELR_EL1`: points to the instruction address where the fault occurred.

```
/* Minimal evidence-gathering sketch (conceptual). */
.text
.global el1_data_abort_stub
.type el1_data_abort_stub, %function
el1_data_abort_stub:
    mrs     x0, esr_el1         /* syndrome: class + details */
    mrs     x1, far_el1         /* fault address (when applicable) */
    mrs     x2, elr_el1         /* where it happened */
    mrs     x3, spsr_el1        /* saved state of the faulting context
     ↪  */
    /* Now: decide whether to fix up (page-in), signal/kill, or
     ↪  panic. */
    eret
```

Key takeaway for programmers

- Exceptions are **control-flow transfers with proof**.
- Synchronous exceptions are **about a specific instruction**.
- Asynchronous exceptions are **about an arriving event**.
- Entry/return is **not optional**: vector selection, saved state, syndrome, and `ERET` form the contract.

Chapter 2

Exception Levels (EL0–EL3)

2.1 Overview of Privilege Levels

AArch64 defines four **Exception Levels (ELs)** that represent increasing privilege. The EL model answers one core question:

> *Which code is allowed to control the machine, and which code must request services through controlled entry points?*

Privilege is not a moral ranking; it is a **safety boundary**:

- Higher ELs can control memory translation, interrupt routing, and privileged system registers.
- Lower ELs run with restrictions and must use **exceptions** (e.g., `SVC`, traps, faults) to cross the boundary.

What changes as EL increases

From a programmer's perspective, higher EL typically implies access to:

- privileged system registers and control bits (MMU enable, caches, translation regime selection)
- exception vector configuration and routing policy
- interrupt masking, prioritization, and controller programming
- memory translation tables and page permissions

Two orthogonal axes you must not mix

- **Privilege axis**: EL0 → EL3 (who is allowed to do what).
- **Security axis**: Secure vs Non-secure state (who belongs to which world).

EL0–EL2 are normally in **Non-secure** state on general-purpose OS systems; EL3 is the **secure monitor** boundary manager.

2.2 EL0: User Space

EL0 is where normal applications run: C/C++ programs, runtimes, language VMs, services, and user-level libraries.

What EL0 can do well

- Execute unprivileged instructions at full speed
- Use the virtual memory view provided by the OS
- Request OS services via syscalls (typically `SVC #0` on Linux)

What EL0 cannot do

- Directly program MMU/translation tables, exception vectors, interrupt controllers
- Access many privileged system registers
- Directly manage device registers mapped as privileged

Programmer-facing view: EL0 exits

In practice, EL0 leaves normal flow through:

- **Syscalls** (intentional): `SVC`
- **Faults** (unintentional): aborts on illegal memory access or privilege violations
- **Signals/async events** (delivered via kernel mechanisms)

```
/* EL0 user-space: invoke a syscall by raising a synchronous
 ↪  exception. */
.text
.global el0_syscall_getpid
.type el0_syscall_getpid, %function
el0_syscall_getpid:
    mov     x8, #172          /* __NR_getpid on Linux AArch64 */
    svc     #0                /* EL0 -> EL1 synchronous exception */
    ret                       /* return value in x0 */
```

2.3 EL1: Kernel Space

EL1 is typically where the OS kernel runs. It is responsible for:

- defining the process virtual address spaces (translation tables, permissions)
- handling exceptions from EL0 (syscalls, faults, debug traps)
- scheduling, signals, process control, and isolation
- controlling device access (directly or via drivers)

Why EL1 exists (programmer translation)

EL1 is the layer that makes EL0 safe and productive:

- You can crash your process; you should not crash the machine.
- You can request I/O; you should not directly program arbitrary devices.
- You can allocate memory; you should not rewrite translation tables.

What EL1 does on EL0 entry

Conceptually, on an EL0 synchronous exception:

- kernel vectors receive control (EL1 vector table)
- saved state is available via `ELR_EL1` and `SPSR_EL1`
- the cause is classified via `ESR_EL1` (and `FAR_EL1` for address faults)
- kernel decides: perform syscall, fix up fault, deliver signal, or terminate

```
/* EL1 conceptual handler stub: read evidence then route. Not real
↪  Linux code. */
.text
.global el1_sync_entry_concept
```

```
.type el1_sync_entry_concept, %function
el1_sync_entry_concept:
    mrs     x0, esr_el1         /* what happened */
    mrs     x1, elr_el1         /* where it happened */
    mrs     x2, spsr_el1        /* saved state */
    /* If data/instruction abort: FAR_EL1 is meaningful */
    mrs     x3, far_el1
    /* Route to: syscall dispatcher / fault handler / debug handler
     ↪  */
    eret
```

2.4 EL2: Hypervisor

EL2 is the **virtualization privilege level**. When virtualization is used, EL2:

- hosts one or more guest kernels (often running at EL1 in a virtualized context)
- traps and emulates privileged operations performed by guests
- controls stage-2 translation (guest physical → host physical mapping)

Programmer-facing reality of EL2

Many systems run without a hypervisor; in that case EL2 may be unused or minimally configured. When present, EL2 changes how you reason about **who owns the machine**:

- the guest kernel believes it is in charge, but EL2 can intercept sensitive actions
- certain exceptions can be routed to EL2 (configured by the hypervisor)
- performance and timing behavior can differ due to traps/VM exits

```
/* Conceptual: hypercall from a guest context (illustrative). */
.text
.global guest_hypercall_example
.type guest_hypercall_example, %function
guest_hypercall_example:
    /* HVC triggers a synchronous exception; target typically EL2 if
     ↪  enabled. */
    hvc     #0
    ret
```

What you should remember

EL2 is about **controlling a guest** with strong isolation. If you are debugging odd kernel behavior in a VM, always consider whether a trap to EL2 is involved.

2.5 EL3: Secure Monitor

EL3 is the **secure monitor** level. It is the control point for transitions between:

- **Secure state** (trusted services, secure firmware)
- **Non-secure state** (normal OS world: EL0/EL1/EL2)

What EL3 typically owns

- secure boot chain coordination and early platform initialization
- secure world entry/exit policy and mediation
- handling secure monitor calls (`SMC`)

What EL3 is not (for this booklet)

EL3 is not your day-to-day syscall world. You do not call EL3 to open files. You usually encounter EL3 when:

- platform firmware provides services via SMC
- secure boot or security policy affects what EL1/EL2 can configure
- debugging early boot, trustzone services, or platform power management paths

```
/* Conceptual: secure monitor call (platform-specific meaning). */
.text
.global smc_example
.type smc_example, %function
smc_example:
    /* Arguments would be placed in x0..xN by a calling convention
     ↪  defined by firmware. */
    smc     #0
    ret
```

2.6 Typical EL Transitions in Real Systems

This section maps the abstract EL model to the flows programmers actually see.

2.6.1 Common flow on Linux without virtualization

- Application runs at **EL0**.
- Syscall uses `SVC`: **EL0** → **EL1**.
- Kernel handles request and returns with `ERET`: **EL1** → **EL0**.

```
/* EL0 -> EL1 -> EL0 syscall round-trip (illustrative). */
.text
.global write_syscall_roundtrip
.type write_syscall_roundtrip, %function
write_syscall_roundtrip:
    /* x0=fd, x1=buf, x2=len */
    mov     x8, #64           /* __NR_write (Linux AArch64) */
    svc     #0                /* enter EL1 */
    /* x0 holds return value; negative typically indicates an error
     ↪  convention */
    ret
```

2.6.2 Fault-driven transitions

A bad access in EL0 triggers a synchronous exception into EL1:

- EL0 executes a load/store or instruction fetch that violates translation/permissions.
- hardware takes a synchronous abort: **EL0** $\rightarrow$ **EL1**.
- kernel may fix up (e.g., page-in) and return, or deliver a signal / kill the process.

```
/* Typical EL0 mistake: load from an invalid pointer. */
.text
.global el0_fault_example
.type el0_fault_example, %function
el0_fault_example:
    mov     x0, #0
    ldr     x1, [x0]          /* likely data abort: EL0 -> EL1 */
    ret
```

2.6.3 With virtualization enabled

Common patterns:

- EL0 (guest user) uses `SVC`: guest EL0 $\rightarrow$ guest EL1.
- Guest EL1 may be trapped by EL2 on sensitive operations: guest EL1 $\rightarrow$ EL2.
- Hypervisor returns control back to the guest via **exception return mechanisms**.

2.6.4 Secure world interactions

Secure monitor calls are not part of normal syscalls, but may appear in:

- power management, trusted key storage, secure services
- platform-specific firmware interfaces

Flow conceptually: **Non-secure EL1/EL2 $\rightarrow$ EL3 $\rightarrow$ (secure services) $\rightarrow$ return**.

Discipline summary

- **Syscalls**: controlled EL0 $\leftrightarrow$ EL1 transition.
- **Faults**: rule violation triggers EL0 $\rightarrow$ EL1 for diagnosis and policy.
- **Virtualization**: EL2 may intercept guest privileged operations.
- **Security**: EL3 mediates secure/non-secure boundaries; mostly firmware-driven.

Chapter 3

Execution States and Security Context

3.1 AArch64 vs AArch32 Execution States

ARMv8 and later architectures support two architectural execution states:

- **AArch64**: 64-bit execution state (general-purpose registers `X0--X30`, 64-bit address model, A64 instruction set).
- **AArch32**: 32-bit execution state (general-purpose registers `R0--R15`, 32-bit address model, A32/T32 instruction sets).

What “execution state” means (strictly)

Execution state is not “a mode bit like user/kernel”. It determines:

- the instruction set being decoded and executed
- the register view and operand sizes
- the architectural rules for exceptions, system registers, and context saving

Modern OS reality

On contemporary 64-bit Linux and most modern mobile/embedded platforms:

- the kernel runs in **AArch64**
- user programs run in **AArch64**

AArch32 may exist for legacy compatibility on some systems, but it is not required for understanding the normal EL0↔EL1 syscall and fault model in this booklet.

Exception-state impact

An exception can change EL, but does not automatically imply a change of execution state. State transitions (AArch32 ↔ AArch64) are a configuration and boot-time policy matter.

Practical debugging clue

If you see `Xn` registers, `SP`, `PC` in 64-bit form and A64 mnemonics (`ldr x`, `stp x`, `eret`), you are in AArch64. If you see `Rn`, `SP`, `LR`, `PC` in 32-bit form, you are in AArch32.

```
/* AArch64: 64-bit register names and A64 instructions. */
.text
.global state_aarch64_example
.type state_aarch64_example, %function
state_aarch64_example:
    mov     x0, #1
    ldr     x1, [x2]
    ret
```

```
/* AArch32 syntax varies by toolchain; shown only as a conceptual
↪  contrast. */
```

```
/* The key concept: R0-R15 and different instruction encodings. */
.text
.global state_aarch32_concept
.type state_aarch32_concept, %function
state_aarch32_concept:
    /* mov r0, #1 */
    /* ldr r1, [r2] */
    /* bx  lr */
    ret
```

3.2 Secure vs Non-Secure World

The ARM security model introduces a **security state** orthogonal to privilege level:

- **Non-secure state**: the "normal world" where general-purpose OSes (Linux/Android) run.
- **Secure state**: the "secure world" intended for trusted code and services.

Orthogonality rule

Do not conflate:

- **Privilege (EL0–EL3)** with
- **Security state (Secure vs Non-secure)**

In typical systems:

- EL0/EL1 (and EL2 if used) operate in **Non-secure state**.
- EL3 is the **secure monitor** responsible for managing transitions between security states.

Why this matters to programmers

Even if your code never calls secure services directly, security state affects:

- which memory and peripherals are visible
- which system resources can be configured by the normal OS
- how certain platform services (keys, crypto engines, firmware interfaces) are accessed

Practical misconception to avoid

A secure world is not "a more privileged kernel". It is a **separate security domain** with its own isolation policy and service boundary, managed through controlled transitions.

3.3 TrustZone Conceptual Model

TrustZone is ARM's conceptual framework for **two-world isolation**:

- a normal world OS and applications (Non-secure state)
- a secure world runtime offering trusted services (Secure state)

Core TrustZone idea

The CPU and interconnect enforce a label (secure vs non-secure) that affects:

- memory regions (secure RAM vs normal RAM)
- device access (secure-only peripherals vs shared devices)
- interrupts (some interrupts can be routed as secure)

How code crosses the boundary

Cross-world transitions are performed through **secure monitor calls** using `SMC`. From a programmer's viewpoint, `SMC` is like:

> *A synchronous exception used as a portal into the secure monitor, which then dispatches to secure services.*

```
/* Conceptual: Non-secure world requests a secure service via SMC. */
.text
.global tz_smc_concept
.type tz_smc_concept, %function
tz_smc_concept:
    /* x0..xN carry a service identifier and arguments (platform
     ↪  ABI). */
    mov     x0, #0            /* service id (illustrative only) */
    smc     #0                /* enter EL3 secure monitor */
    /* return value typically in x0 (platform-defined convention) */
    ret
```

What is guaranteed vs platform-defined

- Guaranteed by architecture: `SMC` causes an exception-like entry to the monitor with defined state saving/return rules.
- Platform-defined: the service IDs, argument semantics, and which services exist.

Practical view in OS stacks

In many systems, user code does not issue `SMC` directly. Instead, the path is often:

- EL0 app requests a service via syscall or driver API

- EL1 kernel/driver issues `SMC` to firmware/secure runtime
- EL3 dispatches and returns

3.4 Role of EL3 in Secure Boot and Transitions

EL3 is the control point that:

- runs early in boot on many platforms
- configures initial security policies
- provides the monitor that handles Secure $\leftrightarrow$ Non-secure transitions

3.4.1 EL3 in secure boot (conceptual sequence)

A secure boot chain is a policy-driven process that aims to ensure only authenticated code runs at privileged layers. At a high level, EL3 typically participates by:

- establishing a **root of trust** and verifying subsequent boot stages
- configuring security partitions (which memory/peripherals are secure-only)
- handing off control to the normal-world firmware/OS at EL2 or EL1 (platform-dependent)

3.4.2 EL3 as the transition authority

After boot, EL3 remains the authority that:

- receives `SMC` requests and routes them
- enforces policy about what the normal world may request
- can switch context between secure and non-secure execution environments

What programmers should internalize

- EL3 is **not part of the syscall fast-path**. Syscalls are EL0→EL1.
- EL3 is a **platform security and firmware boundary**. You meet it through `SMC`-mediated services.
- The architectural mechanism is consistent; the service catalog is platform-specific.

Example: layered request path (common in practice)

```
/* Conceptual layering: EL0 -> EL1 syscall, then EL1 -> EL3 SMC. */
.text
.global layered_request_concept
.type layered_request_concept, %function
layered_request_concept:
    /* EL0: request a privileged operation via syscall */
    mov     x8, #0            /* syscall number placeholder */
    svc     #0                /* EL0 -> EL1 */

    /* In EL1 (kernel/driver), a secure service might be requested:
     ↪  */
    /* smc #0  (executed in EL1 context to enter EL3 monitor) */

    ret
```

Boundary discipline

TrustZone and EL3 exist to ensure:

- the normal OS cannot silently take ownership of secure assets

- secure services remain isolated even if the normal world is compromised

This booklet keeps the focus on what you need as a programmer: how to recognize when security context is involved, how transitions occur, and what is architecturally guaranteed.

Chapter 4

Exception Types in AArch64

4.1 Synchronous Exceptions

A **synchronous exception** is taken as a direct, architecturally precise consequence of executing an instruction or performing the memory access required by that instruction. In AArch64 practice, this category includes both **intentional control transfers** (syscalls, debug traps) and **unintentional faults** (aborts).

Synchronous exception families (programmer view)

- **Supervisor Call (SVC)**: controlled entry to the OS/service layer.
- **Hypervisor Call (HVC)**: controlled entry to EL2 (when virtualization is present).
- **Secure Monitor Call (SMC)**: controlled entry to EL3 (platform/firmware boundary).
- **Breakpoint / Debug trap**: `BRK` and related debug exceptions.
- **Instruction abort**: fault on instruction fetch (translation, permission, etc.).

- **Data abort**: fault on load/store (translation, permission, alignment, etc.).
- **Illegal/undefined instruction or trapped instruction**: execution violates architectural rules for the current context.

Evidence and diagnosis (what you will read)

For synchronous exceptions taken to EL1, the kernel/handler typically consults:

- `ESR_EL1`: syndrome (exception class + details)
- `ELR_EL1`: address of the faulting/triggering instruction (return PC)
- `SPSR_EL1`: saved state (PSTATE snapshot)
- `FAR_EL1`: fault address for address-related aborts

4.1.1 Example 1: syscall via SVC (intentional synchronous exception)

```
/* EL0: invoke a Linux syscall using SVC. */
.text
.global demo_svc_getpid
.type demo_svc_getpid, %function
demo_svc_getpid:
    mov     x8, #172          /* __NR_getpid (Linux AArch64) */
    svc     #0                /* synchronous exception: EL0 -> EL1 */
    /* return value in x0 */
    ret
```

4.1.2 Example 2: debug trap via BRK (intentional synchronous exception)

```
/* EL0: BRK triggers a debug exception when enabled/handled by the
↪  environment. */
```

```
.text
.global demo_brk
.type demo_brk, %function
demo_brk:
    brk     #0                  /* synchronous debug exception */
    ret
```

4.1.3 Example 3: data abort (unintentional synchronous fault)

```
/* EL0: invalid memory access typically triggers a data abort. */
.text
.global demo_data_abort
.type demo_data_abort, %function
demo_data_abort:
    mov     x0, #0
    ldr     x1, [x0]            /* synchronous fault: likely
     ↪  translation fault */
    ret
```

4.1.4 Example 4: illegal or privileged instruction at EL0

```
/* EL0: reading many privileged system registers is illegal and
 ↪  traps/faults. */
.text
.global demo_privileged_mrs
.type demo_privileged_mrs, %function
demo_privileged_mrs:
    mrs     x0, sctlr_el1       /* illegal at EL0: synchronous
     ↪  exception */
    ret
```

Common causes of synchronous exceptions

- explicit trap instructions: `SVC`, `HVC`, `SMC`, `BRK`
- page not present / unmapped VA (translation fault)
- access permissions (user/kernel, read/write/execute permission faults)
- alignment faults when the architecture/OS configuration requires alignment
- illegal opcode or instruction not permitted in the current execution context
- trapped system register accesses (policy-controlled)

4.2 IRQ and FIQ

IRQ and **FIQ** are **interrupt** exception types. They are usually **asynchronous**: they arrive due to external events (timers, devices, inter-processor interrupts), not because the current instruction is invalid.

4.2.1 IRQ (normal interrupt request)

IRQ is the general interrupt class used for most device and timer interrupts. The OS typically:

- receives the interrupt at a configured EL (commonly EL1 for a normal OS kernel)
- queries an interrupt controller to identify the source
- dispatches a device/timer handler
- returns to the interrupted context via `ERET`

4.2.2 FIQ (fast interrupt request)

FIQ is a distinct interrupt class intended for high-priority or latency-sensitive interrupts. Architecturally, it has separate routing and masking controls from IRQ.

Programmer-facing differences

- IRQ and FIQ are not “faults”; your code is not wrong because an interrupt arrived.
- IRQ/FIQ handling quality affects latency, responsiveness, and real-time behavior.
- In kernel debugging, an unexpected IRQ storm is a system condition, not a user bug.

4.2.3 Conceptual example: an interrupt arrives while EL0 runs

```
/* Conceptual: EL0 code can be interrupted by IRQ/FIQ at almost any
 ↪  time. */
.text
.global demo_interruptible_loop
.type demo_interruptible_loop, %function
demo_interruptible_loop:
    mov     x0, #0
1:
    add     x0, x0, #1
    /* An IRQ/FIQ may arrive here; control transfers to the
     ↪  configured vector. */
    b       1b
```

Common causes of IRQ/FIQ

- periodic timer interrupts (scheduler tick, high-resolution timers)

- device interrupts (network, storage, UART, GPU, etc.)
- inter-processor interrupts (IPIs) for scheduling and coordination
- performance monitoring or watchdog events (platform dependent)

4.3 SError (System Error)

SError is the AArch64 exception type used to signal **system errors**, often related to error reporting from the memory system, interconnect, or other implementation-defined sources.

Key properties

- SError may be **asynchronous** (reported at a later time than the originating event).
- The precise cause can be more platform-specific than a page fault or syscall.
- Many systems treat certain SErrors as fatal or as conditions requiring immediate containment.

Why SErrors are different from data aborts

A data abort is usually about a **virtual address translation/permission rule** being violated by a specific access. An SError is often about the **system fabric reporting an error condition** (e.g., parity/ECC/interconnect faults), and may not map cleanly to one user instruction.

4.3.1 Conceptual example: deferred error reporting

```
/* Conceptual sketch: the faulting event may occur earlier than when
 ↪  SError is delivered. */
.text
```

```
.global demo_serror_concept
.type demo_serror_concept, %function
demo_serror_concept:
    /* Program executes normal memory operations... */
    ldr     x1, [x0]
    str     x1, [x2]
    /* ...an internal system error may be reported asynchronously as
     ↪  an SError. */
    ret
```

Common causes of SError (system-level)

- memory subsystem error reporting (e.g., ECC/parity detection, implementation-defined handling)
- interconnect or bus fabric errors signaled to the CPU
- platform-specific external aborts that are not modeled as ordinary translation/permission faults
- error containment or RAS-driven reporting paths (platform dependent)

4.4 Common Causes of Each Exception Type (Quick Mapping)

This mapping is intentionally operational: it tells you what to suspect first.

Synchronous exceptions (first suspects)

- **SVC/HVC/SMC/BRK**: intentional trap instruction was executed

- **Data abort**: bad pointer, unmapped page, permission violation, alignment violation
- **Instruction abort**: jump to unmapped memory, execute-permission violation, corrupted function pointer
- **Illegal/trapped instruction**: wrong context/privilege, unsupported instruction, trapped system register access

IRQ/FIQ (first suspects)

- timer tick / high-resolution timers
- device interrupts and interrupt controller routing
- IPIs (multi-core scheduling and coordination)
- interrupt storms (misconfigured device/driver)

SError (first suspects)

- memory or interconnect error reporting (may be deferred)
- platform RAS signaling and containment policy
- external abort-like conditions not represented as normal translation/permission faults

Discipline note

When debugging, do not start with guesses. Start with the architectural evidence: `ESR_ELx` (class), `ELR_ELx` (where), `FAR_ELx` (address if relevant), and the execution context (EL and security state).

Chapter 5

Exception Vector Tables

5.1 Vector Table Structure

An AArch64 **exception vector table** is a fixed-layout block of code containing entry points for exception handling. When an exception is taken to a given Exception Level (EL), hardware selects a vector entry based on:

- **Exception type**: synchronous, IRQ, FIQ, or SError
- **Origin and stack context**: whether the exception came from the same EL or a lower EL, and which stack pointer was in use

Conceptual layout (AArch64)

The table provides **16 entries** arranged as:

- 4 exception types (Sync, IRQ, FIQ, SError)
- for each type, 4 origin/context cases:

- from current EL using `SP0`
- from current EL using `SPx`
- from lower EL using `AArch64`
- from lower EL using `AArch32` (if applicable)

What the vector entry must do

A vector entry is not the full handler. It is a **first-stage landing pad** that must quickly:

- establish a safe stack (if needed) and preserve the required registers
- read syndrome/state registers as needed (`ESR_ELx`, `FAR_ELx`, `ELR_ELx`, `SPSR_ELx`)
- branch to the appropriate higher-level handler (syscall, fault, interrupt dispatch, etc.)

Minimal conceptual skeleton

```
/* Conceptual vector table skeleton (not complete OS code). */
.text
.align  11                    /* typical 2KB alignment for a vector
↪  table base */
.global vectors_el1
vectors_el1:
    /* Each entry is placed at a fixed offset from the base. */
    /* Entry 0x000: Sync from current EL using SP0 */
    b       el1_sync_sp0
    /* Entry 0x080: IRQ  from current EL using SP0 */
    b       el1_irq_sp0
    /* Entry 0x100: FIQ  from current EL using SP0 */
    b       el1_fiq_sp0
```

```
    /* Entry 0x180: SError from current EL using SP0 */
    b       el1_serror_sp0

    /* Entry 0x200: Sync from current EL using SPx */
    b       el1_sync_spx
    /* Entry 0x280: IRQ from current EL using SPx */
    b       el1_irq_spx
    /* Entry 0x300: FIQ from current EL using SPx */
    b       el1_fiq_spx
    /* Entry 0x380: SError from current EL using SPx */
    b       el1_serror_spx

    /* Entry 0x400: Sync from lower EL using AArch64 */
    b       el1_sync_lower_a64
    /* Entry 0x480: IRQ from lower EL using AArch64 */
    b       el1_irq_lower_a64
    /* Entry 0x500: FIQ from lower EL using AArch64 */
    b       el1_fiq_lower_a64
    /* Entry 0x580: SError from lower EL using AArch64 */
    b       el1_serror_lower_a64

    /* Entry 0x600: Sync from lower EL using AArch32 (if used) */
    b       el1_sync_lower_a32
    /* Entry 0x680: IRQ from lower EL using AArch32 */
    b       el1_irq_lower_a32
    /* Entry 0x700: FIQ from lower EL using AArch32 */
    b       el1_fiq_lower_a32
    /* Entry 0x780: SError from lower EL using AArch32 */
    b       el1_serror_lower_a32
```

This skeleton illustrates the structure and offsets. Real systems often use a compact prologue within each entry instead of immediate branches, but the fixed offset map remains the same.

5.2 Vector Offsets and Alignment

Vector entry selection is **offset-based**. Hardware computes:

$$vector\_address = vector\_base + entry\_offset$$

Fixed offset spacing

Each vector entry region occupies a fixed-size slot; the canonical AArch64 layout uses **0x80-byte spacing** between consecutive entries in a group. This enables predictable placement of prologue code (saving context, switching stacks).

Alignment requirement (base address)

The base of the vector table must satisfy a strict architectural alignment constraint (commonly 2KB alignment in AArch64 practice). You should always enforce this at assembly/link time to avoid undefined or faulting behavior.

```
/* Enforce a safe base alignment for the vector table. */
.text
.align  11                  /* 2^11 = 2048-byte alignment */
.global vectors_el1_aligned
vectors_el1_aligned:
    /* entries at fixed offsets relative to this base */
    b       el1_sync_sp0
```

Why spacing matters

If your entry code exceeds the slot capacity or overlaps the next slot, the table becomes invalid. Therefore:

- keep the immediate vector entry prologue small and deterministic
- branch to larger handlers located elsewhere

5.3 Vector Tables per Exception Level

Each EL that can take exceptions has its own vector base register:

- **EL1**: `VBAR_EL1` defines the vector base when exceptions are taken to EL1
- **EL2**: `VBAR_EL2` defines the vector base when exceptions are taken to EL2
- **EL3**: `VBAR_EL3` defines the vector base when exceptions are taken to EL3

Practical meaning

- On a typical Linux system, the primary vector table you care about for syscalls and faults is **EL1**'s.
- In virtualization, guest/host arrangements may involve EL2 vectoring for traps and hypervisor interrupts.
- In secure firmware, EL3 has its own vectoring for secure monitor duties.

Conceptual diagram (policy, not code)

- EL0 code never directly "sets its own vectors".
- EL1/EL2/EL3 configure their own `VBAR_ELx` during early boot or initialization.

5.4 Selecting the Active Vector Table

Selecting the active vector table is done by writing the appropriate **VBAR** register at the target EL. This is a privileged operation and is part of early initialization.

5.4.1 Setting `VBAR_EL1` (conceptual)

```
/* Conceptual EL1 initialization: point VBAR_EL1 at the EL1 vector
 ↪  table base. */
.text
.global setup_vbar_el1_concept
.type setup_vbar_el1_concept, %function
setup_vbar_el1_concept:
    /* x0 = address of vectors_el1_aligned */
    adr     x0, vectors_el1_aligned
    msr     vbar_el1, x0
    isb                         /* ensure subsequent exceptions use
     ↪  the new vectors */
    ret
```

5.4.2 Selecting between multiple tables (common pattern)

Many kernels maintain separate vector tables for different phases or configurations:

- early boot vectors (minimal, safe)
- normal runtime vectors (full handlers, per-CPU stacks ready)
- special vectors (debug, crash, or alternate stack policies)

Switching is simply changing `VBAR_ELx` to a different aligned base, then executing an `ISB`.

```
/* Switch EL1 vectors to an alternate table (conceptual). */
.text
.global switch_vbar_el1_concept
.type switch_vbar_el1_concept, %function
switch_vbar_el1_concept:
    adr     x0, vectors_el1_aligned
    msr     vbar_el1, x0
    isb
    ret
```

Why the ISB matters

The instruction synchronization barrier ensures the processor recognizes the updated vector base before taking subsequent exceptions. Without it, the change may not take effect immediately in the way you expect.

Cross-check discipline

When debugging vectoring issues, confirm:

- `VBAR_ELx` points to the intended aligned address
- the table has correct offsets and does not overflow slot boundaries
- the entry reached matches the exception type and origin context (sync/IRQ/FIQ/SError and SP0/SPx/lower EL)

Chapter 6

Exception Entry Mechanics

6.1 What Happens on Exception Entry

When an exception is taken in AArch64, hardware performs a **precise control transfer** to an exception vector entry at a target Exception Level (EL). This transfer is not a normal branch or call: it is a privileged state transition governed by architectural rules.
At a high level, exception entry performs four categories of work:

1. **Choose a vector entry** based on exception type (Sync/IRQ/FIQ/SError) and origin context (current EL vs lower EL, SP0 vs SPx, AArch64 vs AArch32).

2. **Record return information** so software can resume execution later (or terminate cleanly).

3. **Capture the old state** (PSTATE snapshot) and establish the new execution context (new EL, masks, and routing rules).

4. **Populate syndrome information** describing the exception cause (for most synchronous exceptions and aborts).

The minimal programmer truth

- Exceptions redirect control to a **vector base + offset**.
- The CPU **saves where you were** and **what state you were in**.
- The CPU switches to a **privileged context** where handler code can inspect the reason and act.

Conceptual entry sketch (EL0 to EL1)

```
/* Conceptual flow: an EL0 event causes entry into EL1 vectors. */
.text
.global el0_trigger_example
.type el0_trigger_example, %function
el0_trigger_example:
    mov     x8, #172          /* syscall number (example) */
    svc     #0                /* synchronous exception => EL1 entry
     ↪  */
    ret                       /* resumed after exception return */
```

6.2 PC, PSTATE, and SPSR Saving

AArch64 exception entry is centered around three core facts:

- the CPU must know **where to return** (the PC at the time of exception)
- the CPU must remember **what execution state** it interrupted (PSTATE snapshot)
- the handler must have **evidence** about why the exception happened (syndrome registers)

6.2.1 Return address: `ELR_ELx`

On entry to ELx, hardware sets:

- `ELR_ELx` ← return address (the instruction address to resume at)

For many synchronous exceptions, the return address corresponds to the faulting instruction or the next instruction depending on the exception class and rules.

6.2.2 Saved program state: `SPSR_ELx`

Hardware saves the interrupted context's PSTATE into:

- `SPSR_ELx` ← saved PSTATE (condition flags, interrupt masks, execution state, and other control bits)

6.2.3 Live state in handler: PSTATE at the target EL

The handler begins executing with a new PSTATE appropriate for ELx (privileged), with masks and routing behavior that allow controlled handling.

6.2.4 Syndrome: `ESR_ELx` and `FAR_ELx`

For most synchronous exceptions, hardware provides:

- `ESR_ELx`: exception class + details (what kind of exception, key attributes)
- `FAR_ELx`: fault address for address-related aborts (when applicable)

Evidence capture stub (conceptual)

```
/* EL1: capture the architectural evidence at entry. */
```

```
.text
.global el1_capture_evidence
.type el1_capture_evidence, %function
el1_capture_evidence:
    mrs     x0, elr_el1         /* return address */
    mrs     x1, spsr_el1        /* saved PSTATE */
    mrs     x2, esr_el1         /* syndrome */
    mrs     x3, far_el1         /* fault address (valid for aborts) */
    ret
```

Discipline note

Never assume a crash reason from symptoms. The minimal proof chain is: **ESR** (class) + **ELR** (where) + **FAR** (which address, if any) + saved state (**SPSR**).

6.3 Stack Pointer Selection (SP0 vs SPx)

AArch64 provides distinct stack pointer models that matter for exception entry.

6.3.1 The two SP names

- **`SP0`**: stack pointer associated with EL0 context
- **`SPx`**: stack pointer associated with the *current* EL (e.g., SP1 at EL1, SP2 at EL2, SP3 at EL3)

6.3.2 Why the CPU cares

On exception entry, the hardware selects the vector entry based on whether the interrupted context at the target EL was using:

- `SP0` (often used to service exceptions that must preserve a separate user stack model)
- `SPx` (the privileged stack for the target EL)

6.3.3 Typical OS policy (conceptual)

Most kernels follow a strict rule:

- EL0 uses an unprivileged user stack.
- EL1 uses a kernel stack (per-thread or per-CPU).
- On entry from EL0 to EL1, the kernel ensures it runs on a safe EL1 stack before doing heavy work.

Vector-level prologue: switch to a known-safe stack (conceptual)

```
/* Conceptual entry prologue: establish a safe kernel stack then
 ↪  branch. */
.text
.global el1_sync_lower_a64
.type el1_sync_lower_a64, %function
el1_sync_lower_a64:
    /* In a real kernel, SP is already the EL1 stack per entry
     ↪  rules/policy.
       Some designs still set up per-CPU/thread stacks here. */
    /* Save minimal volatile state quickly */
    stp     x0, x1, [sp, #-16]!
    stp     x2, x3, [sp, #-16]!
    /* Read syndrome and route */
    mrs     x0, esr_el1
    b       el1_sync_dispatch
```

Common pitfall

If an entry path mistakenly uses an untrusted or corrupted stack, the handler can fault again before it can even diagnose the original issue. Therefore: keep early entry code short and stack-safe.

6.4 Privilege and State Changes

Exception entry can change:

- **Exception Level** (EL0 → EL1 for syscalls/faults in normal OS designs)
- **interrupt masks** (IRQ/FIQ masking policy on entry)
- **access to privileged registers and memory regions**
- **execution context** (security state and virtualization routing are platform-configurable)

6.4.1 Privilege change (EL transition)

The most common transition for programmers is:

- **EL0** (user) triggers an exception
- handler executes at **EL1** (kernel)

The kernel's job is to:

- validate and service the request (syscall) or diagnose the fault (abort)
- preserve isolation and prevent escalation
- return to EL0 via `ERET` with controlled state restoration

6.4.2 State changes and barriers

After privileged control-register changes (including vector base updates), an **ISB** is typically required to ensure subsequent instruction execution and exception behavior observes the new configuration.

Example: a syscall entry vs a fault entry

Both are synchronous exceptions, but the post-entry policy differs.

```
/* EL0: syscall (intentional). */
.text
.global el0_syscall_example
.type el0_syscall_example, %function
el0_syscall_example:
    mov     x8, #64           /* __NR_write (example) */
    svc     #0                /* EL0 -> EL1 */
    ret
```

```
/* EL0: fault (unintentional). */
.text
.global el0_fault_example
.type el0_fault_example, %function
el0_fault_example:
    mov     x0, #0
    ldr     x1, [x0]          /* EL0 -> EL1 data abort */
    ret
```

In both cases, EL1 receives control and reads `ESR_EL1`. For `SVC`, it dispatches to the syscall table. For a data abort, it consults `FAR_EL1` and the memory subsystem state to decide whether to fix up (e.g., demand paging) or to terminate/deliver a signal.

Entry invariants you should memorize

- Return PC is recorded in `ELR_ELx`.
- Interrupted PSTATE is recorded in `SPSR_ELx`.
- Exception cause is described in `ESR_ELx` (and `FAR_ELx` when applicable).
- Control transfers to `VBAR_ELx` + fixed offset, and the handler returns with `ERET`.

Chapter 7

Exception Return Mechanics

7.1 Restoring Execution Context

Exception return in AArch64 is a **hardware-defined restoration** of the interrupted context. Unlike a normal function return, which restores state only by convention, an exception return restores state using architectural registers captured on exception entry.
A correct return has two responsibilities:

1. **Restore general-purpose / SIMD / system-visible state** that the handler chose to save on a stack or in per-CPU storage (this part is an OS/firmware policy decision).

2. **Restore architectural execution state** (PC + PSTATE) using the exception-link and saved-state registers `ELR_ELx` and `SPSR_ELx` (this part is architectural, not policy).

Two-layer model (memorize this)

- **Policy layer (software)**: which registers did the entry code save? where? how is the kernel stack arranged?

- **Architectural layer (hardware)**: `ERET` returns to `ELR_ELx` and restores PSTATE from `SPSR_ELx`.

Typical EL1 return prologue (conceptual)

```
/* Conceptual: restore a small set of registers, then return via
 ↪  ERET. */
.text
.global el1_return_concept
.type el1_return_concept, %function
el1_return_concept:
    /* Restore registers saved by the entry prologue
     ↪  (policy-defined). */
    ldp     x2, x3, [sp], #16
    ldp     x0, x1, [sp], #16
    /* Architectural return: PC and PSTATE restored from
     ↪  ELR_EL1/SPSR_EL1. */
    eret
```

This skeleton is intentionally minimal: real kernels save many more registers and often use a structured trap frame. The essential point is that `ERET` is the final step that restores architectural state.

7.2 `ERET` Instruction Semantics

`ERET` (Exception Return) completes an exception by restoring:

- **PC** from `ELR_ELx`
- **PSTATE** from `SPSR_ELx`

Then execution continues in the restored context, which can be at a lower EL (commonly EL0) or, in some flows, at the same EL (returning from a nested exception).

What `ERET` is not

- It is not a normal `RET`. `RET` uses `X30` (LR) and does not restore PSTATE.
- It is not optional. It is the architectural mechanism for returning from an exception and restoring privilege state.

Minimal proof: the two registers that matter

If `ELR_ELx` or `SPSR_ELx` is wrong, `ERET` cannot "fix it later". The return either:

- resumes the wrong location (bad `ELR_ELx`)
- resumes with the wrong flags/masks/state (bad `SPSR_ELx`)
- or faults immediately due to invalid state

Example: explicitly adjusting return PC (fixup pattern)

Some handlers implement a fixup by modifying `ELR_ELx` before `ERET` (e.g., skipping a faulting instruction in a controlled recovery scenario).

```
/* Conceptual: advance ELR_EL1 to skip an instruction, then return.
 ↪  */
.text
.global el1_skip_faulting_insn_concept
.type el1_skip_faulting_insn_concept, %function
el1_skip_faulting_insn_concept:
    mrs     x0, elr_el1
```

```
    add     x0, x0, #4          /* A64 instruction width is 4 bytes */
    msr     elr_el1, x0
    eret
```

This is a powerful mechanism and therefore dangerous: use only when the architecture and OS policy guarantee correctness.

7.3 Returning Across Exception Levels

Most programmer-visible returns are **EL1 → EL0**, because syscalls and faults are typically handled in EL1.

7.3.1 EL0 → EL1 → EL0 (syscall round-trip)

```
/* EL0: invoke a syscall; kernel returns via ERET. */
.text
.global el0_syscall_roundtrip
.type el0_syscall_roundtrip, %function
el0_syscall_roundtrip:
    mov     x8, #172            /* __NR_getpid (example) */
    svc     #0                  /* EL0 -> EL1 */
    /* After ERET, we're back in EL0 and continue here. */
    ret
```

What the kernel must ensure before returning

Before executing `ERET`, the handler must ensure:

- `ELR_EL1` points to the correct EL0 resume PC
- `SPSR_EL1` encodes a valid EL0 return state (including interrupt masks policy)

- user-visible registers (e.g., syscall return in `X0`) are set according to ABI
- stack and trap-frame state are consistent (no corruption, correct alignment)

7.3.2 Nested exceptions and same-EL returns

Exceptions can occur while already in EL1 (e.g., an IRQ arriving while the kernel runs). In that case:

- the exception may be taken to EL1 (same EL) using a different vector entry (SP0 vs SPx context)
- `ERET` returns to the interrupted EL1 context

```
/* Conceptual: IRQ hits while executing in EL1, then returns back to
 ↪  EL1. */
.text
.global el1_work_loop_concept
.type el1_work_loop_concept, %function
el1_work_loop_concept:
1:
    /* Kernel work... */
    add     x0, x0, #1
    /* IRQ may arrive here -> EL1 IRQ vector -> ERET -> back to this
     ↪  loop */
    b       1b
```

7.3.3 Crossing EL2/EL3

Returning from EL2 or EL3 also uses `ERET` with the corresponding `ELR_EL2`/`SPSR_EL2` or `ELR_EL3`/`SPSR_EL3`. The mechanics remain the same; what changes is:

- the security/virtualization routing policy
- which state is permitted on return

7.4 Common Return Pitfalls

Return bugs are among the hardest to debug because they often appear as "random crashes" far from the cause. These are the highest-frequency failure modes in exception return code.

7.4.1 Pitfall 1: Corrupting `ELR_ELx` or `SPSR_ELx`

- Wrong `ELR_ELx` ⇒ return to the wrong address (often immediate instruction abort).
- Wrong `SPSR_ELx` ⇒ invalid return state, wrong masks, or privilege mismatch.

7.4.2 Pitfall 2: Using `RET` instead of `ERET`

A normal `RET` does not restore PSTATE and does not perform an exception-level return. If you `RET` from a vector path, you typically remain in EL1 and break the return contract.

```
/* Incorrect pattern (conceptual): returning with RET from an
 ↪  exception path. */
.text
.global wrong_return_example
.type wrong_return_example, %function
wrong_return_example:
    /* ... handler work ... */
    ret                         /* WRONG: does not restore PSTATE/EL
     ↪  */
```

7.4.3 Pitfall 3: Not restoring registers per the handler's own save policy

Even if `ERET` is correct, corrupting general-purpose registers or SIMD state breaks user-space execution silently. Syscall returns and signal delivery are especially sensitive.

7.4.4 Pitfall 4: Stack misalignment and frame corruption

Many entry paths require strict stack alignment. If your save/restore pairs do not match exactly, the restore sequence will load the wrong values and the system will fail on return.

```
/* Common stack bug pattern: mismatched push/pop sizes
 ↪  (illustrative). */
.text
.global bad_stack_restore_concept
.type bad_stack_restore_concept, %function
bad_stack_restore_concept:
    stp     x0, x1, [sp, #-16]!
    stp     x2, x3, [sp, #-16]!
    /* ... */
    ldp     x0, x1, [sp], #16  /* wrong order: x2/x3 not restored
     ↪  first */
    ldp     x2, x3, [sp], #16
    eret
```

7.4.5 Pitfall 5: Forgetting required barriers after control changes

When handlers change control registers that affect execution or exception behavior (vectors, MMU state, masks), architectural barriers (especially `ISB`) are required to ensure the new state is observed before returning.

Return discipline checklist (short)

- Restore exactly what you saved, in reverse order, with correct stack adjustment.
- Ensure `ELR_ELx` and `SPSR_ELx` are valid for the intended return context.
- Return with `ERET`, not `RET`.
- Keep vector return paths minimal and deterministic; branch to larger code if needed.

Chapter 8

System Registers for Exceptions

8.1 ELR_ELx Registers

ELR_ELx (Exception Link Register) holds the **return address** for an exception taken to ELx. It is the architectural source of the resumed PC when executing ERET.

Key facts

- ELR_EL1 is written on exceptions taken to EL1.
- ELR_EL2 is written on exceptions taken to EL2.
- ELR_EL3 is written on exceptions taken to EL3.
- ERET returns to PC ← ELR_ELx.

What ELR_ELx answers during debugging

- *Where was the CPU about to continue when the exception was taken?*

- In most synchronous cases: *Which instruction address is responsible for the trap/fault?*

Example: capture ELR at entry

```
/* EL1: capture the return PC for diagnostics or dispatch. */
.text
.global read_elr_el1
.type read_elr_el1, %function
read_elr_el1:
    mrs     x0, elr_el1
    ret
```

Common pitfall

`ELR_ELx` is not a general-purpose scratch register. If corrupted, `ERET` returns to the wrong address, often producing an immediate instruction abort or another exception loop.

8.2 `SPSR_ELx`

`SPSR_ELx` (Saved Program Status Register) holds a snapshot of the interrupted PSTATE when the exception was taken to ELx.

Key facts

- `SPSR_ELx` is the **saved PSTATE** for the interrupted context.
- `ERET` restores PSTATE from `SPSR_ELx`.
- It encodes condition flags, interrupt masks, and other state fields required to resume correctly.

What SPSR_ELx answers during debugging

- *Was the faulting context EL0 or a higher EL?*
- *Were IRQ/FIQ masked at the time?*
- *What were the condition flags (NZCV) when the exception occurred?*

Example: capture SPSR at entry

```
/* EL1: capture saved PSTATE. */
.text
.global read_spsr_el1
.type read_spsr_el1, %function
read_spsr_el1:
    mrs     x0, spsr_el1
    ret
```

Common pitfall

Incorrectly constructing or modifying SPSR_ELx before ERET can cause invalid return state, wrong interrupt masking, or privilege mismatches that fault immediately on return.

8.3 ESR_ELx

ESR_ELx (Exception Syndrome Register) is the primary **classification and detail** register for exceptions taken to ELx. It answers: *What kind of exception was this, and what key attributes does the architecture provide?*

Key fields (conceptual, without bit tables)

- **Exception Class (EC)**: identifies the broad category (SVC, data abort, instruction abort, illegal instruction, breakpoint, etc.).
- **Instruction Length (IL)**: indicates instruction size for some classes (A64 is fixed-width 4 bytes; the IL field matters for mixed-state contexts).
- **Instruction Specific Syndrome (ISS)**: class-specific details (e.g., abort status, access type, level, etc.).

What `ESR_ELx` answers during debugging

- *Was it a syscall (SVC) or a fault (abort) or a debug trap (BRK)?*
- *If it was an abort, what kind (translation vs permission vs alignment) and what attributes were involved?*
- *If it was an instruction exception, what was trapped/illegal?*

Example: read ESR and branch by class (conceptual)

```
/* EL1: classify exception by ESR_EL1 exception class (EC). */
.text
.global classify_by_esr_concept
.type classify_by_esr_concept, %function
classify_by_esr_concept:
    mrs     x0, esr_el1

    /* Extract EC in a conceptual way:
       EC is a high field; exact bit positions are
        ↪  architecture-defined.
```

```
    Here we show the intent: shift and mask to get EC into x1. */
lsr     x1, x0, #26         /* place EC into low bits (conceptual)
 ↪  */
and     x1, x1, #0x3f       /* EC is a 6-bit class (conceptual) */

/* Dispatch table lookup would use x1 as the index. */
ret
```

Common pitfall

Do not treat ESR_ELx as "the fault address". That is FAR_ELx when applicable. ESR_ELx tells you the **class and attributes**, not the address.

8.4 FAR_ELx

FAR_ELx (Fault Address Register) holds the faulting **virtual address** for address-related aborts taken to ELx, when the architecture defines it as valid for the exception class.

Key facts

- Most commonly used for **data aborts** and **instruction aborts**.
- It provides the **virtual address** that caused the fault (not the physical address).
- It is not meaningful for all exception classes (e.g., SVC does not use FAR).

What FAR_ELx answers during debugging

- *Which virtual address triggered the abort?*
- *Is the fault address near null, near a freed region, or in a protected mapping?*

Example: capture FAR on abort entry (conceptual)

```
/* EL1: capture FAR and combine with ESR/ELR for a full diagnosis. */
.text
.global capture_abort_state_concept
.type capture_abort_state_concept, %function
capture_abort_state_concept:
    mrs     x0, esr_el1         /* class + attributes */
    mrs     x1, far_el1         /* fault VA (for aborts) */
    mrs     x2, elr_el1         /* faulting instruction address */
    mrs     x3, spsr_el1        /* saved state */
    ret
```

Common pitfall

A non-canonical assumption is that FAR is always valid. It is only meaningful for specific exception classes. Always verify the exception class in ESR before trusting FAR.

8.5 Reading and Interpreting Exception State

A correct exception diagnosis is built from a minimal evidence set:

> **ESR_ELx (what) + ELR_ELx (where instruction) + FAR_ELx (which address, if any) + SPSR_ELx (what state)**

8.5.1 A disciplined triage workflow (EL1 example)

1. Read `ESR_EL1` and determine the broad class:
    - syscall (`SVC`)

- abort (instruction/data)
- debug trap (BRK)
- illegal/trapped instruction
- other

2. Read ELR_EL1 to locate the responsible instruction address.
3. If and only if ESR indicates an abort/address fault, read FAR_EL1.
4. Read SPSR_EL1 to determine the interrupted context properties (EL, masks, flags).

8.5.2 Example 1: syscall path evidence

```
/* EL1: syscall dispatch typically keys off ESR (class indicates SVC
 ↪  from lower EL). */
.text
.global el1_syscall_dispatch_concept
.type el1_syscall_dispatch_concept, %function
el1_syscall_dispatch_concept:
    mrs     x0, esr_el1         /* should indicate SVC from EL0 */
    mrs     x1, elr_el1         /* return PC */
    mrs     x2, spsr_el1        /* saved state */
    /* FAR not needed for SVC */
    /* Dispatch to syscall table using x8 from the saved register
     ↪  frame (policy-defined). */
    eret
```

8.5.3 Example 2: data abort evidence and first interpretation

```
/* EL1: abort triage combines ESR + FAR + ELR. */
```

```
.text
.global el1_data_abort_triage_concept
.type el1_data_abort_triage_concept, %function
el1_data_abort_triage_concept:
    mrs     x0, esr_el1         /* abort class + abort attributes */
    mrs     x1, far_el1         /* fault VA */
    mrs     x2, elr_el1         /* faulting instruction address */
    mrs     x3, spsr_el1        /* context state */
    /* Next steps (policy):
       - classify translation vs permission vs alignment using ESR
        ↪  details
       - consult page tables / VM subsystem
       - fixup (page-in) or signal/kill
    */
    eret
```

8.5.4 Example 3: illegal instruction vs privileged access

```
/* EL0: attempt to read a privileged register => synchronous
 ↪  exception. */
.text
.global el0_illegal_priv_example
.type el0_illegal_priv_example, %function
el0_illegal_priv_example:
    mrs     x0, sctlr_el1       /* illegal at EL0 */
    ret
```

The handler reads:

- `ESR_EL1` to distinguish undefined instruction vs trapped privileged access

- `ELR_EL1` to identify the exact instruction location
- `SPSR_EL1` to confirm the exception originated from EL0

Practical summary

- **ELR** answers *where to return and where it happened*.
- **SPSR** answers *what the interrupted state was*.
- **ESR** answers *what kind of exception and why, in architectural terms*.
- **FAR** answers *which virtual address*, when the exception class defines it as meaningful.

Chapter 9

Linux Syscalls on AArch64

9.1 What Is a System Call?

A **system call** (syscall) is the controlled mechanism by which an unprivileged program (EL0) requests a privileged service implemented by the kernel (EL1). It exists because:

- user code must not directly program devices, change page tables, or access privileged registers
- the kernel must validate requests, enforce permissions, and preserve isolation

Syscall vs function call (the important difference)

A function call:

- stays in the same privilege level
- returns by convention (`RET` using LR)

A syscall:

- enters the kernel via a **synchronous exception**
- executes in **EL1** with privileged access
- returns to EL0 via **`ERET`** (exception return), not a normal `RET`

What syscalls provide

- file I/O (open/read/write/close)
- process control (fork/exec/exit/wait)
- memory management (mmap/munmap/mprotect/brk)
- signals and timers
- networking and IPC primitives

9.2 Syscall Path: EL0 → EL1

On AArch64 Linux, the syscall path is the most common privilege transition a programmer triggers.

High-level path

1. EL0 sets up syscall arguments in registers.
2. EL0 places the syscall number into a dedicated register.
3. EL0 executes `SVC #0` to raise a synchronous exception.
4. CPU enters the EL1 synchronous exception vector.

5. Kernel decodes the reason (SVC from EL0), dispatches the syscall handler, and produces a return value.
6. Kernel returns to EL0 using `ERET`; user code continues at the instruction after `SVC`.

Architectural evidence on entry

At EL1 entry, the kernel can read:

- `ESR_EL1`: indicates the exception class is SVC (from lower EL)
- `ELR_EL1`: return PC (address after `SVC`)
- `SPSR_EL1`: saved EL0 state

Minimal syscall entry/return sketch

```
/* EL0: syscall request. */
.text
.global el0_syscall_example
.type el0_syscall_example, %function
el0_syscall_example:
    /* x0..x5 hold arguments; x8 holds syscall number (Linux
    ↪  AArch64). */
    svc     #0
    /* After kernel ERET, x0 holds return value. */
    ret
```

9.3 SVC Instruction Semantics

`SVC` (Supervisor Call) is an instruction that intentionally triggers a **synchronous exception**. It is the architectural gateway from unprivileged code to the supervisor/OS layer.

Semantics (programmer-accurate view)

- `SVC #imm` causes an exception taken to the configured handler EL (on Linux, this is EL1).
- The immediate operand (`#imm`) is not used as a syscall number on AArch64 Linux; Linux uses a dedicated register for the syscall number.
- The return address recorded in `ELR_EL1` points to the instruction after `SVC`.
- The kernel returns using `ERET`, restoring EL0 state from `SPSR_EL1`.

Minimal `SVC` example

```
/* EL0: explicitly call into kernel. */
.text
.global demo_svc
.type demo_svc, %function
demo_svc:
    svc     #0                  /* synchronous exception to EL1 */
    ret
```

Common misconception

`SVC` does not "jump to the kernel function you want". It raises an exception; the kernel inspects state (including the syscall number register) and dispatches.

9.4 Syscall Numbering and ABI Rules

Linux syscalls are identified by an integer syscall number plus a fixed ABI for argument passing and returns. The ABI is intentionally simple and fast.

9.4.1 Register convention (Linux AArch64 syscall ABI)

- Syscall number: `X8`
- Up to 6 arguments: `X0--X5`
- Return value: `X0`

Error reporting rule (kernel interface view)

Linux syscalls return a value in `X0`. On error, the kernel returns a **negative** value representing an error code in the kernel ABI. In normal user-space programming, the C library typically translates this into:

- function returns `-1`
- `errno` is set to the positive error number

If you call syscalls directly (without libc), you must handle the raw return convention yourself.

9.4.2 Example 1: direct `getpid` syscall

```
/* long getpid_syscall(void); returns pid in x0. */
.text
.global getpid_syscall
.type getpid_syscall, %function
getpid_syscall:
    mov     x8, #172          /* __NR_getpid (Linux AArch64) */
    svc     #0
    ret
```

9.4.3 Example 2: direct `write` syscall (3-arg syscall)

```
/* ssize_t write(int fd, const void* buf, size_t len); */
.text
.global write_syscall
.type write_syscall, %function
write_syscall:
    /* Expected input:
       x0 = fd
       x1 = buf
       x2 = len
    */
    mov     x8, #64           /* __NR_write (Linux AArch64) */
    svc     #0
    /* x0 = bytes written, or negative error code */
    ret
```

9.4.4 Example 3: minimal error check without libc

```
/* If x0 is negative, treat it as an error code (kernel ABI view). */
.text
.global write_syscall_checked
.type write_syscall_checked, %function
write_syscall_checked:
    mov     x8, #64
    svc     #0
    /* Check sign bit: negative => error */
    cmp     x0, #0
    b.ge    1f
```

```
    /* Error path: x0 holds -errno (kernel ABI). Convert to
     ↪  errno-like positive. */
    neg     x0, x0              /* x0 = errno (positive) */
1:
    ret
```

9.4.5 ABI rules you must obey

- **Do not assume** syscalls preserve registers the way function calls do. The kernel interface is not a normal ABI; treat it as a boundary.
- Pass exactly the required arguments in `X0--X5` and the syscall number in `X8`.
- Assume **return in `X0`** and handle negative error returns if bypassing libc.
- Use `SVC #0` as the entry instruction in standard Linux AArch64 user space.

Practical summary

- A syscall is a controlled EL0→EL1 transition implemented as a synchronous exception.
- `SVC` triggers the transition; `X8` selects the syscall; `X0--X5` carry arguments.
- The kernel returns via `ERET`; user-space resumes after `SVC` with the result in `X0`.

Chapter 10

Syscall Calling Convention

10.1 Register Usage for Syscalls

On Linux AArch64, a syscall is invoked from EL0 using `SVC #0`. The kernel interface uses a fixed register convention that is intentionally small and fast.

Register map (Linux AArch64 syscall ABI)

- **Syscall number**: `X8`
- **Arguments (up to 6)**: `X0--X5`
- **Return value**: `X0`

Strictness rule

A syscall ABI is not "flexible like C varargs". The kernel entry path will interpret registers exactly as specified. If you place values in the wrong registers, the kernel will execute the wrong syscall or interpret arguments incorrectly.

Minimal template

```
/* Generic syscall template:
   x8 = syscall number
   x0..x5 = args
   result in x0
*/
.text
.global syscall0_template
.type syscall0_template, %function
syscall0_template:
    svc     #0
    ret
```

10.2 Argument Passing Rules

10.2.1 Argument registers and width

- Arguments are passed in `X0--X5`.
- Integers and pointers use 64-bit registers in AArch64 user space.
- Smaller integer types are passed in `Xn` with the usual integer promotion rules applied by C/C++ calling code.

10.2.2 Pointers and user memory

- Pointer arguments refer to **EL0 virtual addresses** in the calling process.
- The kernel must validate user pointers and may fail the syscall if the pointer is invalid or lacks permissions.

10.2.3 More than 6 arguments

Linux syscalls are designed to fit in 6 registers for the fast path. If a conceptual API needs more data, it usually passes a pointer to a user-defined structure in memory as one argument.

Example 1: 0-argument syscall (`getpid`)

```
/* long getpid_syscall(void) */
.text
.global getpid_syscall
.type getpid_syscall, %function
getpid_syscall:
    mov     x8, #172          /* __NR_getpid */
    svc     #0
    ret
```

Example 2: 3-argument syscall (`write`)

```
/* ssize_t write_syscall(int fd, const void* buf, size_t len)
   x0=fd, x1=buf, x2=len
*/
.text
.global write_syscall
.type write_syscall, %function
write_syscall:
    mov     x8, #64           /* __NR_write */
    svc     #0
    ret
```

Example 3: passing a pointer to a struct (pattern for complex syscalls)

```c
/* Conceptual C layout for a syscall that takes a pointer to a request
↪  structure. */
struct request {
    long op;
    long flags;
    void* buf;
    long len;
};
```

```asm
/* EL0: x0 points to a request structure; syscall reads fields from
↪  user memory. */
.text
.global syscall_struct_arg_concept
.type syscall_struct_arg_concept, %function
syscall_struct_arg_concept:
    /* x0 = pointer to struct request */
    mov     x8, #0            /* syscall number placeholder */
    svc     #0
    ret
```

This pattern keeps the syscall ABI stable: registers remain the same; the structure can evolve by versioning.

10.3 Return Values and Error Handling

10.3.1 Return value location

- The syscall return value is placed in `X0`.

- For syscalls that return two values, Linux typically uses `X0` (primary) and sometimes `X1` by convention for specific cases, but the standard expectation is: **use `X0`** unless the syscall contract says otherwise.

10.3.2 Kernel ABI error rule (raw syscall interface)

At the raw kernel interface, errors are returned as:

- `X0` = negative error code (i.e., `-errno`)

User-space C libraries normally translate this into:

- return `-1`
- set `errno` to the positive error code

If you bypass libc, you must implement your own translation.

Example: translate raw negative return to errno-like positive

```
/* If x0 < 0, convert to a positive errno in x0. */
.text
.global normalize_errno_concept
.type normalize_errno_concept, %function
normalize_errno_concept:
    cmp     x0, #0
    b.ge    1f
    neg     x0, x0              /* x0 = errno */
1:
    ret
```

Example: raw `openat` style pattern (4-arg syscall)

```
/* long openat(int dirfd, const char* path, int flags, int mode)
   x0=dirfd, x1=path, x2=flags, x3=mode
*/
.text
.global openat_syscall_concept
.type openat_syscall_concept, %function
openat_syscall_concept:
    mov     x8, #56           /* __NR_openat (Linux AArch64) */
    svc     #0
    /* x0 = fd or -errno */
    ret
```

10.3.3 Return vs fault

A syscall can fail in two distinct ways:

- **Normal failure**: syscall returns `-errno` in `X0` (e.g., permission denied).
- **Fault during argument access**: the kernel may detect invalid user memory and return an error; but some faults can also surface asynchronously to user space as signals depending on the path and context.

For direct syscall usage: treat **negative `X0`** as the primary error signal.

10.4 Differences from Function Calls

Syscalls resemble function calls syntactically (set regs then call), but they differ in the most important ways:

10.4.1 1) Control transfer mechanism

- Function call: `BL` / `RET` within the same EL.
- Syscall: `SVC` triggers an exception; kernel returns via `ERET`.

10.4.2 2) ABI ownership

- Function call ABI is defined by the platform C ABI (AAPCS64) and respected by compilers.
- Syscall ABI is defined by the **kernel interface** and may clobber state differently than a normal call.

10.4.3 3) Register preservation expectations

- For function calls, compilers rely on caller/callee-saved conventions.
- For syscalls, do not assume the same preservation model; treat the kernel boundary as a special interface.

10.4.4 4) Error reporting

- Function calls typically signal errors via return values, exceptions, or out-parameters by API design.
- Syscalls use the raw kernel ABI: negative return codes, translated by libc into `errno`.

Side-by-side example: function vs syscall

```
/* Function call: stays in EL0, uses BL/RET. */
.text
```

```
.global call_local_function
.type call_local_function, %function
call_local_function:
    bl      local_function
    ret

local_function:
    add     x0, x0, #1
    ret
```

```
/* Syscall: enters EL1 via SVC, returns via ERET (in kernel). */
.text
.global call_syscall_getpid
.type call_syscall_getpid, %function
call_syscall_getpid:
    mov     x8, #172
    svc     #0
    ret
```

Practical discipline

- Use libc wrappers unless you have a specific reason to issue raw syscalls.
- If you do raw syscalls: obey `X8`, `X0--X5`, return in `X0`, and handle `-errno`.
- Never confuse `SVC`/`ERET` control flow with `BL`/`RET`.

Chapter 11

Syscall Entry in the Linux Kernel

11.1 Exception Routing for `SVC`

On AArch64, `SVC` raises a **synchronous exception**. Linux configures the system such that an `SVC` executed in EL0 is taken to the **EL1 synchronous exception vector**. In other words:

> **EL0 executes `svc #0` ⇒ CPU vectors into EL1 ⇒ kernel dispatches syscall ⇒ return to EL0 via `eret`**

What the kernel relies on at entry

Upon entry to EL1, Linux depends on architectural evidence registers:

- `ESR_EL1`: classifies the exception as SVC from a lower EL
- `ELR_EL1`: return address into user code (instruction after `SVC`)
- `SPSR_EL1`: saved user PSTATE (origin context and masks)

Why routing must be strict

If SVC is not routed to the expected vector, the syscall ABI breaks:

- wrong entry path (wrong stack, wrong context, wrong save policy)
- inability to locate syscall number and arguments reliably
- failure to return to user space safely

11.2 Kernel Entry Code (Conceptual Walkthrough)

Linux kernel entry code for syscalls is optimized for:

- fast and deterministic entry from EL0
- correct context capture (registers, flags, return PC)
- quick dispatch based on syscall number
- safe return with the correct ABI-visible result

Conceptual stages of syscall entry

1. **Vector landing**: control arrives at the EL1 sync vector entry for "lower EL, AArch64".
2. **Minimal save**: save the user register state needed to run C code safely (trap frame).
3. **Classify**: confirm this sync exception is an SVC path (not an abort/debug trap).
4. **Extract syscall ABI inputs**:
    - syscall number from `X8`
    - arguments from `X0--X5`

5. **Dispatch**: index syscall table, call the handler (kernel C implementation) with sanitized inputs.

6. **Finalize**: place return value in `X0`; apply tracing/audit hooks if enabled.

7. **Return**: restore state and execute `ERET`.

A minimal conceptual vector entry sketch

```
/* Conceptual EL1 sync vector entry for exceptions from EL0
 ↪  (AArch64).
   This is not real Linux code; it captures the architectural
    ↪  mechanics. */
.text
.global el1_sync_from_el0_concept
.type el1_sync_from_el0_concept, %function
el1_sync_from_el0_concept:
    /* Save a minimal trap frame (policy-defined). */
    stp     x0, x1, [sp, #-16]!
    stp     x2, x3, [sp, #-16]!
    stp     x4, x5, [sp, #-16]!
    stp     x6, x7, [sp, #-16]!
    stp     x8, x9, [sp, #-16]!    /* x8 holds syscall number;
     ↪  preserve it */

    /* Read syndrome to classify */
    mrs     x10, esr_el1

    /* Branch to a dispatcher that distinguishes SVC vs abort vs
     ↪  other sync */
```

```
    b       el1_sync_dispatch_concept
```

The essential idea: the vector entry is short; it saves state and jumps to structured dispatch logic.

11.3 Switching Stacks and Context

Syscall entry must ensure the kernel runs on a **trusted EL1 stack** with a coherent per-thread context.

11.3.1 Why stack switching exists

User stacks are untrusted:

- user memory may be unmapped, maliciously altered, or near overflow
- kernel must never rely on EL0 stack integrity

Therefore, Linux arranges that on entry to EL1 from EL0, execution proceeds using a kernel stack associated with the current thread (or per-CPU stack for early entry stages), then transitions into the normal per-task kernel stack discipline.

Conceptual stack transition pattern

```
/* Conceptual: establish a known-safe EL1 stack (policy placeholder).
   Real kernels derive the stack from current task/thread info. */
.text
.global establish_el1_stack_concept
.type establish_el1_stack_concept, %function
establish_el1_stack_concept:
    /* x0 = pointer to per-thread kernel stack top (conceptual) */
```

```
    mov     sp, x0
    ret
```

11.3.2 Context capture (trap frame)

To safely call kernel C code, the entry path builds a trap frame containing:

- user GPRs (at least those needed by ABI and syscall dispatch)
- return state (`ELR_EL1`, `SPSR_EL1`)
- sometimes additional metadata (thread flags, syscall tracing state)

Capturing ELR/SPSR (conceptual)

```
/* Conceptual: capture architectural return state into a frame. */
.text
.global capture_return_state_concept
.type capture_return_state_concept, %function
capture_return_state_concept:
    mrs     x0, elr_el1
    mrs     x1, spsr_el1
    /* store x0/x1 into the trap frame (not shown) */
    ret
```

11.3.3 Common misconception

The kernel does not “return” to user space by restoring LR and executing `RET`. It returns by restoring **exception return state** and executing `ERET`.

11.4 Returning to User Space

Returning to user space is the reverse of entry, but it is not symmetric in code size: the return path is often even more disciplined because it must restore a correct user-visible state.

11.4.1 What must be true before `ERET`

Before executing `ERET`, the kernel ensures:

- `ELR_EL1` points to the user resume address (after `SVC`)
- `SPSR_EL1` encodes a valid EL0 return state
- user registers are restored, with syscall return value in `X0`
- pending signals or work are handled per kernel policy before re-entering EL0

Conceptual return stub

```
/* Conceptual: restore saved registers and return to EL0. */
.text
.global return_to_user_concept
.type return_to_user_concept, %function
return_to_user_concept:
    /* Restore registers in reverse order (must match saves). */
    ldp     x8, x9, [sp], #16
    ldp     x6, x7, [sp], #16
    ldp     x4, x5, [sp], #16
    ldp     x2, x3, [sp], #16
    ldp     x0, x1, [sp], #16
```

```
    /* x0 should already hold the syscall return value. */
    eret
```

11.4.2 Example: syscall return value placement

Syscall handlers compute a return value and place it in `X0` for user space.

```
/* Conceptual: set return value then return to user. */
.text
.global set_ret_and_eret_concept
.type set_ret_and_eret_concept, %function
set_ret_and_eret_concept:
    mov     x0, #0            /* success */
    eret
```

11.4.3 Practical failure modes (high frequency)

- trap-frame mismatch: restore does not match save order/size ⇒ corrupted user context
- wrong `ELR_EL1` or `SPSR_EL1` ⇒ return to wrong address or invalid state
- forgetting to preserve `X8` before dispatch ⇒ wrong syscall executed
- using user stack or untrusted pointers too early ⇒ nested faults during entry

Summary

- `SVC` from EL0 is routed to the EL1 synchronous vector.
- Kernel entry code captures state, switches to a trusted stack, and dispatches by `X8`.
- Return to user space is done by restoring context and executing `ERET`.

Chapter 12

Signals, Faults, and Exceptions

12.1 Page Faults vs Syscalls

From a programmer's perspective, both syscalls and page faults cause an **EL0 → EL1** transition. Architecturally, both are **synchronous exceptions**. The crucial difference is intent and meaning:

- **Syscall**: an intentional request for service (`SVC #0`).
- **Page fault**: an unintentional exception caused by a memory access that violates the current translation/permission state.

Evidence difference (kernel view)

- Syscall entry: `ESR_EL1` indicates an SVC-class exception; `FAR_EL1` is not relevant.
- Page fault entry: `ESR_EL1` indicates an abort class (data/instruction abort), and `FAR_EL1` holds the faulting virtual address.

Syscall example (intentional)

```
/* EL0: syscall entry via SVC. */
.text
.global demo_syscall_entry
.type demo_syscall_entry, %function
demo_syscall_entry:
    mov     x8, #172          /* __NR_getpid */
    svc     #0                /* EL0 -> EL1 */
    ret
```

Page fault example (unintentional)

```
/* EL0: likely page fault by dereferencing a null pointer. */
.text
.global demo_page_fault_null
.type demo_page_fault_null, %function
demo_page_fault_null:
    mov     x0, #0
    ldr     x1, [x0]          /* data abort: fault VA = 0 */
    ret
```

Why page faults are not always “bugs”

A page fault is a mechanism, not automatically an error:

- **Recoverable** faults exist (e.g., demand paging, copy-on-write).
- **Fatal** faults exist (e.g., unmapped address, forbidden access).

Linux often resolves recoverable faults transparently and returns to EL0 as if nothing happened. If it cannot resolve the fault, it converts it into a signal or terminates the process.

12.2 Illegal Instructions

An **illegal instruction** exception occurs when the CPU cannot legally execute the current instruction in the current context. Common cases include:

- undefined/unallocated instruction encoding
- executing an instruction not permitted at EL0 (privileged instruction)
- trapped instruction due to system configuration (e.g., certain system-register accesses)

Programmer-facing symptoms

- process terminates with **`SIGILL`** in many cases
- debugger shows a fault at the instruction address (from `ELR_EL1`)

Example: privileged system register access at EL0

```
/* EL0: attempt to read a privileged register => synchronous
 ↪  exception. */
.text
.global demo_illegal_mrs
.type demo_illegal_mrs, %function
demo_illegal_mrs:
    mrs     x0, sctlr_el1     /* illegal at EL0 */
    ret
```

Example: intentional illegal instruction for testing

```
/* EL0: BRK is a debug trap; in many contexts it leads to SIGTRAP or
 ↪  debugger stop. */
```

```
.text
.global demo_brk_trap
.type demo_brk_trap, %function
demo_brk_trap:
    brk     #0
    ret
```

Kernel classification

Linux classifies the exception using `ESR_EL1`:

- illegal instruction class vs trapped system instruction vs breakpoint
- origin EL and state from `SPSR_EL1`

12.3 Access Violations

Access violations are faults caused by illegal memory access. They typically appear as:

- **Data abort**: illegal load/store (common)
- **Instruction abort**: illegal instruction fetch / execute permission violation (common with bad function pointers)

Common causes

- null pointer dereference
- use-after-free (address still mapped sometimes, but permissions or mapping may change)
- stack overflow or guard-page hit

- writing to a read-only mapping (e.g., COW/RO pages)
- executing from non-executable memory (NX/execute permission fault)

Example: write to a read-only location (conceptual)

```
/* EL0: store through an invalid or protected pointer => data abort.
 ↪  */
.text
.global demo_store_violation
.type demo_store_violation, %function
demo_store_violation:
    /* Suppose x0 points to a read-only mapping or unmapped page */
    mov     x1, #123
    str     x1, [x0]          /* data abort if not writable */
    ret
```

Example: execute from a non-executable page (conceptual)

```
/* EL0: branch to an address that is not executable or not mapped. */
.text
.global demo_execute_violation
.type demo_execute_violation, %function
demo_execute_violation:
    br      x0                /* may cause instruction abort */
```

The minimum evidence

For aborts, Linux relies on:

- `FAR_EL1`: faulting virtual address

- `ELR_EL1`: address of faulting instruction
- `ESR_EL1`: abort attributes (what kind of abort)

12.4 How Linux Converts Exceptions into Signals

Linux uses hardware exceptions as input events and converts many of them into **POSIX signals** for user space. The conversion is policy-driven: some exceptions are recoverable, some are fatal, and some are delivered to a debugger.

Signal mapping (common, programmer-facing)

Typical outcomes for EL0-origin exceptions:

- **SIGSEGV**: invalid memory access, protection fault (data/instruction abort that cannot be resolved)
- **SIGBUS**: certain address errors or bus-related faults (platform/condition dependent)
- **SIGILL**: illegal instruction / undefined instruction / prohibited execution
- **SIGTRAP**: breakpoint or debug trap events (e.g., `BRK`), or single-step

Conceptual kernel decision tree

1. Exception enters EL1; kernel reads `ESR_EL1`, `ELR_EL1`, and possibly `FAR_EL1`.
2. Kernel determines whether it can **fix up** the condition:
    - demand paging: map in the page, update page tables, retry/continue
    - copy-on-write: create a private writable page, retry/continue

3. If not fixable in user context, kernel prepares a **signal frame** on the user stack and schedules signal delivery.

4. Kernel returns to EL0 via `ERET`; on return, user-space observes the signal handler or default action.

Example: user signal handler for SIGSEGV (C)

```c
/* Demonstration: install a SIGSEGV handler (simplified). */
#include <signal.h>
#include <stdint.h>
#include <unistd.h>

static void on_segv(int sig) {
    const char msg[] = "SIGSEGV received\n";
    (void)sig;
    write(2, msg, sizeof(msg)-1);
    _exit(128 + SIGSEGV);
}

int main(void) {
    signal(SIGSEGV, on_segv);
    volatile int *p = (int*)0;
    return *p; /* triggers a fault that becomes SIGSEGV */
}
```

Example: trap to debugger vs signal

```
/* BRK may be handled by a debugger, or converted to SIGTRAP if no
 ↪  debugger intercepts. */
.text
.global demo_brk_signal_path
```

```
.type demo_brk_signal_path, %function
demo_brk_signal_path:
    brk     #0
    ret
```

Practical diagnosis rule

When a Linux process dies with a signal, interpret it as:

- a kernel policy decision made after reading architectural exception evidence
- not “random behavior”: the original cause is always visible through the exception class and fault address (when applicable)

Summary

- Syscalls and page faults both enter EL1 via synchronous exceptions; they differ in cause and evidence.
- Illegal instructions are classified by syndrome and typically map to SIGILL or SIGTRAP.
- Access violations are aborts; FAR/ELR/ESR form the minimum proof set.
- Linux converts unresolvable user-space exceptions into signals and delivers them by constructing a signal frame and returning to EL0.

Chapter 13

Debugging and Observing Exceptions

13.1 Using Disassembly to Trace Exceptions

When an exception happens, your most reliable anchor is the **faulting/triggering instruction address**. In Linux user-space failures, you typically observe an address through:

- a crash report or debugger stop at an instruction pointer (PC)
- a kernel log (for fatal paths)
- a signal handler context (`ucontext`) when installed

The disassembly workflow (disciplined)

1. Identify the **PC** where execution stopped (or the address reported).
2. Disassemble the region around that address.
3. Classify what the instruction is doing:

- `svc` ⇒ syscall path
- `brk` ⇒ debug trap / SIGTRAP path
- `ldr/str/ldp/stp` with a suspicious base register ⇒ likely data abort
- indirect `br xN` / `blr xN` ⇒ bad function pointer can become instruction abort

4. Confirm by reading architectural evidence (ESR/FAR) when available (kernel or low-level environment).

Example: syscall is visible in disassembly

```
/* Disassembly-like snippet: syscall path is explicit. */
.text
.global demo_disasm_syscall
.type demo_disasm_syscall, %function
demo_disasm_syscall:
    mov     x8, #172
    svc     #0                /* the exception trigger is visible */
    ret
```

Example: a crash often points at the memory instruction

```
/* If this faults, the PC usually lands on the LDR/STR itself. */
.text
.global demo_disasm_fault
.type demo_disasm_fault, %function
demo_disasm_fault:
    ldr     x1, [x0]          /* data abort if x0 is invalid */
    add     x1, x1, #1
    ret
```

Example: bad indirect branch (common for instruction aborts)

```
/* If x0 is not a valid executable address, BR can trigger
 ↪  instruction abort. */
.text
.global demo_bad_indirect_branch
.type demo_bad_indirect_branch, %function
demo_bad_indirect_branch:
    br      x0
```

What disassembly gives you

- It tells you **what kind of operation** was attempted at the failure point.
- It helps you reason about **which register values matter** (base register, index, function pointer register).

13.2 Reading ESR and FAR for Diagnosis

When you have access to low-level exception state (kernel, hypervisor, firmware, or a controlled lab environment), the minimum evidence set is:

> **ESR_ELx** (what) + **ELR_ELx** (where) + **FAR_ELx** (which address, if any) + **SPSR_ELx** (context)

13.2.1 Practical interpretation flow

1. Read `ESR_ELx` and extract the **exception class**:
    - SVC ⇒ syscall

- Data abort / instruction abort ⇒ fault analysis
- Breakpoint / debug ⇒ trap analysis
- Illegal instruction ⇒ SIGILL-like behavior in OS context

2. If class is abort, read `FAR_ELx` and treat it as the **faulting virtual address**.

3. Use `ELR_ELx` to locate the responsible instruction and disassemble around it.

Example: capture ESR/FAR/ELR at EL1

```
/* EL1: capture evidence into registers for logging or structured
 ↪  dispatch. */
.text
.global el1_read_fault_evidence
.type el1_read_fault_evidence, %function
el1_read_fault_evidence:
    mrs     x0, esr_el1         /* class + attributes */
    mrs     x1, elr_el1         /* faulting/triggering instruction
     ↪  address */
    mrs     x2, far_el1         /* fault address (abort classes) */
    mrs     x3, spsr_el1        /* saved context */
    ret
```

A strict rule

Never use `FAR_ELx` unless `ESR_ELx` indicates an abort class where FAR is architecturally meaningful.

13.3 Common Exception Patterns in User Programs

This section lists the patterns that dominate real crashes and debug sessions.

13.3.1 Pattern 1: Null pointer or wild pointer load/store

- Exception: data abort
- Symptom: SIGSEGV (often), PC points to `ldr/str`
- FAR: near 0 (null) or a suspicious address (wild)

```
/* Null dereference. */
.text
.global pat_null_deref
.type pat_null_deref, %function
pat_null_deref:
    mov     x0, #0
    ldr     x1, [x0]
    ret
```

13.3.2 Pattern 2: Use-after-free / stale pointer

- Exception: data abort or silent corruption (if mapping still valid)
- Symptom: crashes appear “random” because address may sometimes remain mapped

13.3.3 Pattern 3: Execute permission / bad function pointer

- Exception: instruction abort
- Symptom: crash at indirect branch (`br xN` / `blr xN`)

```
/* Bad function pointer: x0 points to non-code or unmapped region. */
.text
.global pat_bad_funcptr
.type pat_bad_funcptr, %function
pat_bad_funcptr:
    blr     x0
    ret
```

13.3.4 Pattern 4: Stack overflow into guard page

- Exception: data abort on stack access
- Symptom: SIGSEGV with FAR near the stack guard boundary

13.3.5 Pattern 5: Intentional traps (debug) mistaken for crashes

- Exception: breakpoint/debug exception
- Symptom: SIGTRAP or debugger stop

```
/* Intentional debug stop. */
.text
.global pat_debug_brk
.type pat_debug_brk, %function
pat_debug_brk:
    brk     #0
    ret
```

13.3.6 Pattern 6: Syscall misuse without libc

- Exception: not necessarily; most errors appear as negative return codes

- Symptom: syscall returns `-errno` but the program treats it as success

```
/* Bug: ignores negative x0 error return from syscall. */
.text
.global pat_syscall_ignore_error
.type pat_syscall_ignore_error, %function
pat_syscall_ignore_error:
    mov     x8, #64           /* write */
    svc     #0
    /* BUG: assumes x0 >= 0 without checking */
    ret
```

13.4 Typical Bugs and Misunderstandings

These are the misunderstandings that repeatedly block correct reasoning for AArch64 exceptions and syscalls.

13.4.1 Bug 1: Confusing syscalls with faults

- **Syscall**: deliberate `svc`
- **Fault**: abort/illegal instruction/debug trap

Disassembly resolves this quickly: if the triggering instruction is `svc`, it is not a crash by itself.

13.4.2 Bug 2: Treating FAR as always valid

FAR is meaningful for abort classes, not for SVC/BRK classes. Always gate FAR usage by ESR class.

13.4.3 Bug 3: Thinking “exception = interrupt”

An exception is the umbrella mechanism. Interrupts (IRQ/FIQ) are only one subset and are typically asynchronous. Most user crashes are **synchronous** (abort/illegal instruction) not IRQ/FIQ.

13.4.4 Bug 4: Assuming function-call register rules apply to syscalls

Syscalls are a kernel ABI, not the C ABI. If you write raw syscalls, obey: `X8` (number), `X0--X5` (args), `X0` (result), negative values indicate errors.

13.4.5 Bug 5: Misreading the fault point for instruction aborts

With bad indirect branches, the faulting instruction may be `br`/`blr`, but the **root cause** is the corrupted function pointer value in the source register.

A concise debugging checklist

- Identify the faulting PC and disassemble around it.
- If available, read `ESR` to classify (syscall vs abort vs illegal vs debug).
- If abort, read `FAR` and treat it as the faulting VA.
- Trace the register that produced the address (base/index for loads/stores; target reg for indirect branches).
- Separate root cause (bad pointer value) from symptom (fault at `ldr/str` or `br/blr`).

Chapter 14

Performance and Design Considerations

14.1 Cost of Exceptions and Syscalls

An exception is a **control-plane event**: it forces the CPU to stop normal instruction flow, switch context, and run privileged code. Even when the handler is efficient, exceptions and syscalls have unavoidable costs:

- **Pipeline disruption**: exception entry breaks sequential execution and prediction.
- **Privilege transition overhead**: EL0 $\rightarrow$ EL1 entry and EL1 $\rightarrow$ EL0 return via `ERET`.
- **State saving/restoring**: trap frames, register spill/fill, and kernel bookkeeping.
- **Cache and TLB effects**: kernel entry touches different code/data footprints.
- **Security mitigations and checks**: pointer validation, permission checks, and policy enforcement.

Two classes of overhead

- **Architectural overhead**: vectoring, ELR/SPSR management, `ERET` semantics.
- **Kernel policy overhead**: scheduling points, tracing/auditing, signal checks, memory management.

Minimal syscall loop (shows the boundary cost)

```
/* EL0: repeated syscalls force repeated exception transitions. */
.text
.global tight_syscall_loop_concept
.type tight_syscall_loop_concept, %function
tight_syscall_loop_concept:
    mov     x8, #172          /* __NR_getpid */
    mov     x9, #1000
1:
    svc     #0
    subs    x9, x9, #1
    b.ne    1b
    ret
```

This pattern is functionally correct but performance-hostile: each iteration pays the full exception entry/return cost.

14.2 Fast Paths vs Slow Paths

Not all syscalls and exceptions have the same effective cost. A good mental model:

- **Fast path**: simple checks, no blocking, no major memory management work, minimal bookkeeping.

- **Slow path**: involves scheduling, blocking, page faults, heavy validation, device interaction, or complex subsystem work.

14.2.1 Fast-path examples (typical)

- reading cached process metadata (where applicable)
- syscalls that complete without blocking or complex resource management
- fault handling that resolves quickly (e.g., page already present but needs minor permission fixup in some policies)

14.2.2 Slow-path examples (typical)

- syscalls that block on I/O (disk/network)
- syscalls that trigger scheduling decisions (sleep, futex contention, waits)
- page faults that require disk I/O (demand paging)
- copy-on-write faults that allocate and copy pages

A page fault can dominate a syscall cost

A syscall without faults may be small compared to a syscall that touches memory and triggers:

- page-in from storage
- major TLB and cache churn

Conceptual illustration: syscall + possible page fault

```
/* EL0: write may fault if buf points to an unmapped page (then slow
 ↪  path). */
.text
.global write_may_fault_concept
.type write_may_fault_concept, %function
write_may_fault_concept:
    /* x0=fd, x1=buf, x2=len */
    mov     x8, #64           /* __NR_write */
    svc     #0
    ret
```

If the kernel must fault-in user pages while copying from `buf`, the effective latency becomes a slow-path event.

14.3 Avoiding Excessive Syscalls

The primary performance rule for system programming is:

> **Amortize syscalls. Do more work per entry.**

14.3.1 Common syscall-amplification mistakes

- reading/writing one byte at a time using `read`/`write`
- opening and closing files repeatedly inside hot loops
- using frequent time queries or process queries in tight loops
- using many small `mmap`/`munmap` operations instead of pooling

14.3.2 Better patterns (high-level)

- **Batch I/O**: larger buffers, fewer `read`/`write` calls.
- **Reuse resources**: keep file descriptors open when appropriate.
- **Event-driven designs**: wait on multiple events rather than polling.
- **Use user-space caching**: avoid repeated queries for stable data.

Example: bad vs good syscall granularity

```
/* BAD: many syscalls (conceptual). */
ssize_t write_bytewise(int fd, const unsigned char* p, size_t n) {
    for (size_t i = 0; i < n; ++i) {
        /* each call crosses EL0->EL1 */
        if (write(fd, &p[i], 1) != 1) return -1;
    }
    return (ssize_t)n;
}
```

```
/* GOOD: amortize syscalls by batching. */
ssize_t write_buffered(int fd, const unsigned char* p, size_t n) {
    /* one syscall for many bytes (or a small number of large syscalls) */
    return write(fd, p, n);
}
```

14.3.3 Direct syscalls vs libc wrappers

Most user programs should use libc wrappers because they:

- implement correct errno translation and edge-case handling
- may apply optimized strategies (e.g., vDSO paths where applicable for some operations)

Direct syscalls are justified in low-level runtimes, special sandboxes, or educational labs—but should not be the default for application code.

Raw syscall batching example (conceptual)

```
/* Prefer one write syscall for many bytes over many small writes. */
.text
.global write_once_concept
.type write_once_concept, %function
write_once_concept:
    /* x0=fd, x1=buf, x2=len */
    mov     x8, #64           /* __NR_write */
    svc     #0
    ret
```

14.4 Exception-Aware System Design

Exception-aware design is the discipline of shaping software so that:

- exceptions are **rare in hot paths**
- when exceptions happen, they are **predictable and recoverable**
- the system preserves correctness and isolation under fault conditions

14.4.1 Design principle 1: Separate normal flow from exceptional flow

- Syscalls are part of normal flow, but treat them as boundary crossings.
- Faults (aborts, SIGSEGV, SIGILL) must remain exceptional; if they become common, your design is broken.

14.4.2 Design principle 2: Avoid using faults for control flow

Do not rely on SIGSEGV or page faults as a normal conditional branch mechanism in application logic. Page faults are for memory management; signals are for error containment and notification.

14.4.3 Design principle 3: Plan for restartable boundaries

In robust systems, a small set of boundaries must be restartable or fail-safe:

- I/O operations (retry strategy, idempotency awareness)
- memory allocation pressure (fallback, pooling)
- concurrency primitives (timeouts, cancellation)

14.4.4 Design principle 4: Prefer fewer, well-defined transitions

- Reduce the number of kernel crossings in performance-critical loops.
- When a crossing is unavoidable, do the maximum useful work per crossing.

14.4.5 Design principle 5: Measure where exceptions occur

For performance and reliability, you should be able to answer:

- Where are syscalls concentrated?
- Which calls trigger blocking or page faults?
- Which faults become signals (SIGSEGV/SIGBUS/SIGILL), and why?

Example: exception-aware loop structure (conceptual)

```c
/* Conceptual pattern:
   - do most work in user space
   - perform syscall only when necessary and in batches
*/
void process_stream(int fd) {
    unsigned char buf[64 * 1024];
    for (;;) {
        ssize_t n = read(fd, buf, sizeof(buf));  /* boundary crossing */
        if (n <= 0) break;
        /* heavy processing in user space */
        /* ... */
        /* occasional batched output syscall */
    }
}
```

Summary

- Exceptions and syscalls are inherently more expensive than in-process control flow.
- Fast vs slow paths are dominated by kernel work, blocking, and memory faults.
- Reduce syscall count by batching, reuse, and event-driven architecture.
- Design so faults remain exceptional, and measure boundary crossings in hot paths.

Appendices

Appendix A — Common Exception Scenarios

This appendix lists frequent user-space exception scenarios on AArch64 Linux, how they present, and how to reason about them. For each scenario, the **minimum proof set** remains: **ESR** (what) + **ELR** (where) + **FAR** (which address, if any) + user-visible signal outcome.

A.1 Null Pointer Access

Trigger: load/store through address 0 (or near 0).
Exception class: synchronous data abort (usually).
Typical user-space outcome: `SIGSEGV`.
Key evidence:

- `FAR_EL1` ≈ `0x0`
- `ELR_EL1` points at the faulting `ldr/str`
- `ESR_EL1` indicates a data abort class

Minimal repro (AArch64 user code):

```
/* Null pointer dereference: likely data abort -> SIGSEGV. */
.text
```

```
.global null_deref_demo
.type null_deref_demo, %function
null_deref_demo:
    mov     x0, #0
    ldr     x1, [x0]          /* faulting instruction */
    ret
```

Common misunderstanding:

- "It crashed on LDR, so LDR is broken."
  The instruction is correct; the base register value is invalid.

Typical root causes:

- uninitialized pointer
- pointer overwritten (buffer overflow, use-after-free corruption)
- NULL returned from an allocator or lookup and not checked

Immediate diagnostic step: inspect the base register used by the memory instruction at the faulting PC.

A.2 Stack Overflow

Trigger: stack grows into an unmapped guard page or exceeds stack mapping limits.
Exception class: synchronous data abort on a stack access.
Typical user-space outcome: `SIGSEGV` (commonly).
Key evidence:

- `FAR_EL1` near the stack boundary / guard region
- `ELR_EL1` points to an instruction touching stack memory (often in function prologue/epilogue)

Common ways this happens:

- deep or infinite recursion
- large local objects (large arrays/structs) on the stack
- alloca-like dynamic stack growth without limits

Conceptual AArch64 prologue that can fault if SP crosses a guard page:

```
/* Large stack frame allocation can fault if it crosses into an
 ↪  unmapped guard page. */
.text
.global stack_frame_demo
.type stack_frame_demo, %function
stack_frame_demo:
    sub     sp, sp, #0x40000  /* allocate 256 KB (illustrative) */
    str     x0, [sp]          /* touch stack: may fault if stack
     ↪  guard hit */
    add     sp, sp, #0x40000
    ret
```

Common misunderstanding:

- "The crash points to a store, so the store is wrong."
  The store is correct; SP points to an address that is not mapped/writable.

Immediate diagnostic steps:

- check recursion depth and large locals
- inspect SP at the crash site and compare with expected stack region

A.3 Privileged Instruction in EL0

Trigger: executing an instruction that requires EL1+ privileges from user space (EL0).
Exception class: synchronous exception (illegal/trapped instruction class).
Typical user-space outcome: `SIGILL` (often), sometimes `SIGSEGV` depending on the specific trap policy.
Key evidence:

- `ELR_EL1` points to the privileged instruction
- `ESR_EL1` indicates an illegal/trapped instruction class
- `FAR_EL1` is typically not relevant for this class

Minimal repro: privileged system register access from EL0:

```
/* Attempt to read an EL1 system register from EL0: illegal/trapped.
 ↪  */
.text
.global privileged_mrs_demo
.type privileged_mrs_demo, %function
privileged_mrs_demo:
    mrs     x0, sctlr_el1     /* privileged: not allowed in EL0 */
    ret
```

Common misunderstanding:

- "If I can write assembly, I can read any system register."
  Privileged registers are protected by the architecture; EL0 code cannot access them directly.

Correct design:

- use syscalls, device drivers, or approved interfaces for privileged operations
- if measuring CPU state, use user-allowed facilities (or privileged tooling)

A.4 Invalid Syscall Numbers

Trigger: executing `svc #0` with an undefined syscall number in `X8`.
Exception class: still an SVC synchronous exception; the kernel dispatch rejects the number.
Typical user-space outcome:

- syscall returns **negative error code** in `X0` at the raw ABI level
- libc wrappers typically translate to `-1` with `errno` set appropriately

Key evidence:

- `ESR_EL1` indicates SVC from EL0
- `ELR_EL1` points after the `svc` instruction
- No FAR usage (not an abort)

Minimal repro:

```
/* Invalid syscall number: kernel returns an error in x0. */
.text
.global invalid_syscall_demo
.type invalid_syscall_demo, %function
invalid_syscall_demo:
    mov     x8, #0x7fffffff   /* intentionally invalid syscall number
     ↪  */
    svc     #0
    /* x0 now holds a negative error code (raw kernel ABI). */
    ret
```

Correct error handling without libc:

```
/* Convert negative x0 to a positive errno-like value in x0
 ↪  (conceptual). */
.text
.global normalize_syscall_error_demo
.type normalize_syscall_error_demo, %function
normalize_syscall_error_demo:
    mov     x8, #0x7fffffff
    svc     #0
    cmp     x0, #0
    b.ge    1f
    neg     x0, x0              /* x0 = errno (positive) */
1:
    ret
```

Common misunderstanding:

- "Invalid syscall should crash the process."
  It usually does not. It is a normal syscall failure handled by the kernel dispatcher.

Summary (Operational)

- Null pointer and stack overflow are typically **data aborts** → signals (often `SIGSEGV`).
- Privileged instructions in EL0 are typically **illegal/trapped instruction exceptions** → `SIGILL`/`SIGTRAP` depending on context.
- Invalid syscall numbers are **normal syscall failures**: check return values; do not expect a crash.

Appendix B — Exception Flow Diagrams (Conceptual)

This appendix provides conceptual flow diagrams for the most common exception-driven control paths on AArch64. These are **architecture-accurate at the control-flow level** while remaining OS/firmware agnostic.

B.1 EL0 → EL1 Syscall Flow

```
EL0 User Code
  |
  |  (1) Prepare syscall ABI
  |      X8 = syscall number
  |      X0..X5 = arguments
  v
SVC #0  (synchronous exception)
  |
  |  (2) Hardware exception entry
  |      ELR_EL1  = return PC (after SVC)
  |      SPSR_EL1 = saved PSTATE (EL0 state)
  |      ESR_EL1  = SVC class + details
  v
EL1 Vector Table (VBAR_EL1 + Sync-from-lower-EL entry)
  |
  |  (3) Kernel entry prologue
  |      - switch to trusted EL1 stack (policy)
  |      - save user regs into trap frame (policy)
  v
Syscall Dispatcher
```

```
  |
  |  (4) Validate + dispatch by X8
  |      - check syscall number range
  |      - call syscall handler
  v
Syscall Handler (Kernel)
  |
  |  (5) Produce result
  |      X0 = return value OR -errno (raw kernel ABI)
  v
Kernel return-to-user path
  |
  |  (6) Restore user context (policy)
  |      - restore regs from trap frame
  |      - ensure ELR_EL1 / SPSR_EL1 correct
  v
ERET  (exception return)
  |
  |  (7) Hardware return
  |      PC     <- ELR_EL1
  |      PSTATE <- SPSR_EL1
  v
EL0 resumes after SVC
```

Minimal syscall trigger (EL0)

```
/* EL0: syscall entry point. */
.text
```

```
.global syscall_trigger_concept
.type syscall_trigger_concept, %function
syscall_trigger_concept:
    mov     x8, #172          /* __NR_getpid (example) */
    svc     #0
    ret
```

B.2 Fault Handling Flow

This is the dominant path for page faults, permission violations, and bad pointers in user programs.

```
EL0 User Code
  |
  |  (1) Execute memory access or instruction fetch
  v
Faulting operation
  |
  |  Examples:
  |    - load/store to unmapped VA
  |    - write to read-only page
  |    - execute from non-executable page
  v
Synchronous Abort (exception)
  |
  |  (2) Hardware exception entry to EL1
  |      ESR_EL1 = abort class + attributes
  |      ELR_EL1 = faulting instruction address
  |      FAR_EL1 = faulting virtual address (abort classes)
```

```
  |      SPSR_EL1 = saved EL0 state
  v
EL1 Vector Table (Sync-from-lower-EL entry)
  |
  |  (3) Kernel abort entry
  |      - switch to trusted stack
  |      - save minimal context
  v
Fault classification
  |
  |  (4) Decode ESR:
  |      - translation vs permission vs alignment
  |      - read vs write vs execute attributes (conceptual)
  v
Fixup attempt?
  |
  |  (5a) If fixable:
  |       - map page / handle COW / update permissions
  |       - possibly retry/continue
  |       - prepare to return to EL0
  |
  |  (5b) If not fixable:
  |       - convert to signal (SIGSEGV/SIGBUS, etc.)
  |       - build signal frame on user stack
  v
Return to EL0 (ERET)
  |
```

```
  |   (6a) If fixup succeeded: user code continues
  |   (6b) If signal pending: user sees handler or default action
  v
EL0: continue or terminate
```

Minimal fault trigger (EL0)

```
/* EL0: likely data abort by null dereference. */
.text
.global fault_trigger_concept
.type fault_trigger_concept, %function
fault_trigger_concept:
    mov     x0, #0
    ldr     x1, [x0]
    ret
```

Kernel evidence capture (EL1 conceptual)

```
/* EL1: capture abort evidence for diagnosis/dispatch. */
.text
.global el1_capture_abort_evidence_concept
.type el1_capture_abort_evidence_concept, %function
el1_capture_abort_evidence_concept:
    mrs     x0, esr_el1         /* abort class + attributes */
    mrs     x1, elr_el1         /* faulting instruction address */
    mrs     x2, far_el1         /* fault VA (abort classes) */
    mrs     x3, spsr_el1        /* saved state */
    ret
```

B.3 Secure World Transition Overview

AArch64 systems that implement TrustZone conceptually split execution into:

- **Non-secure world**: normal OS and applications
- **Secure world**: trusted firmware/services (policy-dependent)

The transition is typically orchestrated by **EL3** (Secure Monitor). The common conceptual gateway is an `SMC` instruction.

```
Non-secure world (EL0/EL1/EL2)
  |
  |  (1) Request secure service
  |      - OS/firmware-defined calling convention
  v
SMC (Secure Monitor Call)
  |
  |  (2) Exception entry to EL3 (secure monitor)
  |      - save return state into ELR_EL3 / SPSR_EL3
  |      - set up secure context routing
  v
EL3 Secure Monitor
  |
  |  (3) Validate request + dispatch secure service
  |      - may switch to secure EL1 runtime or secure payload
  v
Secure payload / service (platform-defined)
  |
  |  (4) Perform operation, produce result
```

```
  v
Return path to non-secure world
  |
  |  (5) Restore non-secure context
  v
ERET (from EL3)
  |
  |  (6) Hardware restores PC/PSTATE to resume non-secure execution
  v
Non-secure world resumes
```

Minimal secure monitor call trigger (conceptual)

```
/* Conceptual: SMC requests a secure monitor service
 ↪  (platform-defined semantics). */
.text
.global smc_trigger_concept
.type smc_trigger_concept, %function
smc_trigger_concept:
    smc     #0                  /* secure monitor call */
    ret
```

Operational notes (design discipline)

- **SMC is not a Linux syscall**. It is a firmware/secure monitor interface.
- The calling convention and service IDs are platform-defined.
- The architectural mechanics remain: exception entry captures return state and returns via ERET.

Appendix C — Preparation for Advanced Topics

This appendix prepares you for advanced AArch64 exception and privilege topics that extend beyond the core EL0→EL1 syscall and fault model. The goal is to establish the minimum conceptual vocabulary and practical readiness for EL2 virtualization, EL3 secure monitor flows, and bare-metal exception handling.

C.1 Virtualization and EL2

What changes when EL2 exists:

- EL2 can intercept and virtualize events that would otherwise be handled by EL1.
- Some exceptions and privileged operations are **trapped to EL2** depending on configuration.
- The system now has a **host/guest model**: guest OS runs at EL1 (or a virtualized EL1 view), while the hypervisor runs at EL2.

Why a systems programmer cares:

- performance: extra trap layers can affect syscall/interrupt latency under virtualization
- correctness: assumptions about privileged operations may fail if trapped/emulated
- debugging: an exception you think is "kernel" may be handled by the hypervisor first

Conceptual trap flow (guest perspective):

```
Guest EL0 -> Guest EL1 (kernel) does privileged operation
    |
    | (if configured to trap)
    v
```

```
EL2 Hypervisor trap handler
    |
    | emulate / deny / forward
    v
Return to guest context (ERET from EL2)
```

Minimal readiness checklist:

- understand EL0/EL1 exception entry/return (`ELR_ELx`, `SPSR_ELx`, `ESR_ELx`)
- recognize that "same instruction" can have different outcomes under trapping
- practice reading exception class and origin to detect virtualization involvement

C.2 Secure Monitor Calls (SMC)

SMC in one sentence: `SMC` is a privileged gateway into the **secure monitor** (EL3) for TrustZone-enabled systems.

What it is not:

- not a Linux syscall
- not a normal function call

Conceptual model:

- Non-secure world requests a secure service using `SMC`
- EL3 secure monitor validates and dispatches
- returns via `ERET` to the originating world/context

Minimal conceptual trigger:

```
/* Conceptual secure monitor call. Service IDs and calling convention
 ↪  are platform-defined. */
.text
.global smc_call_concept
.type smc_call_concept, %function
smc_call_concept:
    smc     #0
    ret
```

Readiness checklist:

- distinguish SVC (OS syscall) from SMC (secure monitor)
- understand that EL3 owns secure/non-secure transitions
- treat SMC calling convention as **platform/firmware-defined**, not universal

C.3 Exception Handling in Bare-Metal Systems

Bare-metal exception handling uses the same architectural exception mechanism, but the **policy is yours**: there is no Linux kernel to build trap frames, deliver signals, or manage user/kernel separation.

What you must implement explicitly:

- vector table placement and `VBAR_ELx` initialization
- stack selection and reliable stacks per EL (and often per exception type)
- context save/restore policy (GPRs, SIMD/FP if used)
- fault reporting (UART/logging), halt, or recovery strategy

Minimal EL1 bare-metal pattern (conceptual):

```
/* Bare-metal style: install EL1 vectors and provide a minimal
 ↪  synchronous handler stub. */
.text
.align  11
.global vectors_el1_bare_concept
vectors_el1_bare_concept:
    b       el1_sync_sp0_bare_concept
    b       el1_irq_sp0_bare_concept
    b       el1_fiq_sp0_bare_concept
    b       el1_serror_sp0_bare_concept
    /* Remaining entries omitted in this conceptual snippet */

.global el1_sync_sp0_bare_concept
.type el1_sync_sp0_bare_concept, %function
el1_sync_sp0_bare_concept:
    /* Save minimal state */
    stp     x0, x1, [sp, #-16]!
    /* Capture evidence */
    mrs     x0, esr_el1
    mrs     x1, elr_el1
    mrs     x2, far_el1
    /* Policy: log/halt/recover (not shown) */
1:
    b       1b
```

Readiness checklist:

- be able to write a correct vector table skeleton and set `VBAR_EL1`
- enforce stack discipline (alignment, safe prologues, minimal early code)
- define a deterministic policy for unrecoverable exceptions (stop vs reset vs safe-mode)

C.4 Cross-Reference to Upcoming Booklets

This booklet established the core exception and syscall model (EL0→EL1). The following advanced areas build directly on it:

- **Virtualization (EL2)**: traps, hypercalls, stage-2 translation, virtual interrupts, and guest/host exception routing.
- **TrustZone and EL3**: secure monitor design, world switching, secure payload invocation, and secure boot integration.
- **Bare-metal exception engineering**: vector table engineering, per-CPU stacks, minimal handler design, and crash-dump strategies.
- **Kernel deep dive**: full Linux AArch64 entry/exit paths, syscall table mechanics, fast return paths, and tracing hooks.

Practical next steps

- Implement a minimal EL1 vector table in a lab environment and deliberately trigger:
  - `svc` path (if OS-like environment exists)
  - data abort via invalid memory access
  - `brk` for debug trap observation
- Practice evidence-first debugging: `ESR` + `ELR` + `FAR` + saved state.
- Separate architecture from policy: architecture defines entry/return registers; your OS/firmware defines save/restore and recovery.