CS 350

Operating Systems

Course Notes

University of Waterloo

Instructor: Kevin Lanctot

Spring 2026

Contents

1 Operating Systems Overview

1.1 1b: Operating Systems Overview

An **operating system** is the software layer that provides an execution environment for programs. Understanding this layer makes programmers more effective because the OS determines how software accesses the machine.

> **Note**
>
> The OS is both a **facilitator** and a **cop**:
>
> - As a facilitator, it manages resources and provides abstract interfaces to otherwise complicated hardware.
> - As a cop, it isolates running programs and protects the system from buggy or malicious code.

The resources managed by an OS include processor time, main memory, secondary storage, and devices. It must share these resources efficiently while remaining stable even when a process behaves incorrectly.

1.1.1 Hardware Abstractions

The OS replaces raw hardware with more useful abstractions:

Hardware resource	OS abstraction or service
Processor	processes and threads, synchronization, and scheduling
Main memory	address spaces and virtual memory
Secondary storage	files and file systems
Peripherals	devices, drivers, and I/O

Without these abstractions, an application would need to understand the low-level protocol of every storage device, network adapter, display, and other peripheral it might encounter.

1.1.2 Library and Multitasking Operating Systems

> **Definition 1.1** **Library OS**
>
> A *library OS* is a library of operating-system services linked with a program rather than a full, independently protected OS.

A library OS can be suitable for a simple device that runs one trusted program, but it cannot make good use of the processor while that program waits for I/O, and it provides little protection from faulty code.

Definition 1.2 **Multitasking OS**

A *multitasking OS* can keep multiple processes in existence at the same time. When one process blocks, the OS can run another.

Multitasking requires the OS to prevent one process from monopolizing the processor or corrupting another process's memory.

1.1.3 Kernel and User Space

Definition 1.3 **Kernel**

The *kernel* is the part of the OS that responds to system calls, interrupts, and exceptions. It mediates communication between applications and hardware and remains active from boot until shutdown.

- A **system call** is how a user program requests a kernel service.
- An **interrupt** or **exception** is how hardware or the processor transfers control to the kernel.

The complete operating system can also include user-space libraries, command interpreters, interfaces, and utility programs.

- **User space** contains ordinary applications. These programs cannot interact with hardware or kernel memory directly.
- **Kernel space** contains privileged kernel code and data.

Crossing from user space into kernel space is substantially more expensive than an ordinary function call, so applications normally use well-defined system call interfaces rather than invoking kernel functions directly.

1.1.4 Kernel Architectures

Definition 1.4 **Monolithic Kernel**

A *monolithic kernel* places most OS services, including file systems and device drivers, in kernel space.

Definition 1.5 **Microkernel**

A *microkernel* keeps only essential mechanisms—typically interprocess communication, scheduling, and basic memory management—in the kernel. Other services run as user-space processes.

A **hybrid kernel** lies between these designs.

	Monolithic kernel	Microkernel
Performance	Fewer crossings between protection domains; generally faster.	More messages and user/kernel transitions; generally slower.
Fault isolation	A faulty kernel driver can crash the entire system.	A faulty user-space service can often fail without crashing the kernel.

Note

The fundamental trade-off is that a monolithic kernel is usually faster, while a microkernel is usually more stable because it isolates more services.

2 Processes and the Kernel

2.1 2a: Processes

Definition 2.1 **Process**

A **process** is an instance of a running program together with its execution environment: a virtual address space, open resources, credentials, and kernel bookkeeping.

The kernel creates and manages processes and gives each one a numeric **process identifier (PID)**. Isolation gives each process the useful illusion that it has exclusive access to the machine's resources, even though the OS multiplexes the processor, memory, and devices among many processes.

Note

A program is passive code stored in a file; a **process** is a live execution of that program. Several processes can execute the same program at once.

2.2 2b: Process System Calls

The process abstraction from Definition 2.1 comes with an API that lets programs create other processes, replace their programs, wait for completion, and terminate.

2.2.1 Creating and Reaping Processes

`fork()` creates a child that initially copies the calling process. Both processes resume immediately after the call, but receive different results:

```
pid_t child = fork();
if (child == 0) {
    /* child */
} else {
```

```
    /* parent: child is the child's PID */
}
```

- The child receives `0`.
- The parent receives the child's PID.
- Failure returns `-1` to the parent and creates no child.
- The parent and child execute independently; **their execution order is not determined**.

`exit(status)` terminates the calling process. The kernel retains enough information for the parent to collect its status. `waitpid(pid, &status, options)` waits for a selected child; `WIFEXITED(status)` checks for ordinary termination and `WEXITSTATUS(status)` extracts the exit code. `kill(pid, sig)` sends a signal, and `getpid()` returns the caller's PID.

2.2.2 Replacing a Program

`execv(path, argv)` replaces the calling process's program and address space. On success it **does not return**, but the process keeps its PID and selected kernel resources such as open file descriptors.

> **Note**
>
> A command shell normally uses **`fork()` followed by `execv()`**: the child becomes the requested program, while the parent shell waits or continues. `fork()` creates a process; `execv()` changes what that process runs.

2.3 2c: Entering the Kernel

> **Definition 2.2** — **System Call**
>
> A **system call** is a controlled request from an unprivileged application to the privileged kernel.

Ordinary user code cannot execute privileged instructions or access the kernel defined in Definition 1.3. The processor therefore has at least two privilege modes: **user mode**, where applications run with restrictions, and **kernel mode**, where the OS can configure hardware and access protected state. This boundary protects the machine from faulty or hostile processes.

Three events transfer control to the kernel through closely related trap machinery:

- A **hardware interrupt** is raised asynchronously by a device, such as a timer or disk controller.
- An **exception** is raised synchronously by the processor because of the current instruction, such as division by zero or an invalid memory access.
- A **system call** is a deliberate software-generated trap requesting an OS service.

2.3.1 Calling Convention and Saved State

The course uses x86-64 rather than the MIPS model from CS 241: x64 has named registers such as `rax`, and its ABI passes several arguments in registers. A calling convention specifies where arguments and results live. The first six integer or pointer arguments are normally passed in registers; additional arguments use the stack. It also divides registers into **caller-saved** and **callee-saved** sets and requires stack alignment (the stack pointer is divisible by 8 in the course model). `call` saves a return address and transfers control; `ret` returns to it.

CastorOS invokes a system call with `int 60`. The vector reaches `T_SYSCALL`, changes privilege level, switches to a protected kernel stack, and saves the interrupted user state in a **trap frame**.

The kernel cannot trust a user stack, which may be corrupt or maliciously constructed, so it maintains its own stack for privileged execution.

Definition 2.3 **Trap Frame**

A **trap frame** is the saved processor state—registers, instruction pointer, flags, stack information, and trap metadata—needed to inspect an entry into the kernel and later resume the interrupted context.

The application supplies a **system-call number** so one common entry point can dispatch to the correct kernel service.

2.4 2d: Complete System-Call Path

The complete CastorOS path for a system call (Definition 2.2) is:

1. The application calls a C library wrapper.
2. The wrapper places the system-call number and arguments according to the ABI and executes `int 60`.
3. Low-level trap code enters on the kernel stack, saves the trap frame (Definition 2.3), and passes it through `trap_common` and `trap_entry`.
4. `Syscall_Entry` validates the request and dispatches to the appropriate handler, which may call other kernel subsystems or a device driver.
5. The handler places its result in the saved `rax`; errors are represented by the system-call interface's error convention, such as `errno` in libc.
6. The reverse path restores the trap frame and `iretq` returns to user mode at the instruction after the trap.

Note

The privilege transition and trap frame are essential. The kernel must be able to distrust user input, run on protected state, and restore **exactly the user context that was interrupted**.

An ordinary function call stays at one privilege level and trusts its caller. A system call crosses a protection boundary, validates untrusted arguments, and may block or switch to another thread, so it is considerably more costly.

3 Threads

3.1 3a: Threads and Concurrency

Definition 3.1 **Thread**

A **thread** is a schedulable sequence of execution within a process. It has its own program counter, registers, stack, and scheduling state.

Threads in one process share what makes communication useful—code, global variables, heap memory, and open files—but keep separate stacks and register state so each can follow a different control flow.

Definition 3.2 **Concurrency**

Concurrency means that multiple activities are in progress during overlapping periods. **Parallelism** means that multiple activities execute physically at the same instant on different processors.

A single core can provide concurrency by interleaving threads. A **preemptive** system may interrupt a running thread; a **cooperative** system switches only when a thread yields or blocks.

Threads can improve:

- **utilization**, by running useful work while another thread waits for I/O;
- **latency and responsiveness**, by separating long work from interactive work;
- **parallel speed**, by using several processors;
- program structure, by assigning different roles or priorities to threads.

They are lighter than processes because they share most resources, but this sharing makes accidental interference and data races possible.

If fraction B of a program is inherently serial, Amdahl's law gives

$$T(n) = T(1)\left(B + \frac{1-B}{n}\right), \quad \text{speedup} = \frac{T(1)}{T(n)}.$$

Note

Even with infinitely many processors, speedup is bounded by $\frac{1}{B}$. Adding threads cannot accelerate the program's serial part.

3.2 3b: Scheduling and Context Switching

A runnable thread, in the sense of Definition 3.1, eventually stops running because it **blocks**, voluntarily **yields**, exits, or is **preempted**. A periodic timer interrupt lets the kernel enforce a time quantum, preventing one CPU-bound thread from monopolizing a processor.

3.2.1 Thread States

- **New → ready**: the thread becomes eligible to run.
- **Ready → running**: the scheduler dispatches it.
- **Running → ready**: it yields or its quantum expires.
- **Running → blocked**: it waits for an event or resource.
- **Blocked → ready**: the event occurs.
- **Running → terminated**: execution finishes.

> **Warning**
>
> **Ready** and **blocked** are not synonyms. A ready thread needs only a CPU; a blocked thread cannot make progress until some external condition changes.

> **Definition 3.3** **Context Switch**
>
> A **context switch** saves the execution state of one thread and restores another so the latter can resume.

Context switches cost more than the register save itself: the incoming thread may suffer cache misses, an address-space switch may disrupt the TLB, and floating-point or vector state may also need saving. Longer quanta reduce this overhead but can hurt response time; shorter quanta improve responsiveness but switch more often.

A simple round-robin scheduler takes a thread from the front of a ready queue and returns a still-runnable thread to the back. The exact interleaving remains **nondeterministic**.

3.2.2 CastorOS Frames

On entry from user mode, the **trap frame** from Definition 2.3 records user-visible state. If that kernel execution later performs a context switch (Definition 3.3), a **switch frame** on the kernel stack stores the callee-saved kernel context required to resume the suspended kernel call.

When switching between processes, CastorOS may also load the next address space and saves/restores floating-point state before `switchstack` exchanges the kernel-thread switch frames.

> **Note**
>
> A trap frame answers "how do I return from the kernel to the interrupted context?" A switch frame answers "how do I resume this suspended kernel thread?" They serve different boundaries and may both exist on one kernel stack.

3.3 3c: Thread Implementations

The POSIX interface uses `pthread_create` to start a thread, `pthread_exit` to finish it, and `pthread_join` to wait for and collect another thread's result.

User threads are created and scheduled by an application runtime, making their operations inexpensive. **Kernel threads** are created and scheduled by the kernel; having several lets a process execute on several cores.

Thread packages differ in how **user threads** map to kernel-scheduled threads:

Model	Advantages	Costs and limitations
`n:1`	Scheduling and synchronization can be entirely in user space and therefore very cheap.	No multicore parallelism; one blocking system call can block every user thread.
`1:1`	A blocked thread does not block its siblings; threads can run in parallel on several cores.	Thread operations enter the kernel; fixed kernel stacks and bookkeeping consume memory; one policy must serve many runtimes.
`n:m`	Combines user-level flexibility with parallel kernel execution.	Coordination is complex, especially around blocking, and the kernel cannot see the relative importance of individual user threads.

> **Note**
>
> There is no universally best mapping. The choice trades **operation cost and runtime control** against **blocking behavior, parallelism, and complexity**.

4 Synchronization

4.1 4a: Races and Critical Sections

Concurrent instructions may interleave in many valid orders. Even an expression such as `count++` is a read, computation, and write; two threads can read the same old value and one update is then **lost**.

Definition 4.1 **Race Condition**

A **race condition** exists when a program's result depends on the relative timing or interleaving of concurrent operations.

Definition 4.2 **Critical Section**

A **critical section** is code that accesses shared state and must not overlap with conflicting operations. **Synchronization** constrains concurrency so these sections execute safely.

A correct critical-section protocol should provide:

- **mutual exclusion**: at most one participant is in the protected section;
- **progress**: if the section is free, eligible contenders can decide who enters;
- **bounded waiting**: a contender is not postponed forever (**starvation**);
- independence from accidental assumptions about processor count or speed.

Note

Mutual exclusion protects an **invariant**, not merely a line of code. Every access that can violate the same invariant must follow the same protocol.

4.2 4b: Hardware Memory Ordering

Source-code order is not automatically the order in which other processors observe memory operations. Processors overlap work, buffer stores, speculate, and maintain private caches.

Important operation pairs are:

- **write → read**: a later load may execute before a buffered store becomes globally visible;
- **write → write**: stores can be combined or become visible differently from source order;
- **read → read**: independent loads can proceed non-blockingly or speculatively;
- **read → write**: execution and cache effects may expose an order weaker than the program appears to request.

Cache coherence keeps copies of one cache line consistent, but propagation is not instantaneous and coherence alone does not impose the desired order across different locations.

Definition 4.3 **Sequential Consistency**

An execution is **sequentially consistent** if its result is as though all processors' operations occurred in one total order that preserves each processor's program order, and each write appears atomic in that order.

Note

Sequential consistency does **not** require a predetermined interleaving. It requires that every observed result be explainable by some single interleaving that respects each thread's order.

4.3 4c: Compilers, Fences, and Atomic Instructions

The sequential-consistency model from Definition 4.3 prevents many useful optimizations. Hardware and compilers may perform code motion, keep values in registers, eliminate common subexpressions, and reorder independent accesses unless the language and machine memory models prohibit it.

On x86, fences constrain visibility:

- `lfence` orders loads;
- `sfence` orders stores;
- `mfence` orders both loads and stores.

Atomic instructions such as `xchg` and locked `cmpxchg` perform indivisible read-modify-write operations and provide ordering appropriate for building synchronization. The `lock` prefix extends atomicity and ordering across cores.

Warning

`volatile` is **not** a general synchronization primitive. Correct concurrent code must use the language's atomics or established locks, which incorporate both atomicity and the required memory ordering.

x86 has a relatively strong model but still permits the important store→load relaxation. ARM permits more reorderings. Portable code should rely on synchronization abstractions, not assumptions inherited from one processor.

4.4 4d: Mutexes, Semaphores, and Condition Variables

Consider a bounded circular producer-consumer buffer. Producers add entries, consumers remove them, and shared indices or a `count` record its state. Merely checking whether the buffer is full or empty is insufficient: simultaneous updates can race.

4.4.1 Mutexes

Definition 4.4 **Mutex**

A **mutex** is an ownership-based lock that allows one thread at a time to hold it. The holder must unlock it.

A POSIX mutex is initialized and later destroyed; `pthread_mutex_lock` acquires it, `pthread_mutex_unlock` releases it, and `pthread_mutex_trylock` attempts an acquisition without blocking.

A mutex makes short state transitions atomic, but a thread should not busy-wait while holding it for a condition another thread must establish.

4.4.2 Semaphores

Definition 4.5 **Semaphore**

A **semaphore** stores a nonnegative count. Atomic `wait` decrements the count when positive or blocks; atomic `post` increments it and can wake a waiter.

A binary semaphore can provide exclusion, while a counting semaphore represents available resource units. Semaphores have no inherent owner.

For a buffer of capacity N, initialize `not_full` to N and `not_empty` to 0. A producer waits on `not_full`, inserts, then posts `not_empty`; a consumer does the reverse. With multiple producers or consumers, a separate mutex is still needed to protect index and buffer updates.

4.4.3 Condition Variables

Definition 4.6 **Condition Variable**

A **condition variable (CV)** lets threads sleep until shared state protected by a mutex **may** satisfy an arbitrary predicate.

The standard pattern is:

```
mutex_lock(&m);
while (!condition())
    cv_wait(&cv, &m);  /* atomically unlocks, sleeps, then relocks */
use_or_change_state();
mutex_unlock(&m);
```

`signal` wakes one waiter and `broadcast` wakes all. A CV does not accumulate signals: signaling when no thread waits stores no future credit. Each CV normally corresponds to one predicate of interest, while the associated mutex protects the state used to evaluate it.

Warning

Always recheck the predicate in a **`while` loop**. By the time a woken thread reacquires the mutex, another thread may have changed the state. A wakeup is only permission to check again, not a guarantee that the condition is true.

The unlock-and-sleep action must be atomic with respect to signaling. Otherwise a signal can occur between the predicate check and the sleep, producing a **lost wakeup**.

4.4.4 Choosing a Primitive

Primitive	Represents	Best use
Mutex	Exclusive ownership	Protecting a shared invariant

Primitive	Represents	Best use
Semaphore	Stored resource count or event credits	Managing identical resource units or a pipeline
Condition variable	Waiters for a predicate; no stored signal	Waiting for an arbitrary state change while using a mutex

Definition 4.7 **Data Race**

A **data race** occurs when concurrent threads access the same memory location, at least one access is a write, and the accesses lack the synchronization required by the language memory model.

Protect shared globals consistently. In particular, a check followed by an act must be one synchronized operation when the checked fact can change. Audit every point where preemption could expose a partially updated invariant. With a correct mutex implementation and consistent locking, protected accesses appear sequentially consistent to the participating threads.

4.5 4e: Implementing Synchronization

4.5.1 Test-and-Set and Spinlocks

An atomic exchange can implement test-and-set:

```
while (atomic_exchange(&lock, 1) == 1)
    ;                           /* spin */
/* critical section */
atomic_store_release(&lock, 0);
```

Definition 4.8 **Spinlock**

A **spinlock** repeatedly tests until it acquires a lock. It consumes processor time while waiting, so it is appropriate only when waits are very short or sleeping is impossible.

Acquire ordering prevents protected reads and writes from moving before lock acquisition; release ordering (conceptually including a store fence) ensures they become visible before unlock.

Kernel spinlocks commonly disable interrupts on the local processor so an interrupt handler cannot deadlock trying to acquire a lock held by the interrupted code. A user-space spinlock must **not** disable interrupts.

4.5.2 Blocking Locks and Wait Channels

Definition 4.9 **Wait Channel**

A **wait channel** is a queue of threads blocked for a particular event. Its operations sleep, wake one waiter, or wake all waiters.

Sleeping releases the processor. The state check, queue insertion, and lock release must be coordinated so a wakeup cannot be missed.

A semaphore (Definition 4.5) can therefore be implemented using a small spinlock (Definition 4.8) plus a wait channel (Definition 4.9):

```
wait(s):
    spin_lock(s.lock)
    while (s.count == 0)
        sleep(s.channel, s.lock)  /* returns with lock held */
    s.count--
    spin_unlock(s.lock)

post(s):
    spin_lock(s.lock)
    s.count++
    wake_one(s.channel)
    spin_unlock(s.lock)
```

4.5.3 Lock Granularity

A **coarse-grained** lock is simple and protects a large structure but increases contention. **Fine-grained** locks permit more concurrency but require more metadata and careful reasoning about relationships between pieces.

Hand-over-hand traversal acquires the next node's lock before releasing the current node's lock. Using a fixed traversal order and holding at most two adjacent locks preserves the structure while limiting contention.

CastorOS semaphores combine these ideas: wait channels provide blocking, fine-grained locks protect individual objects, and hand-over-hand locking moves safely between the semaphore and its wait channel.

4.6 4f: Advanced Races and Deadlock

Some unsynchronized reads are sometimes called **benign races**, a special case of the race condition in Definition 4.1. For example, a fast-path read used by double-checked initialization may be intentionally unsynchronized. In language-level code, however, an ordinary data race (Definition 4.7) can be undefined behavior. Double-checked locking is sound only with proper atomic publication and memory ordering.

A useful **lockset** discipline associates each shared variable with one lock and requires every conflicting access to hold that same lock.

Definition 4.10 **Deadlock**

A **deadlock** is a permanent cycle of waiting in which every participant needs a resource held by another participant in the cycle.

All four Coffman conditions are necessary:

1. **mutual exclusion**—a resource cannot be shared;
2. **no preemption**—resources cannot simply be taken away;
3. **hold and wait**—a participant holds resources while requesting more;
4. **circular wait**—participants form a directed waiting cycle.

Deadlock prevention breaks at least one condition. The most common discipline assigns a global partial order to locks and requires every thread to acquire them in that order, eliminating circular wait.

Do not hold a mutex while crossing an abstraction boundary unless the callee's locking behavior is part of the contract; hidden lock acquisition is a common way to create an unexpected cycle.

For deadlock as defined in Definition 4.10, a resource-allocation graph gives these tests:

- no cycle means no deadlock;
- with one instance of each resource, a cycle is both necessary and sufficient for deadlock;
- with multiple instances, a cycle is necessary but **not sufficient**.

4.6.1 Cache Modes and Consistency

x86 supports memory types including write-back (**WB**), write-through (**WT**), uncacheable (**UC**), and write-combining (**WC**). WB is appropriate for ordinary RAM; UC and WC help interact with devices. Non-temporal operations can bypass parts of the cache hierarchy.

Note

Performance-oriented cache modes and weak ordering are safe only when code explicitly establishes the consistency it needs. Synchronization is the bridge between aggressive hardware execution and simple program invariants.

5 Devices and I/O

5.1 5a: Device Interaction

Devices provide **input**, **output**, or both. They connect to the processor and memory through buses: fast internal interconnects serve memory and high-speed devices, while peripheral buses trade speed for flexibility and cost.

The kernel normally interacts with a device through a few registers:

- read **status** to learn whether the device is ready or has completed work;
- write a **command** to request an operation;

- read or write **data** registers to transfer information.

With **port-mapped I/O (PMIO)**, special instructions access a separate port address space. With **memory-mapped I/O (MMIO)**, device registers occupy physical addresses and ordinary-looking loads and stores reach them. A register is usually identified by `device base + register offset`, and the device is assigned an interrupt request line (**IRQ**).

Definition 5.1 **Device Driver**

A **device driver** is kernel code that translates a generic OS request into a particular device's command protocol and handles its completion events.

5.1.1 Polling and Interrupts

Polling repeatedly reads device status. It is simple and can be efficient for a very brief wait, but wastes CPU time during a long or unpredictable operation. With **interrupt-driven I/O**, the driver starts the request, lets the thread sleep, and the device interrupts when attention is required.

A synchronous disk operation can therefore create a request containing a semaphore, issue it, block on the semaphore, and have the completion interrupt handler post the semaphore.

5.1.2 PIO and DMA

- In **programmed I/O (PIO)**, CPU instructions move every data item between the device and memory.
- In **direct memory access (DMA)**, the CPU describes a memory region, and a DMA engine or device controller transfers the bulk data directly.

Note

Polling versus interrupts asks **when and how completion is noticed**. PIO versus DMA asks **who moves the data**. These are independent choices.

A typical write through a device driver (Definition 5.1) is: validate and queue the request, program device registers or a DMA descriptor, start the command, block the caller if necessary, handle the IRQ, acknowledge completion, and wake the waiting thread.

5.2 5b: Hard-Disk Geometry and Performance

A hard disk contains rotating **platters** with recording surfaces and heads. Concentric **tracks** are divided into sectors; tracks at the same radius form a **cylinder**. Heads move together, so switching surfaces within one cylinder is usually cheaper than seeking to another cylinder.

Rotation speed drifts slightly, so the controller must continually track position rather than infer an exact angle solely from elapsed time.

Older interfaces exposed cylinder-head-sector (**CHS**) coordinates with a fixed sector count. Modern disks expose sequential **logical block addresses (LBAs)**. Logical blocks

can group several physical sectors into larger transfer units. The controller privately maps LBAs to physical locations, allowing outer tracks to hold more sectors through zone recording while keeping a simple block interface. Nearby LBAs usually—but not perfectly—preserve physical locality.

Positioning includes acceleration, travel, settling, head selection, and write-position precision. The familiar approximation

$$\text{average seek time} \approx \frac{\text{maximum seek time}}{3}$$

reflects the average distance between two uniformly selected tracks rather than half the maximum distance.

Switching heads within a cylinder avoids arm travel, but its time can still be comparable to a very short seek.

> **Note**
>
> The kernel does not know the controller's exact physical mapping. Zoning, track skew, spare sectors, and remapping mean that LBA order is only a useful approximation to geometry.

Controllers can maintain their own command queue, read ahead, and acknowledge writes from a volatile or persistent cache. The kernel and controller may both schedule requests, sometimes with incomplete knowledge of each other's choices.

5.2.1 Service-Time Calculations

For rotation rate R RPM,

$\text{rotation time} = 60\frac{\text{s}}{R}$, $\text{average rotational latency} = \frac{\text{rotation time}}{2}$.

If a track contains S sectors and the request transfers b adjacent sectors,

$\text{transfer time} = \frac{b}{S} \cdot \text{rotation time}.$

An expected random request is approximately

$\text{service time} = \text{average seek} + \text{average rotation} + \text{transfer time}.$

> **Warning**
>
> Keep units consistent. RPM must become seconds per rotation before it can be added to seek time, and transfer fraction is sectors requested divided by sectors per track.

5.3 5c: Disk Scheduling

Sequential requests are much faster than random requests because positioning dominates small transfers. When several requests are outstanding, scheduling can reduce head motion.

Policy	Rule and advantage	Limitation
FCFS	Serve arrival order; simple and fair.	Poor locality and average service time.
SSTF	Serve the request nearest the current head; good locality.	A far request can starve.
SCAN	Sweep in one direction, then reverse; bounded waiting.	Middle cylinders tend to receive better service.
C-SCAN	Serve in one direction and jump back; more uniform waiting.	The return jump does no useful service.

An aged shortest-seek policy can use an effective score such as

$$T_e = T_{\text{position}} - wT_{\text{wait}},$$

so a request becomes increasingly attractive as it waits. The weight w trades locality for starvation resistance.

5.4 5d: Flash Storage

Flash is organized into **pages** grouped into larger **erase blocks**. A page can be read independently, and programming changes bits from 1 to 0. Restoring bits from 0 to 1 requires erasing an entire block, so an existing page cannot simply be overwritten in place.

The **flash translation layer (FTL)** maps logical blocks to physical pages. An update writes a fresh page, redirects the mapping, and marks the old page invalid. **Garbage collection** copies remaining valid pages and erases blocks to recover free space.

Erase blocks survive only a limited number of cycles, so **wear leveling** distributes erasures across the medium.

Note

Flash exposes a block interface like a disk, but internally it trades extra mapping, garbage collection, and write amplification for low read latency. More storage levels increase density and lower cost, but generally reduce endurance and increase access complexity and latency.

6 File Systems

6.1 6a: Files, Directories, and Names

Definition 6.1 File

A **file** is a persistent, named sequence of bytes plus metadata such as its owner, permissions, size, and timestamps.

Storage is much larger and slower than memory and must survive failures. A file system turns device blocks into durable named objects while controlling sharing and preserving metadata consistency.

Its implementation is commonly layered:

- the **logical or virtual file-system layer** handles names, descriptors, permissions, and a uniform interface;
- the **physical file-system layer** maps file operations to a particular on-disk format and block device.

6.1.1 The Descriptor Interface

`open` resolves a name and returns a small **file descriptor**. `read` and `write` transfer bytes at the open instance's current position; `lseek` changes that position; `close` releases the descriptor. Separate opens can have separate positions, while duplicated or inherited descriptors can refer to the same open file state.

6.1.2 Directories and Paths

A directory maps a filename to a unique inode number (**inumber**). Directories are themselves files with a special interpretation, producing a hierarchy. An absolute path begins at the root; a relative path begins at the current working directory. Components such as `.` and `..` have special meanings, and shell search paths locate executable names.

Directory designs progressed from one system-wide directory, to one directory per user, to the hierarchical namespace used by Unix-like systems.

Definition 6.2 **Hard Link**

A **hard link** is another directory entry mapping a name to an existing inode. The inode is deleted only when its link count reaches zero and no live open reference remains.

Hard links preserve referential integrity because every name directly selects the object, but file systems normally forbid hard links to directories to avoid cycles.

Definition 6.3 **Symbolic Link**

A **symbolic link** is a file whose contents are another pathname. Resolving the link follows that path, which may cross file systems or become dangling.

Mounting attaches the root of one file system at a directory in another, combining several independent storage structures into one namespace.

6.2 6b: File Layout and Implementation

To access byte offset x of a file (Definition 6.1) with block size B, the logical block is $\lfloor \frac{x}{B} \rfloor$ and the within-block offset is $x \bmod B$. Large blocks improve throughput and reduce metadata; small blocks reduce wasted space for small files.

- **Internal fragmentation** is unused space inside an allocated fixed-size unit, such as the tail of a file's last block.
- **External fragmentation** is free space split into unusable gaps between variable-size extents.

6.2.1 Allocation Strategies

Layout	Strength	Weakness
Contiguous	Fast sequential and random access; little metadata.	Growth and external fragmentation.
Linked	Easy growth and no external fragmentation.	Poor random access; pointer overhead and corruption risk.
FAT	Keeps the linked list in a table that can be cached.	Large central table; chains still govern lookup.
Indexed	An inode indexes data blocks for efficient random access.	Index space and extra lookups for large files.

Unix-like indexed files use several **direct pointers** followed by single, double, and possibly triple **indirect pointers**. Direct pointers make small files cheap; each added indirection expands maximum file size without making every inode enormous.

If a pointer has p bytes and a block has B bytes, one indirect block holds $\frac{B}{p}$ pointers. A double-indirect pointer therefore addresses $\left(\frac{B}{p}\right)^2$ data blocks. Multiply the total reachable data blocks by B for maximum byte size.

6.2.2 A Very Simple File System

A compact VSFS-style disk layout contains:

1. a **superblock** describing format and dimensions;
2. inode and data-block **bitmaps** recording free units;
3. an **inode table** containing file metadata and block pointers;
4. the **data region** containing file and directory blocks.

Most blocks should hold file data rather than metadata. An inode's inumber is its index in the inode table. Any on-disk design must also specify how the root is found at boot and how metadata is protected against corruption.

Opening `/foo/bar` starts at the root inode, reads its directory data to find `foo`, reads `foo`'s inode and directory data to find `bar`, then loads `bar`'s inode. Reading the file translates the current byte offset through its block pointers and copies bytes from the selected data block; the file system may then write the inode to update its access time.

Creating a file may update a directory block, directory inode, free-inode bitmap, and new inode; adding data also updates the data bitmap and blocks. A partial-block write must first preserve the untouched bytes, whereas a full block overwrite need not read the old contents.

6.2.3 In-Memory State

Each process has a descriptor table. Its entries reference system-wide **open file descriptions** that store the current offset and mode. These in turn refer to cached inode objects. A shared block or page cache avoids repeated device reads.

> **Note**
>
> A descriptor is not an inode. The chain is approximately **process descriptor → open file description → inode → data blocks**. This explains which state is private, which is shared, and where the current offset lives.

6.3 6c: Crash Consistency and Journaling

A crash can interrupt the several writes needed for one logical update. `fsck` recovers by scanning metadata, reconstructing free-space information, checking link counts and block ownership, and finding reachable or orphaned objects. It can be slow because it examines much of the file system, and the newest update may still be lost even when metadata is recoverable.

Purely synchronous ordered writes preserve consistency but perform poorly. At minimum, allocation should maintain these ordering invariants:

1. never publish a pointer before initializing the structure it points to;
2. never reuse a resource before nullifying every old pointer to it;
3. when moving the last pointer to a live resource, install the new pointer before clearing the old one.

> **Definition 6.4** **Journaling**
>
> **Journaling** uses write-ahead logging: it first records the metadata updates of a transaction in a journal, commits that transaction atomically, and only then installs the updates in their home locations.

With journaling (Definition 6.4), committed transactions can be replayed after a crash and incomplete ones ignored. Once home-location writes are safely checkpointed, journal space can be reused in a circular log. Checksums help distinguish a complete committed record from a torn or partially written one.

File systems typically journal **metadata** rather than all file data, and group several related operations into one transaction. Checkpoints identify the oldest committed transaction that may still require replay.

> **Note**
>
> The journal does not make a multi-block device write physically atomic. It provides a durable record from which the file system can reproduce either the complete committed metadata update or the old state after recovery.

7 Virtual Memory

7.1 7a: Address-Space Virtualization

Virtual memory should support **multitasking**, **efficient translation**, **process isolation**, **efficient physical-memory use** through low fragmentation and controlled sharing, **dynamic process growth**, and an **opaque interface** that hides physical placement from applications.

Physical memory is one shared machine resource. The OS instead gives each process a private-looking **virtual address space** containing code, data, heap, shared regions, and stacks. The MMU translates every virtual address to a physical address under kernel control.

> **Note**
>
> Virtualization creates an illusion through both **time multiplexing** and **space multiplexing**: processes use resources at different times or occupy different portions simultaneously, while each sees a simpler private environment.

Binary units are 2^{10} bytes per KiB, 2^{20} per MiB, 2^{30} per GiB, and 2^{40} per TiB. An n-bit address names 2^n bytes in a byte-addressed space. An object of size S bytes fits $\left\lfloor \frac{2^n}{S} \right\rfloor$ times in that space, ignoring alignment and reserved regions.

7.2 7b: Address Translation

Translation designs trade flexibility, fragmentation, lookup cost, and metadata size.

The six increasingly flexible course designs are **load-time linking**, **base-and-bound relocation**, **segmentation**, **paging**, **two-level paging**, and general **multilevel paging**.

7.2.1 Loading and Base/Bound Relocation

Simple load-time linking adds a chosen physical base to addresses before the program runs. It cannot cheaply move a live process or enforce protection on every access.

With dynamic **base and bound** relocation, a virtual address v is legal when $0 \leq v < \text{bound}$, and hardware produces

$$\text{physical address} = \text{base} + v.$$

The OS changes base/bound on a context switch, while the MMU checks every access. This is fast but gives each process one contiguous region, causing external fragmentation and awkward growth.

7.2.2 Segmentation

Segmentation divides an address into a segment identifier and an offset. Each segment-table entry supplies a physical base, bound, and permissions. Separate code, data, heap, and stack segments permit protection and sharing but still use variable-size physical regions and therefore suffer external fragmentation.

7.2.3 Paging

Paging divides virtual memory into fixed-size **pages** and physical memory into equal-size **frames**. A process's page table maps virtual page numbers (**VPNs**) to physical frame numbers (**PFNs**) and stores status and protection bits.

For page size 2^k bytes,

$$\text{VPN} = \left\lfloor \tfrac{v}{2^k} \right\rfloor, \quad \text{offset} = v \bmod 2^k,$$

$$\text{physical address} = \text{PFN} \cdot 2^k + \text{offset}.$$

For example, with 4 KiB pages, virtual `0x102C` has VPN `0x1` and offset `0x02C`; if that PTE selects frame `0x26`, the physical address is `0x2602C`.

Each process needs its own page table. A valid bit says whether an entry may be used, and read/write/execute and user/kernel bits enforce access policy. PTEs can also request write-through rather than default write-back caching, or disable caching for regions such as memory-mapped device registers.

7.2.4 Multilevel Page Tables

A flat table has 2^{n-k} entries for an n-bit virtual address and 2^k-byte pages, even when most of the address space is unused. Multilevel tables split the VPN into indices and allocate lower-level table pages only for populated regions.

If one page-table page holds E entries, each level contributes $\log_2 E$ index bits. With 64-bit entries and 4 KiB pages, $E = 512$ and each level contributes 9 bits. Conventional x86-64 translation uses four 9-bit indices plus a 12-bit offset. Without caching, one data access can require four page-table memory reads plus the requested access.

> **Note**
>
> Multilevel tables save metadata for **sparse** address spaces, but a page walk adds memory references. Caches in the MMU are therefore essential.

7.3 7c: The TLB and Kernel Mappings

The kernel chooses mappings, allocates tables, and handles exceptional cases; the MMU performs ordinary permission checks and translations on every memory reference.

> **Definition 7.1** **Translation Lookaside Buffer**
>
> The **TLB** is a small associative cache of recent page-table entries. A hit avoids a page-table walk.

Because the TLB from Definition 7.1 operates at processor-pipeline speed, it must be small and fast.

On a miss, a hardware-managed TLB walks page tables itself. A software-managed TLB traps to the kernel, which finds the PTE, inserts it, and retries the faulting instruction.

TLB entries must agree with page tables. The kernel can invalidate an individual mapping with an operation such as `invlpg`. Tagged entries use a process-context identifier (**PCID/ASID**) to keep translations from several address spaces without confusing them. **Global** kernel entries can survive ordinary process switches.

The kernel is commonly mapped into every process's page table so a trap can enter kernel code without first switching the entire address space. Permission bits make those mappings inaccessible in user mode. x86-64 conventionally places user mappings in `0x0000 0000 0000 0000–0x0000 7FFF FFFF FFFF` and kernel mappings in `0xFFFF 8000 0000 0000–0xFFFF FFFF FFFF FFFF`; addresses between those canonical ranges are invalid.

> **Warning**
>
> Changing a PTE is not enough: stale TLB entries on the current and other processors must be invalidated or otherwise synchronized before the old translation can no longer be used.

7.4 7d: Demand Paging

Virtual pages need not all be resident in RAM. Their contents may be in an executable file, a memory-mapped file, swap space, or implicitly all zeros. Demand paging loads a page only when it is first accessed. Smaller resident sets also allow more processes to remain active concurrently.

This supports:

- lazily loaded code and data;
- zero-filled BSS and fresh heap/stack pages;
- **copy-on-write**, where shared read-only pages are copied only on a write;
- shared file-backed mappings;
- address spaces larger than current physical memory.

A process's **resident set** is the set of its virtual pages currently occupying frames. A valid virtual page can be nonresident; therefore a page fault is not necessarily a program error.

7.4.1 Fault Handling

For a valid nonresident page, the kernel generally:

1. traps on the fault and identifies the virtual address and access type;
2. validates that the access belongs to a legal region and has permission;
3. obtains a free frame, possibly selecting a victim;
4. writes back a dirty victim if needed and fetches or constructs the requested page;
5. installs the PTE and updates or invalidates the TLB;
6. restarts the faulting instruction.

The processor must report enough information to retry safely, and instructions must have well-defined restart behavior. Invalid addresses or forbidden access instead cause an error such as a segmentation fault.

In the simplest case, an instruction that performs only one memory access can be restarted by re-executing it after the mapping is installed.

Spatial prefetching, background prezeroing, and superpages can reduce overhead, but fetching useless data wastes bandwidth and frames. The core performance goal is to **minimize costly page faults** while using memory efficiently. Two-level TLBs can keep a small fast first level while a larger second level holds more translations, including entries for different page sizes.

7.5 7e: Page Replacement

If processes collectively need more active pages than RAM can hold, they fault continuously and spend most of their time paging. This collapse is **thrashing**.

Global replacement chooses a victim from all processes, adapting readily but allowing one process to disrupt another. **Local replacement** restricts victims to the faulting process, improving isolation but potentially stranding memory.

A process's **working set** is the pages it has used during a recent window. Its resident set should track this locality: often a small portion of an address space receives most accesses, and that portion changes between program phases. Page-fault frequency is

$$\text{PFF} = \frac{\text{number of page faults}}{\text{number of instructions executed}}.$$

PFF can adjust allocation: a rate above an upper threshold suggests adding frames; a rate below a lower threshold suggests reclaiming some.

Policy	Idea	Property
FIFO	Evict the oldest loaded page.	Simple, but ignores use and can show Belady's anomaly.
Optimal	Evict the page whose next use is farthest away.	Best possible, but requires future knowledge; useful as a benchmark.
LRU	Evict the least recently used page.	Exploits locality but exact implementation is expensive.
Clock	Give referenced pages a second chance while scanning circularly.	Efficient approximation to LRU.

Clock clears a page's reference bit on its first encounter and evicts a page whose bit is already clear. Dirty pages are costlier because they require a write-back; enhanced or two-handed clock schemes can age references and prefer clean victims while background work writes dirty pages.

> **Note**
>
> Replacement is prediction: recent use is evidence of future use, not a guarantee. Good policies exploit locality while keeping their own bookkeeping overhead small.

8 Thread Scheduling

8.1 8a: Scheduling Goals and Basic Policies

For job i, let a_i be arrival time, r_i the time it first runs, s_i its required service time, and f_i its finish time. Then

$$\text{response time} = r_i - a_i, \quad \text{turnaround time} = f_i - a_i.$$

The scheduler treats runnable threads as jobs and balances **responsiveness**, **fairness**, and **efficiency**. Improving one workload metric can worsen another.

Policy	Rule and benefit	Limitation
FCFS	Run jobs in arrival order; simple and starvation-free.	A long job can delay every short job—the convoy effect.
Round robin	Run each ready job for a quantum; responsive and fair.	Very small quanta add switching cost; large quanta approach FCFS.
SJF	Run the shortest job next; minimizes average turnaround when lengths are known.	Nonpreemptive and long jobs may starve.
SRTF	Always run the job with least remaining time; preemptive SJF.	Can repeatedly postpone long jobs.

Warning

SJF and SRTF require knowledge the OS normally lacks: a thread's future CPU burst length. Practical schedulers must estimate behavior from the past.

For the four-job course example, the measured averages are:

Policy	Turnaround	Response
FCFS	11.75	7.25
Round robin	13.00	2.75
SJF	8.50	4.00
SRTF	8.25	3.25

This example illustrates the tradeoff: short-job policies reduce turnaround, while preemption generally improves response time.

8.2 8b: Multilevel Feedback Queues

A **multilevel feedback queue (MLFQ)** approximates SJF/SRTF by learning whether a thread behaves interactively.

- Ready queues are ordered by priority; the scheduler chooses the highest nonempty queue.
- Higher-priority queues use shorter quanta for good response time.
- New and newly awakened threads enter at high priority.
- A thread that consumes its whole quantum is preempted and placed at the back of the next lower queue.
- A thread that blocks quickly remains high, favoring interactive or I/O-bound behavior.
- A newly ready high-priority thread preempts lower-priority execution.

This feedback distinguishes short interactive bursts from sustained CPU work without knowing future service times. A malicious or lucky thread can otherwise game the rules by yielding just before demotion, so implementations account for total CPU use rather than only individual quanta.

> **Note**
>
> Periodic priority boosts move waiting threads upward. They prevent starvation and let the scheduler forget stale behavioral history when a workload changes.

8.3 8c: Completely Fair Scheduling

The Completely Fair Scheduler (**CFS**) models an ideal processor that divides CPU capacity among runnable threads according to positive weights w_i. Instead of fixed priority queues, it selects the runnable thread with the smallest **virtual runtime**. User threads begin with a default weight that the `nice` interface can adjust.

To avoid overloading the arrival-time notation from 8a, let t_i denote the actual processor time accumulated by thread i. Virtual time advances faster for low-weight threads and slower for high-weight threads. Over execution interval Δt_i, a representative update is

$$\Delta v_i = \Delta t_i \cdot \frac{\sum_j w_j}{w_i}.$$

Thus fairness aims to keep normalized service comparable: $\frac{t_i}{w_i}$ should be similar across continuously runnable threads. A balanced tree supports efficient selection of the minimum virtual runtime.

A new thread's virtual runtime is initialized near the current runnable range, not at zero, so it receives prompt service without unfairly claiming all CPU time accumulated before it existed. Threads can use similar real quanta while weights change how quickly their virtual runtimes advance.

> **Note**
>
> CFS does not mean every thread receives the same CPU time. It seeks **weighted fairness**: twice the weight should yield roughly twice the processor share when both threads remain runnable.

8.4 8d: Multiprocessor Scheduling

A single shared ready queue makes global choices and naturally balances load, but access to it is a critical section requiring mutual exclusion. All cores therefore contend for the same scheduler state, and a thread may frequently move between processors.

Per-core queues improve scalability and preserve **processor affinity**, keeping a thread near warm cache and TLB state. They require periodic load balancing, and perfect numerical balance may be worse than leaving a small imbalance that preserves useful locality.

> **Note**
>
> Multiprocessor scheduling balances three competing goals: **even CPU load**, **low synchronization overhead**, and **cache affinity**. Moving a runnable thread can improve the first while hurting the other two.