System Calls and Operating-System Structures

Week 2: Interfaces, crossing costs, protection, and kernel organization

SDB

Autumn 2026

Intermediate/graduate engineering lecture set

Learning outcomes

By the end of this lecture set, you should be able to:

1. distinguish source-level APIs, library wrappers, ABIs, and kernel syscalls;
2. trace syscall entry, validation, blocking, restart, and return paths;
3. write robust Unix code that handles errors, partial I/O, and descriptor lifetime;
4. reason about syscall security, capability-like handles, and attack surfaces;
5. compare monolithic, layered, microkernel, hybrid, library-OS, and unikernel designs; and
6. evaluate architecture using crossing cost, fault containment, TCB size, and evolvability.

Guiding question

Where should an operating-system service execute, and what must cross each protection boundary?

Road map

1. Interfaces: API, ABI, and System Calls
2. Robust Unix Programming Patterns
3. Performance, Observability, and Security
4. Operating-System Structures
5. Synthesis

Road map

1. Interfaces: API, ABI, and System Calls
2. Robust Unix Programming Patterns
3. Performance, Observability, and Security
4. Operating-System Structures
5. Synthesis

Four interfaces that are often conflated

Layer	Contract	Example
Source API	names, types, and language-visible behavior	POSIX `read()`, C `fread()`
Library implementation	buffering, policy, compatibility, wrapper logic	libc translates `errno`; `fread()` buffers
User–kernel ABI	registers, stack, calling convention, syscall numbers, layouts	Linux x86-64 syscall ABI
Kernel object interface	validated operations on kernel-managed objects	file operation, VFS lookup, socket send

Precision

A library function is not necessarily a syscall. `printf()`, `strerror()`, and `execlp()` are library functions; wrappers may issue zero, one, or multiple syscalls, and the exact syscall can differ across libc/kernel versions.

What a system call actually provides

A syscall is a controlled entry into a privileged kernel service. It provides:

- a stable dispatch convention and argument/result representation;
- authority checks against the caller's credentials and handles;
- safe transfer of data across address spaces;
- blocking/wakeup integration with the scheduler;
- accounting, auditing, tracing, and cancellation/restart semantics.

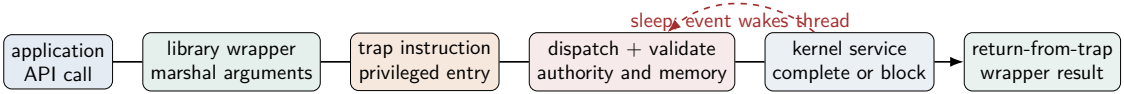
Typical classes

- process/thread lifecycle;
- files, directories, and memory maps;
- IPC, networking, synchronization;
- time, identity, limits, and metadata;
- device- and policy-specific control.

Not a taxonomy law

Categories overlap: a socket is both a file descriptor and an IPC endpoint; `mmap()` is both memory management and file I/O.

System-call lifecycle



- Entry changes privilege within the *same thread*.
- Blocking makes the thread non-runnable; the scheduler may run another thread.
- Errors are ordinary results at the ABI; libc commonly maps a negative error convention to `-1` and `errno`.
- Return may be partial even without an error.

Example ABI: Linux on x86-64

Conceptual register convention

syscall number	`rax`
arguments 1–6	`rdi, rsi, rdx, r10, r8, r9`
result	`rax`
entry instruction	`syscall`

CPU/kernel entry code also handles:

- transition to a protected kernel execution context;
- saving enough architectural state to resume;
- speculation/entry mitigations required by the platform;
- dispatch through the syscall table or equivalent mechanism.

Architecture-specific

AArch64 uses different registers and `svc`; 32-bit x86 and compatibility ABIs differ again. Never hard-code an ABI when a maintained library interface suffices.

Why wrappers matter

They handle calling conventions, symbol/version compatibility, cancellation points, `errno`, and alternate fast paths.

Validation: user pointers are untrusted capabilities to memory

Kernel code must treat every argument as adversarial:

- syscall number, flags, lengths, alignment, and overflow;
- descriptor/handle validity and requested rights;
- pointer range, access direction, and faultability;
- object lifetime, concurrent mutation, and namespace context;
- policy checks from credentials, capabilities, ACLs, or labels.

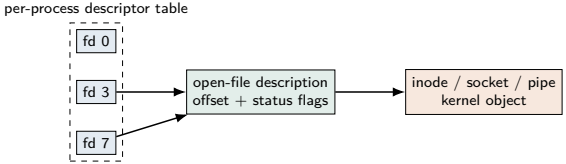
Copying is a semantic boundary

Kernels use guarded primitives such as conceptual `copy_from_user`. A range check alone is insufficient: the page can fault, mappings can change, and user data can be modified concurrently.

TOCTOU

Do not validate mutable user memory and later trust it unchanged. Copy once into kernel-owned memory or use a protocol that pins/locks and defines concurrency.

File descriptors: names for kernel objects and authority



- A descriptor indexes a process table; it is not the file itself.
- `dup()` creates another descriptor referencing the same open-file description, including file offset.
- `fork()` inherits descriptors; `execve()` preserves them unless close-on-exec is set.
- Descriptor possession conveys authority to invoke allowed operations on the referenced object.

Common failure

Leaked descriptors can keep files, pipes, sockets, mounts, or privileges alive and can cross an `execve()` boundary unexpectedly.

Blocking, interruption, restart, and partial completion

Outcome	Meaning and obligation
success	requested operation completed; returned count/value still matters
partial success	some bytes/items completed; advance buffers and retry only if semantics permit
`EINTR`	an interrupting signal was handled before completion; restart behavior depends on call, flags, and signal setup
`EAGAIN`	nonblocking operation cannot progress now; wait for readiness or retry under a policy
short read / EOF	fewer bytes can be valid; zero from a regular stream read commonly indicates EOF
blocking	thread sleeps on a wait queue until state changes, timeout expires, or cancellation/signal occurs

Robustness rule

"No error" does not imply "all requested work completed." Design loops around the API's precise progress and cancellation contract.

Road map

1. Interfaces: API, ABI, and System Calls
2. Robust Unix Programming Patterns
3. Performance, Observability, and Security
4. Operating-System Structures
5. Synthesis

Robust writes: handle progress and interruption

```c
#include <errno.h>
#include <stddef.h>
#include <unistd.h>

int write_all(int fd, const void *buf, size_t len) {
    const unsigned char *p = buf;
    while (len != 0) {
        ssize_t n = write(fd, p, len);
        if (n > 0) { p += (size_t)n; len -= (size_t)n; continue; }
        if (n < 0 && errno == EINTR) continue;
        return -1;                  // includes EPIPE, EAGAIN, device error
    }
    return 0;
}
```

- `ssize_t`, not `int`, represents byte counts/errors.
- A successful `write()` can be short.
- Nonblocking descriptors need readiness integration, not a busy retry loop.
- Broken pipes involve `SIGPIPE`/`EPIPE` policy.

Safe file copy: descriptors, errors, and cleanup

```c
int in = open(src, O_RDONLY | O_CLOEXEC);
if (in < 0) fail("open source");
int out = open(dst, O_WRONLY | O_CREAT | O_TRUNC | O_CLOEXEC, 0644);
if (out < 0) { int e = errno; close(in); errno = e; fail("open dst"); }

unsigned char buf[64 * 1024];
for (;;) {
    ssize_t n = read(in, buf, sizeof buf);
    if (n > 0) { if (write_all(out, buf, (size_t)n) < 0) fail("write"); }
    else if (n == 0) break;
    else if (errno != EINTR) fail("read");
}
if (close(out) < 0) fail("close output"); // delayed write error possible
if (close(in) < 0) fail("close input");
```

For security-sensitive replacement, also reason about symlink races, file mode/ownership, atomic rename, durability, sparse files, metadata, and whether source and destination name the same object.

fork()–execve()–waitpid(): corrected pattern

```c
pid_t pid = fork();
if (pid < 0) fail("fork");
if (pid == 0) {
    execlp("ls", "ls", "-l", (char *)NULL);
    static const char msg[] = "exec failed\n";
    (void)write(STDERR_FILENO, msg, sizeof msg - 1);
    _exit(127);
}

int status;
while (waitpid(pid, &status, 0) < 0) {
    if (errno != EINTR) fail("waitpid");
}
```

- `fork()` returns twice; `exec` replaces the child's program image and returns only on failure.
- Use `_exit()` in the failed child path to avoid flushing inherited stdio buffers or running parent cleanup handlers.
- In a multithreaded process, only async-signal-safe operations are permitted between `fork()` and `exec`; `posix_spawn()` is often safer.

Pipe discipline: EOF depends on closing every writer

```c
int p[2];
if (pipe2(p, O_CLOEXEC) < 0) fail("pipe2");
pid_t pid = fork();
if (pid == 0) {
    close(p[1]);
    if (dup2(p[0], STDIN_FILENO) < 0) _exit(126);
    close(p[0]);
    execlp("wc", "wc", "-w", (char *)NULL);
    _exit(127);
}
close(p[0]);
if (write_all(p[1], "hello world\n", 12) < 0)
    fail("pipe write");
close(p[1]);                    // lets wc observe
    EOF
```

- A pipe is a byte stream with bounded kernel buffering.
- Reads block while the buffer is empty and at least one write end remains open.
- The reader sees EOF only after the buffer drains and *all* write references close.
- Writes up to `PIPE_BUF` have defined atomicity properties; larger writes can interleave.

Memory mapping is an interface to the page cache and VM

- `mmap()` creates a mapping; it does not necessarily read file data immediately.
- First access can fault pages into memory.
- `MAP_PRIVATE` writes use copy-on-write; `MAP_SHARED` can modify the shared mapping/file.
- Access beyond the valid backing object can raise a fault such as `SIGBUS`.

Unsafe original pattern

Mapping and writing a fixed 100 bytes without checking file length, `MAP_FAILED`, or short output can access invalid pages or emit unrelated data.

Use when

Random access, shared-memory protocols, or avoiding explicit copies helps enough to justify page-fault and coherence complexity.

Signals: asynchronous delivery changes control flow

Correct design normally uses:

- `sigaction()`, not legacy `signal()`, for defined control;
- blocked signal masks while installing handlers and updating shared state;
- only async-signal-safe functions inside handlers;
- `sigsuspend()`, `sigwaitinfo()`, `signalfd()`, or an event loop to avoid check-then-sleep races.

Classic lost wakeup

"Check flag; then call `pause()`" races with a signal arriving between the check and sleep. Atomically replace the signal mask while waiting.

Handler rule

Do minimal work: record state using an appropriate atomic/signal-safe mechanism or write to a self-pipe, then handle complexity in normal control flow.

Road map

1. Interfaces: API, ABI, and System Calls
2. Robust Unix Programming Patterns
3. Performance, Observability, and Security
4. Operating-System Structures
5. Synthesis

Where syscall cost comes from

Direct cost

- wrapper and trap/return instructions;
- entry bookkeeping and mitigations;
- dispatch, validation, copying, accounting;
- locks and kernel data-structure traversal.

Indirect/dominant cost

- cache/TLB disruption;
- blocking and scheduling latency;
- page faults or device/network queues;
- contention, write-back, and remote services.

Simple cost model

For N operations of payload B, a useful first model is

$$T \approx N\, t_{\text{fixed}} + NB\, t_{\text{per-byte}} + T_{\text{queueing}} + T_{\text{device}}.$$

Batching reduces the fixed term but can increase waiting latency and memory footprint.

Avoiding or amortizing crossings

Technique	Mechanism	Trade-off
buffering/batching	more work per call	added latency, memory, failure granularity
`readv`/`writev`	scatter/gather buffers	descriptor/vector limits; still copies in many paths
memory mapping	page-based sharing/access	page faults, coherence, invalidation complexity
vDSO	kernel-published data + user code	only suitable for carefully designed read-mostly services
asynchronous queues	submit/completion rings, e.g., `io_uring`	lifecycle, cancellation, pinning, security complexity
user-space networking/storage	map queues and poll devices	dedicates cores; shifts protection and driver burden

Optimization boundary

Removing syscall crossings does not remove the need for authorization, isolation, resource accounting, or synchronization; it relocates their mechanisms.

Observability: choose the tool for the question

- `strace`: syscall names, arguments, results, timing; perturbation can be substantial.
- `perf`: hardware/software counters, sampling, scheduling and fault statistics.
- eBPF tracing: programmable kernel probes and aggregation with safety constraints.
- application tracing: semantic spans and request IDs unavailable to the kernel alone.

Do not infer too much

A trace line shows the interface event, not necessarily physical I/O. A `read()` may hit the page cache; work may have occurred earlier through read-ahead or later through write-back.

Measurement protocol

State workload, kernel/hardware, cache state, concurrency, tracing overhead, metric definition, and latency distribution.

System-call attack surface and mediation

Defensive mechanisms include:

- least-privilege credentials and Linux capabilities;
- seccomp filters restricting syscall numbers/arguments;
- namespaces, cgroups, MAC/LSM policies, and sandbox brokers;
- descriptor passing instead of ambient pathname authority;
- memory-safe implementations, fuzzing, and formal verification.

Filtering is not authorization

Allowing `read()` says nothing about which descriptors may be read. Conversely, blocking one syscall may not remove equivalent authority through another path.

Design objective

Reduce reachable kernel code and ambient authority while preserving the minimal operations the workload needs.

Road map

1. Interfaces: API, ABI, and System Calls
2. Robust Unix Programming Patterns
3. Performance, Observability, and Security
4. Operating-System Structures
5. Synthesis

Architecture is placement plus communication

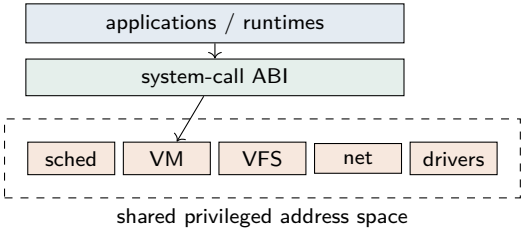
Instead of memorizing labels, characterize a design along these dimensions:

- Which services execute privileged?
- What belongs to the trusted computing base?
- Are calls procedure calls, traps, messages, or shared-memory protocols?
- Which failures can be restarted or contained?
- How are authority and names represented?
- Can components evolve independently?
- What state crosses protection domains?
- Which paths require verification or certification?

Mechanism–policy separation

Structure determines communication and trust boundaries. It does not by itself determine scheduling, caching, naming, or security policy.

Monolithic but modular kernels

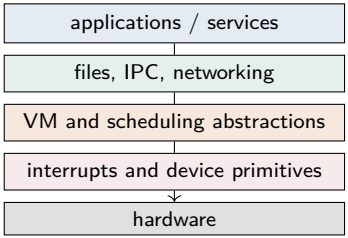


- Kernel subsystems communicate with direct calls and shared internal state.
- Loadable modules allow runtime extension but usually share kernel privilege and failure fate.
- Strengths: mature integration, low call overhead, efficient shared caches.
- Risks: large TCB/attack surface; memory corruption or driver faults can compromise the whole kernel.

Examples

Linux and BSD are commonly described as monolithic, modular kernels—not "hybrid monolithic."

Layering and information hiding

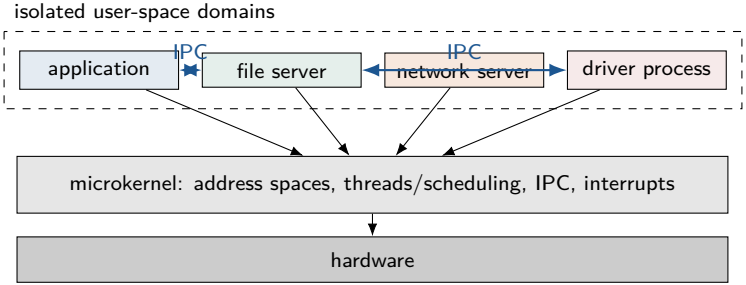


- Each layer exposes an abstraction and hides lower-level representation.
- Benefits: local reasoning, replaceable implementations, test seams, reduced coupling.
- Costs: strict layering can force redundant conversions/copies or awkward dependency inversions.
- Real kernels often use **layering with controlled bypasses**, not a perfectly linear stack.

Examples

VFS over multiple filesystems is a strong interface boundary; network stacks are layered conceptually but optimize across layers.

Microkernel: minimal privileged mechanism



Potential strengths

- smaller privileged TCB;
- fault containment/restartable services;
- explicit authority and message interfaces.

Design costs

- IPC crossings, scheduling handoffs, data transfer;
- distributed state and recovery protocols;
- performance depends strongly on IPC design and workload.

Fast IPC is more than copying bytes

For a client request to a user-space server, latency may include:

$$T_{\text{IPC}} = T_{\text{entry}} + T_{\text{validation}} + T_{\text{handoff}} + T_{\text{data}} + T_{\text{cache/TLB}} + T_{\text{queue}}.$$

Optimization techniques:

- register-based small messages;
- direct process handoff/donation;
- shared-memory data planes;
- batching and asynchronous completion;
- capability lookup integrated with IPC.

End-to-end caveat

Moving a driver into user space may add crossings but reduce recovery time, privileged code, and certification scope. Architecture evaluation must include failure and maintenance cost, not only steady-state throughput.

Hybrid kernels: a descriptive family, not a theorem

Systems commonly called hybrid retain microkernel-inspired abstractions while executing substantial services in one privileged address space for pragmatic performance or compatibility.

- Windows NT: executive subsystems and kernel-mode components above an NT kernel/HAL foundation.
- XNU: Mach-derived primitives plus BSD and I/O Kit components in the kernel.

Do not overread the label

"Hybrid" does not guarantee the best of both worlds. Ask which components are actually isolated, how they communicate, and which faults remain kernel-fatal.

Comparison rule

Compare concrete paths and trust boundaries, not marketing categories.

Beyond the classical taxonomy

Design	Core idea	Trade-off
Exokernel	securely multiplex hardware; applications choose abstractions	flexibility and specialization versus complex library OSes
Library OS	link OS services as application libraries over a host/hypervisor	per-application policy; duplicated state/services
Unikernel	compile one application with minimal OS components into an image	small footprint and attack surface; debugging/compatibility challenges
Multiserver OS	decompose services into isolated cooperating processes	restartability and least privilege; protocol/state complexity
Type-safe / language OS	use language guarantees to constrain components	smaller bug classes; toolchain/runtime trust and unsafe boundaries
Formally verified kernel	prove specified properties of the implementation	strong assurance within assumptions; specification/proof cost

These approaches can be combined with hardware virtualization, containers, capabilities, and conventional kernels.

Trusted computing base and verification boundary

TCB: components whose failure can violate the target security/correctness property.

- Smaller code size can help review and verification, but interface complexity matters too.
- Hardware, boot chain, compiler, firmware, DMA configuration, and privileged services may be in the effective TCB.
- A proof establishes a theorem about a model/specification under assumptions; it is not an all-purpose guarantee.

Example: seL4

seL4 demonstrates that strong machine-checked kernel properties are feasible. System-level assurance still requires correct user-space services, configurations, hardware assumptions, and refinement between models and deployment.

Security lesson

Minimize both privileged implementation and ambient authority reachable through its interfaces.

Architecture comparison: replace rankings with questions

	Monolithic/modular	Microkernel	Hybrid	Library OS / unikernel
Privileged scope	broad kernel services	minimal mechanisms	broad, pragmatically arranged	host/hypervisor plus linked image
Communication	direct calls/shared state	IPC/shared memory	both	local calls to linked services
Fault containment	limited inside kernel	service-domain boundaries	path-dependent	VM/process boundary-dependent
Specialization	configurable/general	replaceable servers	subsystem-specific	high per-application specialization
Key cost	TCB and fault blast radius	IPC/recovery complexity	conceptual and implementation complexity	duplication, tooling, compatibility
Best evidence	workload and fault tests	IPC plus recovery measurements	concrete component map	image/host threat and operations model

Decision principle

Choose an architecture for a specified workload, failure model, threat model, certification target, and maintenance organization—not from a universal "fastest" or "safest" ranking.

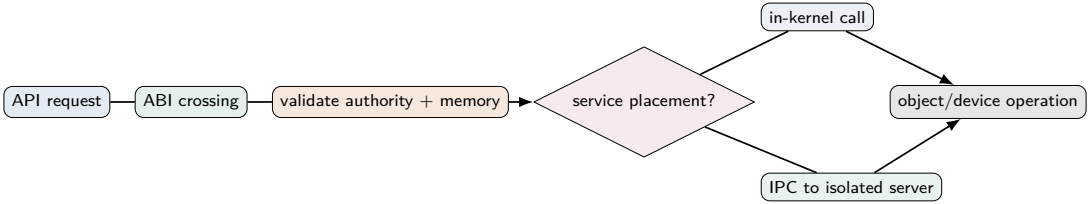
Case study matrix

System	Structural emphasis	Useful lesson	Do not claim
Linux	monolithic, modular kernel; extensive user-space services	integration and performance with a large ecosystem	modules provide fault isolation
Windows NT	hybrid family with object-based executive and subsystem history	compatibility and layered internal interfaces	all services are isolated microkernel servers
XNU/macOS	Mach + BSD + I/O Kit in a privileged kernel	architectures accrete and combine lineages	the "hybrid" label predicts every path
QNX	commercial microkernel with user-space services	fault containment and real-time deployment	microkernel alone guarantees deadlines
seL4	capability microkernel with formal verification results	explicit authority and proof-oriented design	the whole deployed system is automatically verified

Road map

1. Interfaces: API, ABI, and System Calls
2. Robust Unix Programming Patterns
3. Performance, Observability, and Security
4. Operating-System Structures
5. Synthesis

From application call to architecture choice



The recurring trade-off

Direct calls and shared state minimize crossing overhead but enlarge trust and failure domains. Isolated servers make authority and failure boundaries explicit but require IPC, state partitioning, and recovery protocols.

Misconceptions corrected

Misconception	⇒	**Precise statement**
Every libc call is a syscall.	⇒	Libraries may compute, buffer, or invoke different/multiple syscalls.
A syscall always context-switches.	⇒	It changes privilege; scheduling another thread is conditional.
Successful `read`/`write` transfers all bytes.	⇒	Short counts are valid and must drive control flow.
Linux `fork()` literally copies all memory.	⇒	The semantics are copying; implementations commonly use COW.
Microkernels are necessarily slow.	⇒	Performance depends on IPC, workload, batching, caches, and service design.
Modules isolate kernel failures.	⇒	Modules improve extensibility but usually share kernel privilege.
Hybrid means optimal compromise.	⇒	It is a descriptive label; inspect concrete placement and boundaries.
seccomp is a complete sandbox.	⇒	It restricts syscall reachability, not all object authority or resource use.

Key takeaways

- The API, library, ABI, and kernel object interface are distinct contracts.
- Syscall correctness depends on validation, explicit authority, safe copying, progress semantics, and descriptor lifetime.
- Performance is end-to-end: crossings matter, but queueing, faults, copying, caches, and devices often dominate.
- Kernel structure determines trust, communication, and failure boundaries; it does not automatically determine policy quality.
- Robust evaluation combines steady-state benchmarks, tail latency, failure injection, attack-surface analysis, and maintenance cost.

Transfer question

For any OS service, identify the promised abstraction, representation of authority, protection crossings, blocking points, recoverable failures, and measurement plan.

Laboratory: trace and explain a file copy

1. Compile a correct file-copy program with warnings and sanitizers enabled.
2. Trace it using `strace -f`; identify wrapper-visible calls, return counts, and descriptor lifetime.
3. Compare cold-cache and warm-cache runs using timing and `perf stat`.
4. Reduce syscall count using a larger buffer or vectored I/O; explain whether elapsed time changes.
5. Inject failures: missing file, permission denial, full filesystem, broken pipe, and signal interruption.

Report requirement

Distinguish observations from inferences. A syscall trace alone does not prove physical device access, durability, or a context switch.

Next week: process and thread management

- process/thread representation and lifecycle;
- creation, execution, waiting, signals, and termination;
- scheduling queues, policies, and multicore load balancing;
- context-switch state and indirect cache/TLB costs;
- hands-on process trees and scheduling observation.

Preparation

Run `ps -ef -forest`, inspect `/proc/<pid>/status` and `/proc/<pid>/fd`, then predict which attributes survive `fork()` and `execve()`.

Road map

Knowledge check: interfaces

1. Can `gettimeofday()` complete without entering kernel mode? Explain.
2. Why can libc `open()` appear as `openat()` in a Linux trace?
3. A blocking `read()` returns `-1/EINTR`. Under what conditions is retry appropriate?
4. Why is “validate pointer, then dereference later” vulnerable to TOCTOU?
5. Two descriptors created by `dup()` read from a regular file. Do they share an offset?
6. Why can a pipe reader wait forever even after the intended writer closes its descriptor?

Knowledge check: structure

1. Why does a loadable module improve modularity without necessarily improving fault isolation?
2. Name two costs beyond byte copying in a user-space server IPC path.
3. When might a user-space driver outperform its fault-isolation cost?
4. What belongs in the effective TCB of a verified-kernel deployment besides the kernel proof?
5. Why is “microkernel versus monolithic” insufficient to choose an automotive OS?

Discussion extension

Specify a workload and threat model, then draw the smallest protection-domain decomposition that meets latency, recovery, and certification goals.

Exercise: design a syscall-safe record API

A user supplies a pointer to an array of variable-length records and asks the kernel to validate and store them atomically.

1. Identify integer-overflow, nested-pointer, fault, and TOCTOU risks.
2. Propose a bounded wire format with explicit total length and offsets.
3. Decide when copying, validation, allocation, authorization, and commit occur.
4. Define partial-failure, cancellation, retry, and idempotency semantics.
5. Explain how fuzzing and property-based tests would target the interface.

Exercise: architecture for a safety controller

Requirements: bounded control-loop latency, restartable network stack, minimal certified code, device isolation, and field-updatable telemetry.

1. Place scheduler, interrupt handling, drivers, filesystems, networking, and update service into protection domains.
2. Draw IPC and shared-memory paths.
3. State the authority held by each component.
4. Bound blocking and recovery time after a driver crash.
5. Compare the proposal with a modular monolithic design.

Raw syscall demonstration: for ABI study only

```c
#include <sys/syscall.h>
#include <unistd.h>

long result = syscall(SYS_write, STDOUT_FILENO, "hello\n", 6);
```

- `syscall()` is itself a libc helper for issuing a syscall by number; it is not inline assembly here.
- It remains nonportable across OSes and sometimes architectures, and it can bypass wrapper features.
- Use it when studying or accessing a deliberately unsupported raw interface—not as a default replacement for maintained APIs.

Further study

- ABI design: compatibility syscalls, structure versioning, feature negotiation.
- High-performance interfaces: shared rings, zero-copy, polling, userspace drivers.
- Security: capability systems, syscall brokers, seccomp, confused deputies.
- Microkernels: IPC design, multiserver recovery, scheduling-context donation.
- Assurance: refinement proofs, verified compilation, information flow, WCET analysis.
- Architecture: exokernels, library OSes, unikernels, confidential VMs.

Questions?