

What happens when a program runs?

CSE513 · Advanced Operating Systems
Lecture 1 · Processes and threads
Prof. Mehmet Göktürk

The questions behind this course

How can many programs share one machine?

What must be saved when execution changes?

Which resources are shared, and what keeps sharing safe?

How the semester works

Lectures: Weeks 1–6 and 8–10, in two halves.

Week 7: midterm; no class.

Weeks 11–14: paper presentations, 15 minutes in English.

Recent papers connect the foundations to current systems.

Assessment and practical work

Midterm 30% · Final 40% · Presentation 15% · Assignments 15%

Half of the final-exam questions come from presentations.

Assignments will be approximately fortnightly and Ubuntu-oriented.

AI assistance is permitted. Detailed briefs will follow.

Today's route

A short refresher: services, protection and kernel entry.

First half: programs, processes, states and context.

Second half: process creation, threads and communication.

Follow one command from launch to completion.

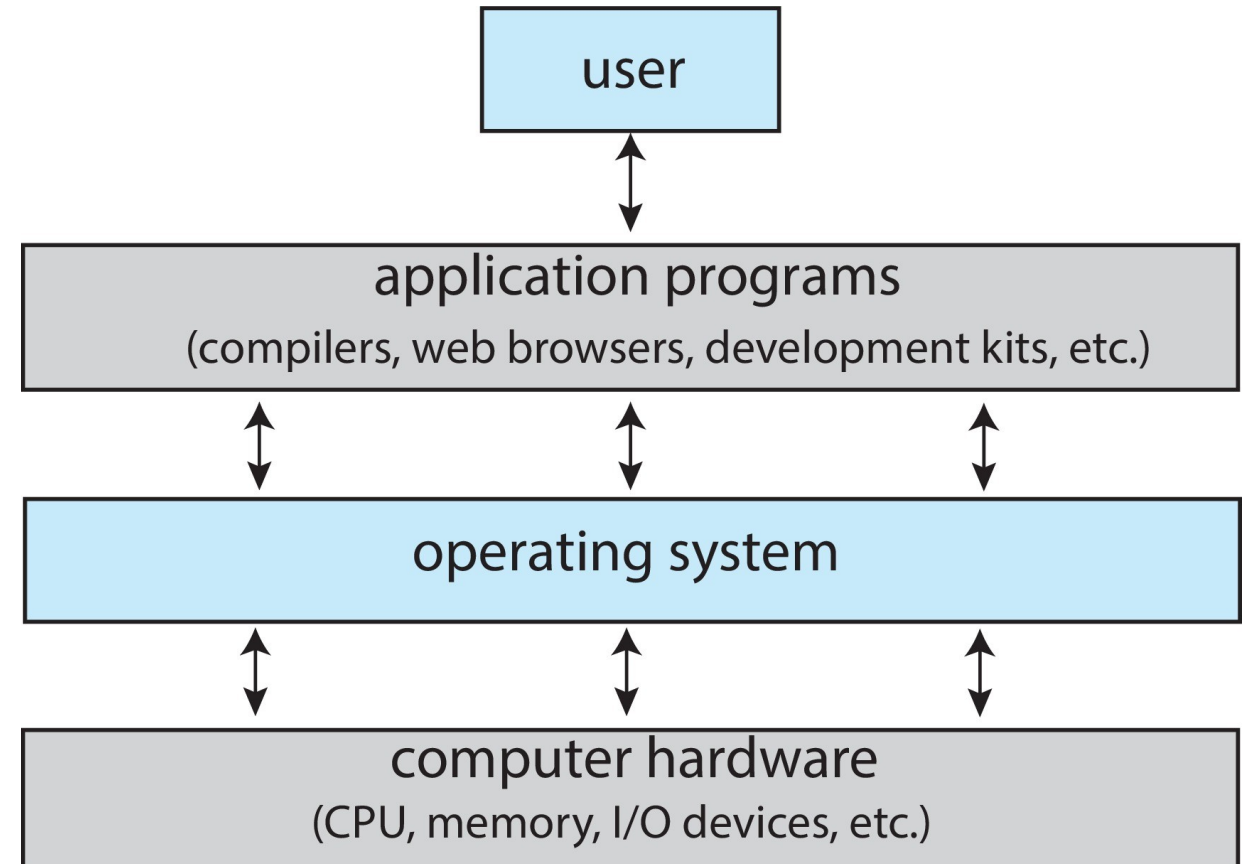
The OS gives programs useful abstractions

Processes give programs execution contexts; virtual memory gives them address spaces.

Files and sockets provide interfaces to data and communication without direct device control.

The kernel allocates resources and enforces protection while applications compete for the same hardware.

A shell is a user-space program. Libraries may do useful work without entering the kernel.



OSC 10e, Chapter 1, slide 7
Silberschatz, Galvin & Gagne

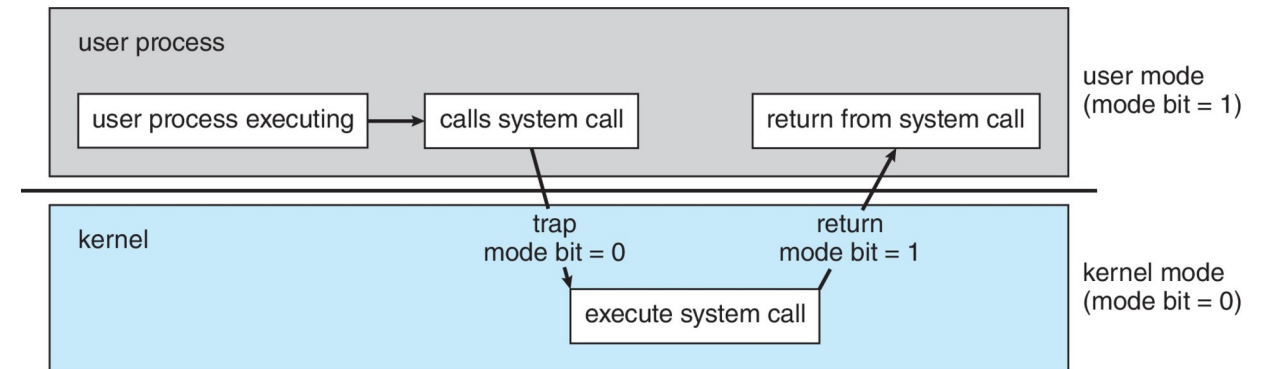
User mode and kernel mode

Application instructions normally execute with restricted privileges; the program cannot perform every hardware operation directly.

A system call uses controlled entry into the kernel, which checks and services the request.

The kernel can return to the same process. Entering kernel mode does not require scheduling another process.

Root is a user identity affecting permissions. It is different from the CPU's kernel execution mode.



Three reasons to enter the kernel

System call: the running program requests an OS service.

Exception: the current instruction causes an event, such as a page fault.

Device interrupt: hardware signals an event, such as I/O completion.

An interrupt does not necessarily cause a process switch.

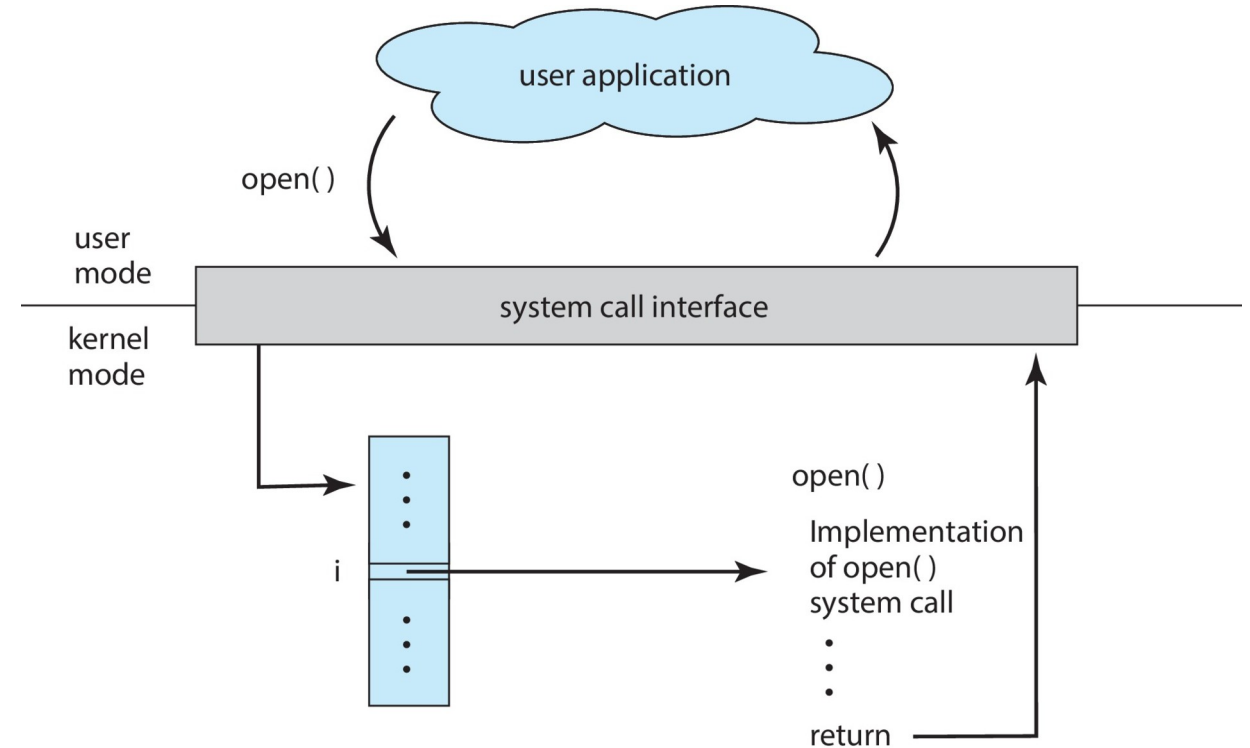
A function call is not always a system call

A library can do work entirely in user space.

A wrapper can request a kernel service.

Buffered output can combine several printf calls into one write.

API names and the operations shown by a tracer may differ.



OSC 10e, Chapter 2, slide 17
Silberschatz, Galvin & Gagne

A program is a description; a process executes

Program: executable instructions and static data.

Process: a running instance, with state and resources.

Run the same executable twice: two processes can result.

Their progress, registers and memory contents can differ.

One executable, two independent processes

Two users can start the same editor executable. Each running instance has its own PID, current instruction position and writable data. One editor can wait for keyboard input while the other is saving a file.

The kernel may share physical pages containing read-only code. That implementation choice does not make the editors' ordinary writable variables shared: the address-space abstraction preserves their separate views.

Prediction: if one editor crashes, must the other terminate? Normally no. Process isolation limits direct interference, although both can still depend on the same external file, service or machine.

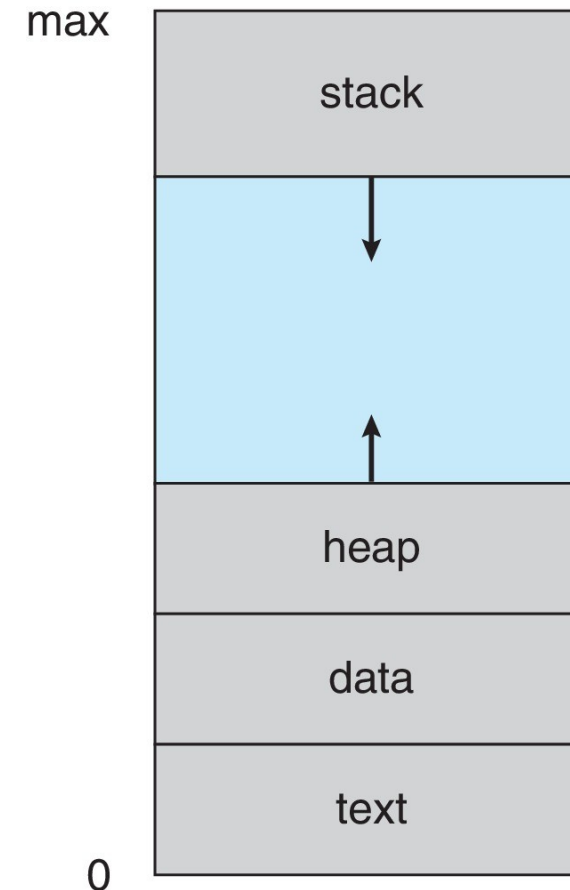
What belongs to a process?

Code, static data, dynamically allocated objects and stacks occupy a process's virtual address space.

The stack supports active calls; the heap supports dynamic allocation. These are useful conceptual regions.

Registers and the program counter describe current execution; they are not extra regions in this memory diagram.

The drawing is schematic. Real layouts include mappings and libraries, and their addresses and growth directions vary.



OSC 10e, Chapter 3, slide 6
Silberschatz, Galvin & Gagne

Locate data before explaining its lifetime

A local variable normally belongs to an active function invocation; its lifetime ends when execution leaves the relevant scope. A dynamically allocated object remains available until released, subject to the language's rules.

A pointer is an address value, not ownership or permission by itself. Keeping a pointer after an object's lifetime ends does not keep the object alive. Likewise, copying a pointer to another process does not establish shared memory.

The diagram's text, data, heap and stack regions are conceptual. Libraries, mapped files and guard regions also occupy virtual addresses. Exact locations and growth directions are not universal.

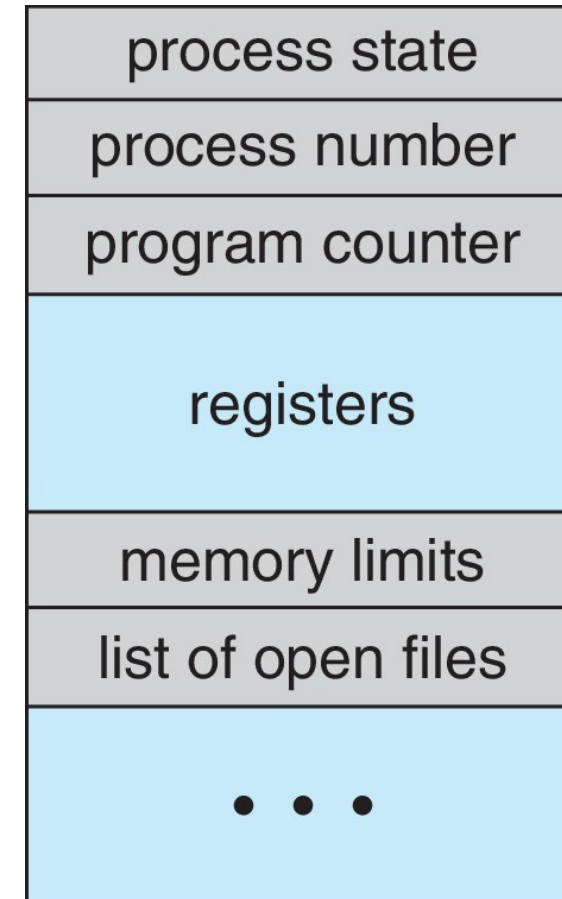
The kernel must remember more than memory

Identity and state: PID, relationships, scheduling information.

Saved execution context: instruction position and registers.

Resource references: memory mappings and open files.

A PCB is a conceptual summary of these records.



OSC 10e, Chapter 3, slide 10
Silberschatz, Galvin & Gagne

A process record is not a memory snapshot

To resume a task, the kernel needs its saved processor context and references to its resources. A program counter tells where to continue; saved registers preserve values that were live when execution stopped.

The address space usually remains represented by memory mappings and resident or backing storage. A context switch does not copy every byte of that space into the PCB and then copy it back.

The PCB picture summarizes responsibilities. Real kernels split information among related structures, and per-thread context must be distinguished from resources shared by the whole process.

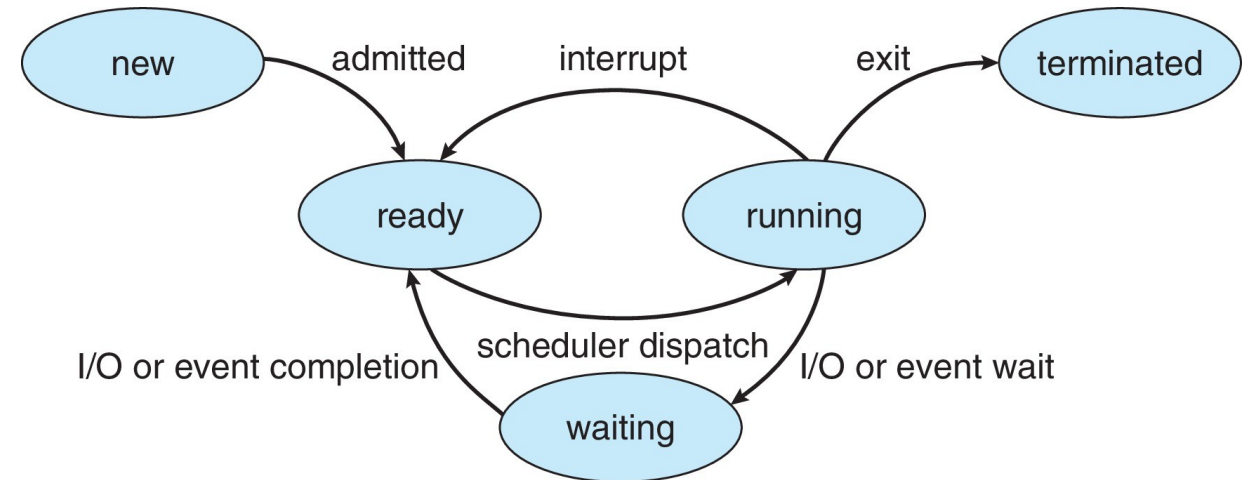
Running, ready and waiting

Running means currently executing on a CPU.
Ready means able to execute but awaiting selection.

Waiting means an event is missing; more CPU time alone cannot complete the blocked operation.

When input arrives, a waiting task normally becomes ready. It need not run immediately.

This is a simplified model; real kernels have more states and commonly schedule threads or tasks.



OSC 10e, Chapter 3, slide 9
Silberschatz, Galvin & Gagne

Ready and waiting require different remedies

Ready: a calculation has all its input and could execute immediately if selected. Its current obstacle is processor allocation, so scheduling policy affects its delay.

Waiting: a client has requested a reply that has not arrived. It cannot finish the operation merely by receiving another CPU time slice. The missing event must occur first.

A responsive system can run ready work while other tasks wait. This improves use of available resources, but it does not shorten the external event itself. “Not running” is therefore an incomplete diagnosis.

Work through a state trace

P is selected and starts calculating: ready $\rightarrow$ running.

P requests input that is not available: running $\rightarrow$ waiting.

The input arrives: waiting $\rightarrow$ ready.

P is selected again: ready $\rightarrow$ running.

Its time slice expires: running $\rightarrow$ ready.

Worked trace: who can run now?

Assume one CPU. At time 0, A runs and B is ready. A requests unavailable disk data: A becomes waiting and B can be selected. This is a possible execution order, not a measured trace.

While B runs, A's I/O completes. A becomes ready. B may continue until a scheduling decision selects A; I/O completion alone does not prove an immediate switch.

If B then blocks for keyboard input, it becomes waiting. A can run because its earlier dependency is satisfied. Track two separate facts at every step: eligibility to run and current CPU ownership.

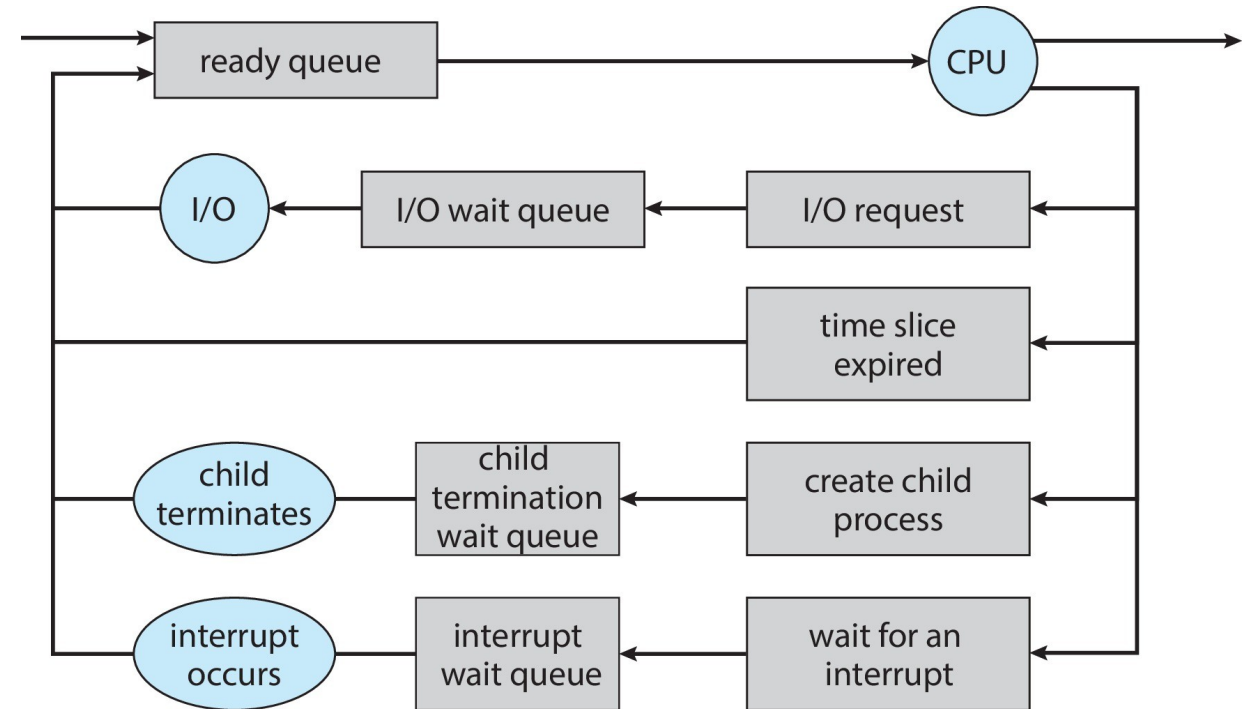
Queues represent different kinds of delay

Ready work competes for processor time.

Waiting work is associated with an event or resource.

Completing the event moves work toward execution.

Scheduling policy decides which ready task runs next.



OSC 10e, Chapter 3, slide 15
Silberschatz, Galvin & Gagne

CPU work and I/O waits alternate

A program's execution often contains CPU bursts separated by waits. Parsing buffered data may be CPU work; requesting unavailable input may lead to a wait. The same application can change behavior across phases.

A CPU-bound workload spends much of its time ready or running. An I/O-bound workload frequently waits for external operations. These descriptions are relative to the workload and system, not permanent labels attached to an executable.

When one process waits, another ready process can use the CPU. This is the basic motivation for multiprogramming. Week 2 develops the scheduling policies and the instructor's mechanic/car analogy.

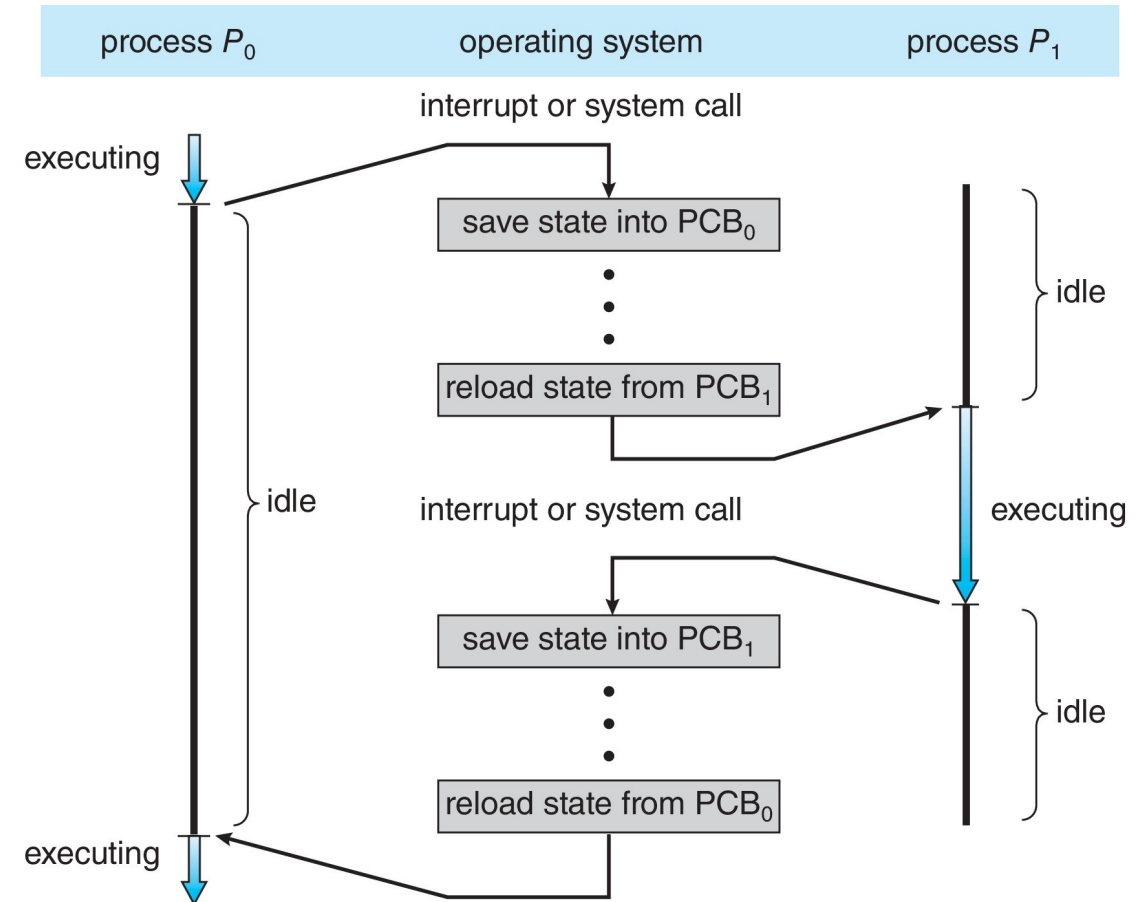
A context switch preserves continuity

Save the outgoing task's execution context.

Select and restore another runnable task.

Resume that task where it previously stopped.

Switching costs time and can disturb caches and translations.



OSC 10e, Chapter 3, slide 16
Silberschatz, Galvin & Gagne

What changes when execution moves from P to Q?

Before the switch, the processor contains values belonging to P's current execution. The kernel records the necessary state so P can later continue correctly; it then restores Q's saved context.

When P and Q use different address spaces, the execution environment must select the appropriate mappings. This need not mean copying all memory or flushing every cache; details depend on the architecture and kernel.

The direct save/restore work and indirect working-set effects are overhead. A useful scheduling policy balances responsiveness and fairness against the cost of switching too frequently.

Kernel entry is not a context switch

P calls a service and returns: P may run throughout.

P blocks inside the service: another task may run.

A timer interrupt occurs: the scheduler may keep P or select Q.

Ask two questions: did privilege change; did the task change?

Classify the event, then classify the switch

Case A: P requests a service; the result is immediately available; the kernel returns to P. There is kernel entry and return, but no task switch is required.

Case B: P requests unavailable input and blocks; Q is selected. Kernel entry is followed by a task switch because P is no longer runnable.

Case C: a timer interrupt arrives while P runs. The scheduler can keep P or choose another task. The interrupt is evidence of an event, not evidence that Q necessarily ran.

Checkpoint: explain the observations

A process is waiting for a network reply. Would a free CPU help?

A read completes immediately. Must another process run?

Two programs finish “at the same time.” Does that prove parallelism?

Explain each answer using state, mode or CPU ownership.

Checkpoint answers and the first-half boundary

A free CPU cannot supply a missing network reply. The waiting task needs the event; only after it becomes runnable can CPU selection help it continue.

An immediately completed read can return to the same task. Kernel execution on behalf of P must not be confused with scheduling a different task.

Similar finish times do not prove parallelism. One core can interleave several activities. After the break, we will follow process creation and examine what changes when several threads share one address space.

Launching a command: three separate operations

fork: create a child process.

exec: replace the calling process's program image.

wait: observe a child's termination and collect its status.

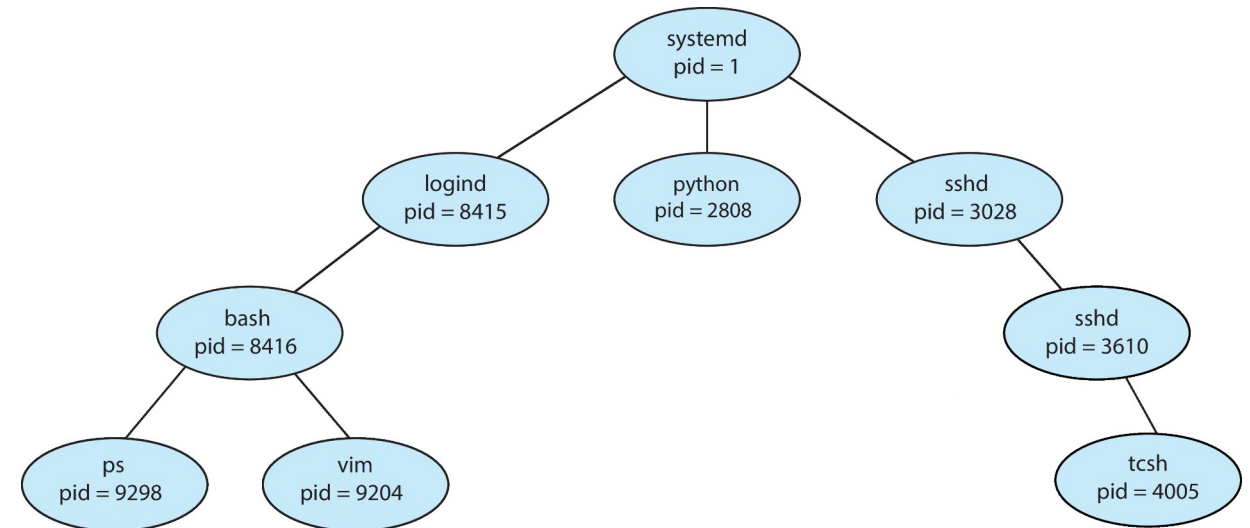
A shell can use optimized mechanisms; this is the conceptual model.

Read a process tree as a relationship map

A process tree records parent–child relationships, not which process is currently running. A parent may be waiting, computing or handling other work while its children execute.

The displayed PIDs are examples from the textbook. On your own machine, process names and ancestry depend on the shell, service manager and environment.

A parent can create several children. A child can create further descendants. The tree alone does not establish communication paths, shared data or the order in which the processes finish.



OSC 10e, Chapter 3, slide 22
Silberschatz, Galvin & Gagne

fork returns into two processes

Parent receives the child PID; child receives zero.

Both continue after fork, following different branches.

Ordinary memory is logically independent after the fork.

Linux commonly delays copying pages until a write requires it.

Follow the return values from fork

Consider a successful fork in a single-threaded program. The parent receives a positive result containing the child's PID; the child receives zero. Both resume after the call, but an if statement selects different branches.

If fork fails, the original process receives -1 and no child was created. Error handling must happen before treating the result as a valid child identifier.

There is no general guarantee that the parent's next statement runs before the child's next statement. If the order matters, establish it with synchronization instead of assuming a convenient scheduler decision.

Copy-on-write preserves separate memory views

Immediately after fork, parent and child have logically separate writable address spaces with initially matching contents. A kernel can let their mappings refer to the same physical pages until one needs to write.

On a write that requires separation, the operating system arranges a private writable copy. The implementation saves unnecessary copying when the child soon replaces its image with exec.

The observable rule for an ordinary variable is unchanged: the child's assignment does not update the parent's variable. Copy-on-write is an optimization of copying semantics, not a shared-memory communication channel.

exec changes the program, not the PID

The calling process receives a new program image.

Successful exec does not return to the old program.

On failure, exec returns and the caller must handle the error.

A child can become `/bin/echo` while remaining the same child process.

Prepare the child before replacing its image

A shell can arrange a child's file descriptors before exec. For example, it can connect the child's standard output to a pipe so that the new program writes to another program's input.

Exec replaces code, data and the execution stack with the new program image. The PID remains the same. Some process attributes survive; others are reset or replaced according to the API's rules.

Open descriptors generally survive unless marked close-on-exec. This is why setup performed before exec can affect the new program. It is also why unintended descriptor inheritance can cause bugs.

A child runs a different program

```
pid_t child = fork();  
  
if (child == 0) { execl("/bin/echo", "echo", "child", (char *)0);  
  
    perror("execl"); _exit(127); }  
  
/* parent: check fork failure, then waitpid(child, ...) */
```

The accompanying source includes all error checks.

Launch example: predict the successful output

The child executes `/bin/echo`, which prints “child” and terminates. The parent calls `waitpid` for that child and prints its status only after termination has been collected.

Therefore the successful output is “child”, followed by “parent: child exit status = 0”. Waiting creates the required ordering; the example does not depend on the parent or child winning the first CPU time slice.

The complete `launch.c` checks fork failure, retries `waitpid` after `EINTR`, and interprets status with `WIFEXITED/WEXITSTATUS`. These checks belong to the program even when a slide shows only its central control flow.

Launch example: what happens when exec fails?

Replace `/bin/echo` with a nonexistent pathname in a copy of the source. Exec now returns with an error instead of replacing the child's program. The child prints an error message and exits with status 127.

The parent still waits successfully for its child; waiting itself did not fail. It observes exit status 127 and reports failure. Separate failure to launch the requested program from failure of the wait operation.

The child uses `_exit` after failed `exec`. In this single-threaded example that avoids an additional flush of inherited `stdio` buffers. Status 127 is the example's chosen failure convention, not a success return from `exec`.

wait is synchronization and cleanup

The parent can work while its child executes.

waitpid can block until the selected child terminates.

Termination leaves status information for the parent to collect.

A zombie is terminated; it is not still executing the program.

Termination, zombies and parent lifetime

A child that has terminated may retain a small kernel record until its status is collected. This is a zombie: its program is no longer executing, although an identifier and termination information remain.

A parent can perform useful work before waiting. Background execution does not remove the responsibility to arrange child-status collection; it changes when and how completion is handled.

A process whose parent exits is a different situation from a zombie. Unix-like systems arrange reparenting and eventual reaping; the exact responsible process depends on the environment. Do not infer either condition merely from “not running”.

Observe a process in Ubuntu

```
sleep 20 &
```

```
demo_pid=$!
```

```
ps -o pid,ppid,stat,comm -p "$demo_pid"
```

```
wait "$demo_pid"
```

A snapshot shows one moment; it is not a complete execution trace.

Interpret a ps observation carefully

PID identifies the observed process; PPID identifies its current parent. STAT summarizes state and may include additional flags. COMM names the command; it is not a complete explanation of the process's behavior.

On Linux, R includes running or runnable, so a snapshot does not directly reveal the textbook distinction between running and ready. A sleeping state also does not by itself identify the precise missing event.

Record what was actually observed. Do not substitute an invented PID or assume a screenshot captures every transition. If sleep has already terminated before ps runs, an absent process row is a timing outcome, not proof that launch failed.

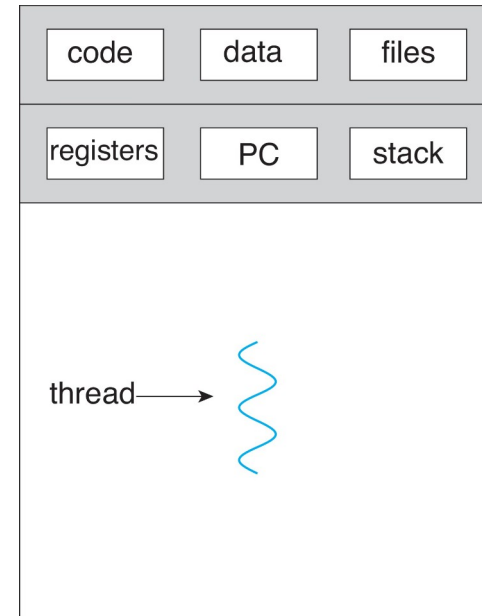
A thread is an execution stream

Each thread has its own instruction position, register state and stack for its active calls.

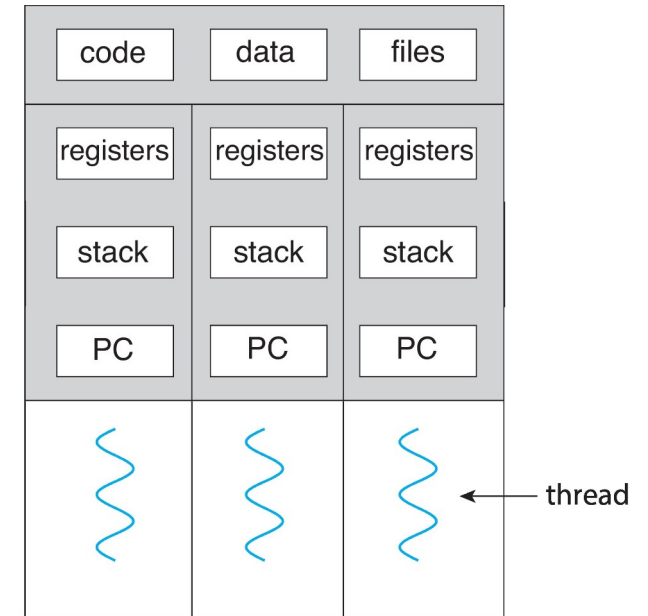
Threads within a process share the address space and many process resources, including open files.

A pointer can allow one thread to access an object another thread uses, including a live stack object.

Separate stacks organize execution. They do not impose memory-protection boundaries between threads.



single-threaded process



multithreaded process

Compare a process boundary with a thread boundary

Two ordinary processes have separate virtual address spaces. Exchanging data normally requires an explicit mechanism, such as a pipe or a shared mapping. This separation provides a useful protection boundary.

Two threads in one process share its address space. They have separate registers, instruction positions and stacks, but can access common objects. This reduces the need to move data across an interface.

The tradeoff is fault containment: an erroneous thread can corrupt shared memory used by other threads. Choosing threads because communication is convenient also means accepting responsibility for synchronization and object lifetime.

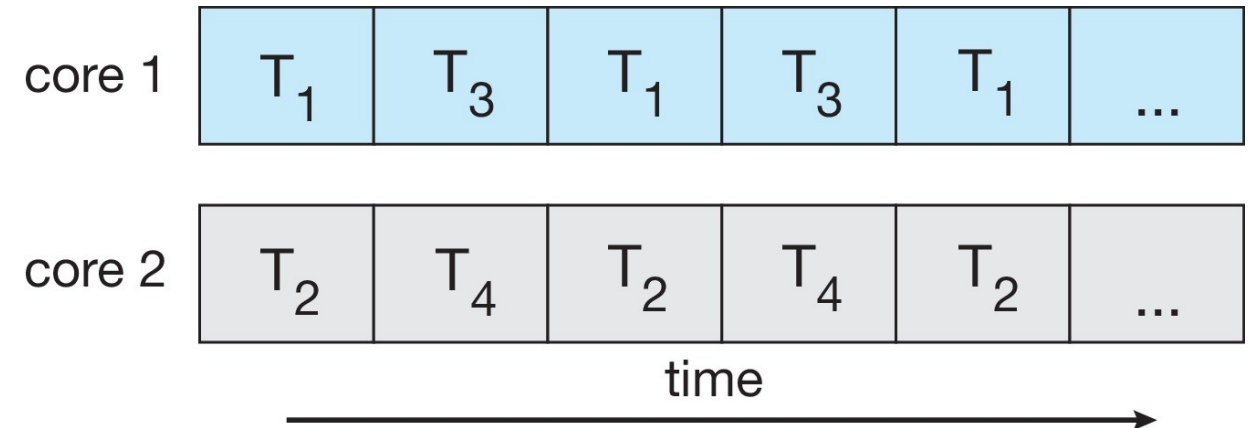
Concurrency and parallelism

Concurrency: several activities make progress over overlapping periods.

One core can interleave their execution.

Parallelism: activities execute simultaneously on different cores.

More threads do not automatically mean faster execution.



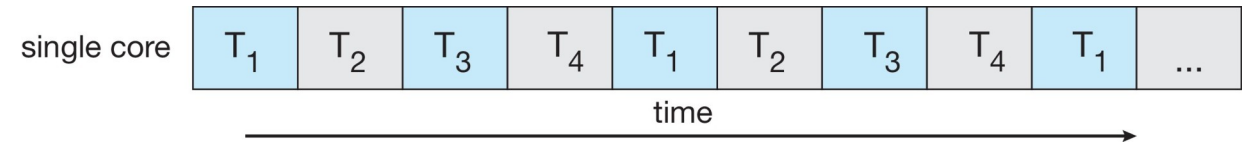
OSC 10e, Chapter 4, slide 9 (multicore figure)
Silberschatz, Galvin & Gagne

Interleaving can create apparent simultaneity

One core executes one thread at a time. T1, T2 and T3 progress over the same interval because the CPU alternates among them.

A user may perceive several activities as happening together. That observation demonstrates overlapping activity, not necessarily simultaneous instruction execution.

On multiple cores, some of these threads may truly run in parallel. Correct shared-data reasoning must allow the relevant interleavings in either case; one core does not make every multi-step operation indivisible.



OSC 10e, Chapter 4, slide 9 (single-core figure)
Silberschatz, Galvin & Gagne

More threads can add work without adding speed

If a task must perform one serial stage before its parallel stage, adding workers cannot remove that serial dependency. Workers also consume memory and require creation, scheduling and coordination.

A shared lock, limited memory bandwidth or slow storage can become the limiting resource. Beyond that point, extra runnable threads may mainly add contention and switching.

For this lecture, distinguish a design's potential for concurrency from a measured speedup claim. A fair performance comparison needs a stated workload, environment and measurement method; our diagrams are not benchmark results.

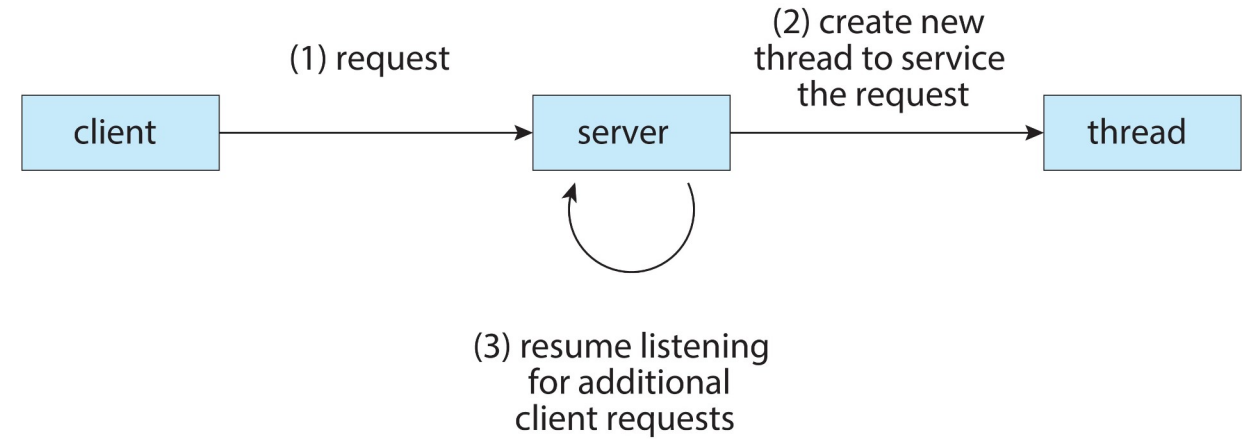
Why a server uses threads

One execution stream accepts incoming work.

Workers can service independent requests.

A blocked request need not stop every other request.

Shared caches and queues need coordination.



A bounded worker pool makes ownership explicit

A server can accept requests into a queue and have a bounded set of workers process them. The queue separates arrival from service and limits the number of active execution streams.

The design still needs rules: who owns each request, who may change the queue, what happens when it is full, and how shutdown wakes waiting workers. Sharing a queue does not answer these questions automatically.

Threads are one approach. Event-driven designs can also overlap waiting operations. The appropriate model depends on the work; the textbook's "create a thread" picture is a motivation, not a requirement for every server.

A shared variable needs a protocol

Suppose increment means: read counter, add one, write counter.

A reads 0; B reads 0.

A writes 1; B writes 1.

Two increments should produce 2, but this interleaving produces 1.

This is a conceptual trace, not valid unsynchronized C code.

Explain the lost update one step at a time

Initial counter = 0. A reads 0 into its temporary value. Before A writes, B also reads 0 into its own temporary value. Both now plan to store 1.

A writes 1; B subsequently writes 1. B's write replaces the same value rather than adding to A's already completed update. The intended final value 2 has been lost.

The lesson is to protect the complete operation or use a suitable atomic operation. This is a conceptual trace. Unsynchronized conflicting accesses in C can be undefined behavior, so do not promise this exact result from a racy C experiment.

Safe sharing in a small thread example

Main initializes a worker argument and starts a thread.

The worker computes a result in that shared object.

Main calls `pthread_join` before reading the result.

The object remains alive until the worker has finished.

See `examples/thread_join.c` for the complete program.

Joining establishes order, not mutual exclusion

In `thread_join.c`, main initializes `input = 7` and creates a worker. The worker writes `result = 49`. Main joins the worker before reading that result, so the read occurs after worker completion.

This arrangement is safe because main does not concurrently modify the input or read the result too early. The argument object also stays alive until the join has completed.

Joining one worker does not protect arbitrary simultaneous accesses among other running workers. If several threads update an object concurrently, a separate coordination mechanism may be required.

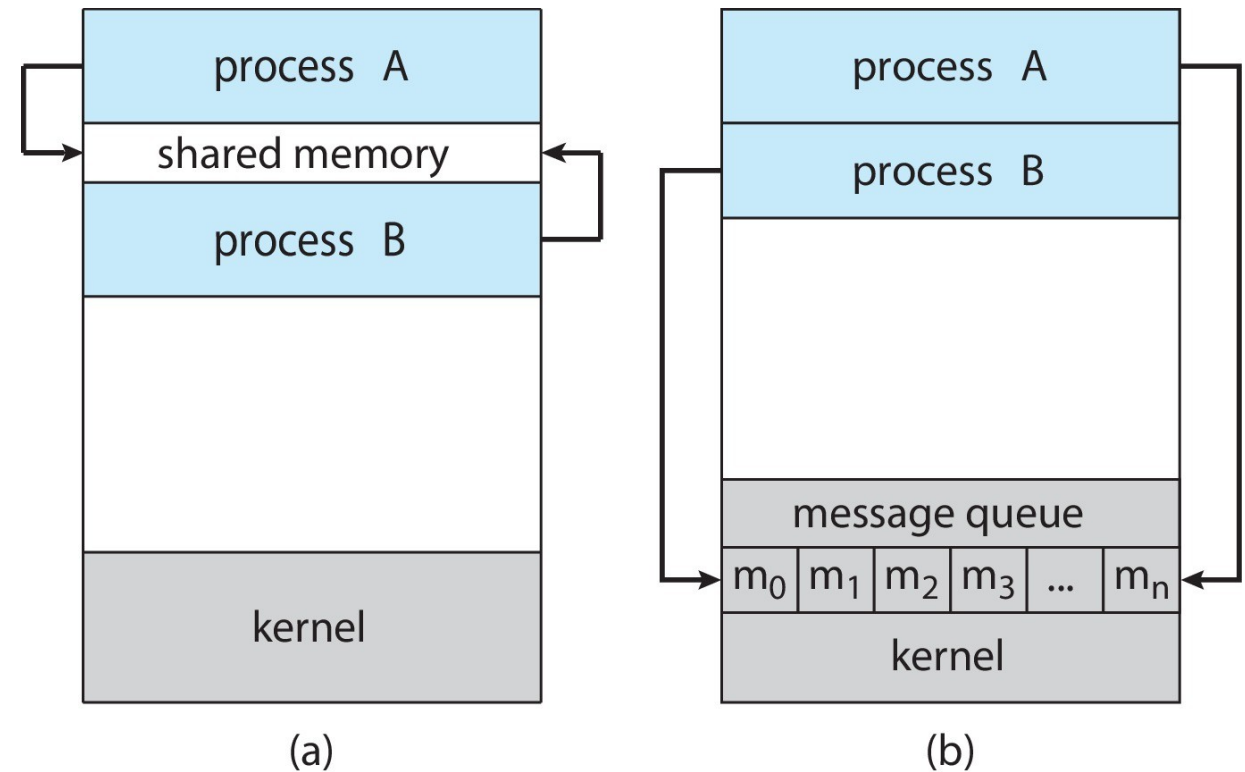
Processes communicate explicitly

Shared memory: processes access a shared region.

Message passing: data crosses a communication interface.

Both need agreement about meaning, order and ownership.

Sharing storage alone does not provide synchronization.



Choose a communication model deliberately

Shared memory provides a common region. Participants must agree on layout, ownership and when data is ready. A flag in that region still needs correct synchronization; merely sharing the flag is insufficient.

Message passing makes an interface explicit: send a request, receive a reply, or consume a stream with a defined format. It still needs error handling, ordering rules and a way to distinguish complete messages.

Neither approach eliminates protocol design. Ask what must cross the boundary, which participant may modify it, how completion is signaled, and what the receiver should do if the sender stops.

A pipe connects output to input

```
printf 'alpha\nbeta\n' | wc -l
```

The producer writes a byte stream.

The consumer reads bytes and counts newline characters.

Expected result: 2. Whitespace formatting may vary.

A pipe does not preserve application message boundaries.

A pipeline is a data-flow agreement

For `printf` followed by `wc -l`, the producer supplies bytes containing two newline characters. The consumer counts newline characters and prints 2. This result follows from the input, not from a particular scheduling order.

The shell arranges the input/output connection. It may execute `printf` as a builtin, so the command text alone is not a reliable count of newly created external processes.

A stream can be read in chunks different from the producer's writes. Programs that exchange records must define framing, for example a delimiter or length field, rather than assuming each read returns one complete application message.

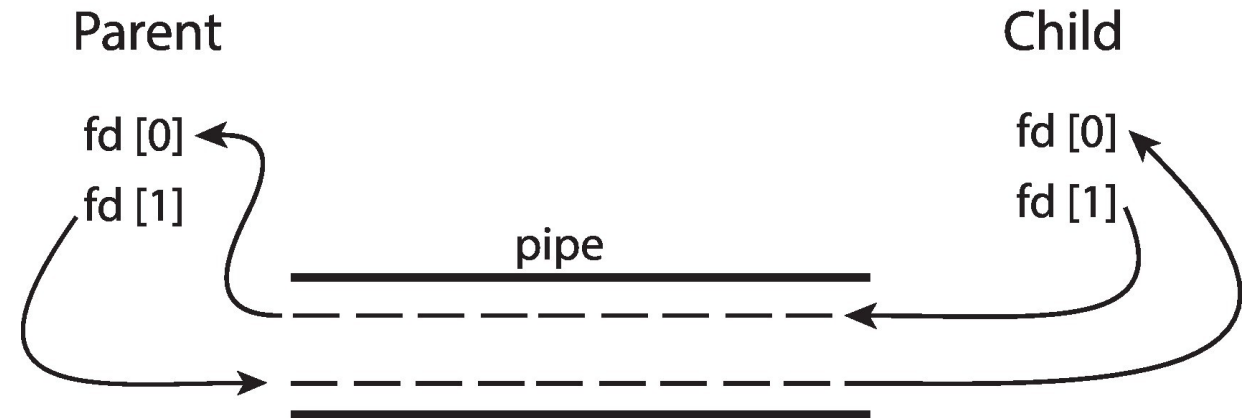
Blocking and end-of-file are different

Empty pipe + a writer still exists: a blocking read may wait.

No writers remain: after buffered bytes are read, read returns EOF.

A full pipe can make a blocking writer wait.

Unused pipe ends must be closed so completion can be observed.



Backpressure connects communication to waiting

If a producer generates data faster than a consumer removes it, the finite pipe buffer can fill. A blocking writer then waits for space; it is not simply ready for more processor time.

If a consumer is faster, it may drain the buffer and wait for data while writers still exist. This temporary emptiness differs from EOF after all writers have closed and buffered bytes are consumed.

These waits couple the participants' progress. A pipeline can therefore contain processes waiting for data or space as well as processes ready for the CPU. Resource cleanup and consumption rate both affect completion.

Trace a command from start to finish

The shell prepares a child and its input/output connections.

The child replaces its image with the requested program.

Execution alternates among computation, kernel services and waits.

A scheduler may interleave other runnable tasks.

Termination supplies a status; the parent eventually collects it.

Worked question: parent and child memory

Before fork: $x = 10$.

After fork: the child changes its ordinary local x to 20.

The parent waits, then prints its own x .

Answer: 10, assuming no explicit shared memory.

Waiting orders completion; it does not merge address spaces.

Worked question: why is the reader stuck?

A child finishes writing to a pipe and closes its end.

The parent reads all available bytes, then waits for more.

The parent accidentally kept a write-end descriptor open.

Answer: EOF cannot arrive while that writer reference remains.

Close every unused end in every participating process.

Explain a complete run without skipping boundaries

For each participant, identify its process identity, program image and open communication ends. An exec changes the image; it does not create an extra participant. A fork creates a new process with inherited state.

At every delay, ask whether the task is ready or waiting and identify the event that can make progress possible. At every kernel entry, ask separately whether another task was actually selected.

Finally, account for completion: close unused pipe ends, let readers observe EOF, and collect child status. A program that computed the right bytes can still fail to finish if these lifecycle obligations are missed.

Prepare for the next lecture

Observe a short background process and explain the snapshot.

Compile and run the launch and thread examples.

Predict each result before running it; explain any difference.

Bring one question about CPU sharing or waiting.

These are practice activities; no graded deadline is assigned here.

What you should now be able to explain

Program vs process; process vs thread.

Ready vs running vs waiting.

Kernel entry vs task switch.

fork vs exec vs wait.

Concurrency vs parallelism; shared memory vs a pipe.

Operating System Concepts, 10th edition: selected material from Chapters 1–4.

Original diagrams: Silberschatz, Galvin and Gagne; official OSC slides.

Linux man-pages: `fork(2)`, `execve(2)`, `waitpid(2)`, `pipe(7)`.

Optional: OSTEP chapters on processes, the process API and threads.

Links, explanations and worked answers are in the accompanying reading.