Week 3: Concurrency & Scheduling

From OS fundamentals to tail-latency experiments

> "Tail latency is often scheduling latency."

Today's Plan

Session 1: Lecture (~90 min)

- Part 0: OS Background Primer
  - Process, context switching, interrupt/trap basics
  - Thread model (kernel threads vs user threads)
- Part 1: Scheduler Fundamentals
  - FIFO/SJF/RR/MLFQ → why modern schedulers exist
  - Practical knobs for experiments: runnable queue, preemption, nice, affinity
- Part 2: Modern Context
  - Tail latency, runtimes, and measuring scheduling latency
- Part 3: Lab Preview
  - Lab 3: scheduling latency under CPU contention

Session 2: Lab Workshop (~90 min)

- Run Lab 3 baseline + contention + one mitigation

Learning Objectives

By the end of Week 3, you should be able to:

1. Explain what the kernel scheduler *actually does* at wakeup, enqueue, pick-next, and context switch
2. Describe how preemption works (timer interrupt, preemption points) and why atomicity matters
3. Reason about **scheduling latency** as a first-class performance metric (not just throughput)
4. Design a controlled experiment that demonstrates tail-latency inflation under CPU contention
5. Apply at least one mitigation (nice/affinity/cgroup) and verify the tail improves

Part 0: OS Background Primer

Process, thread, interrupt, context switch

Process Abstraction

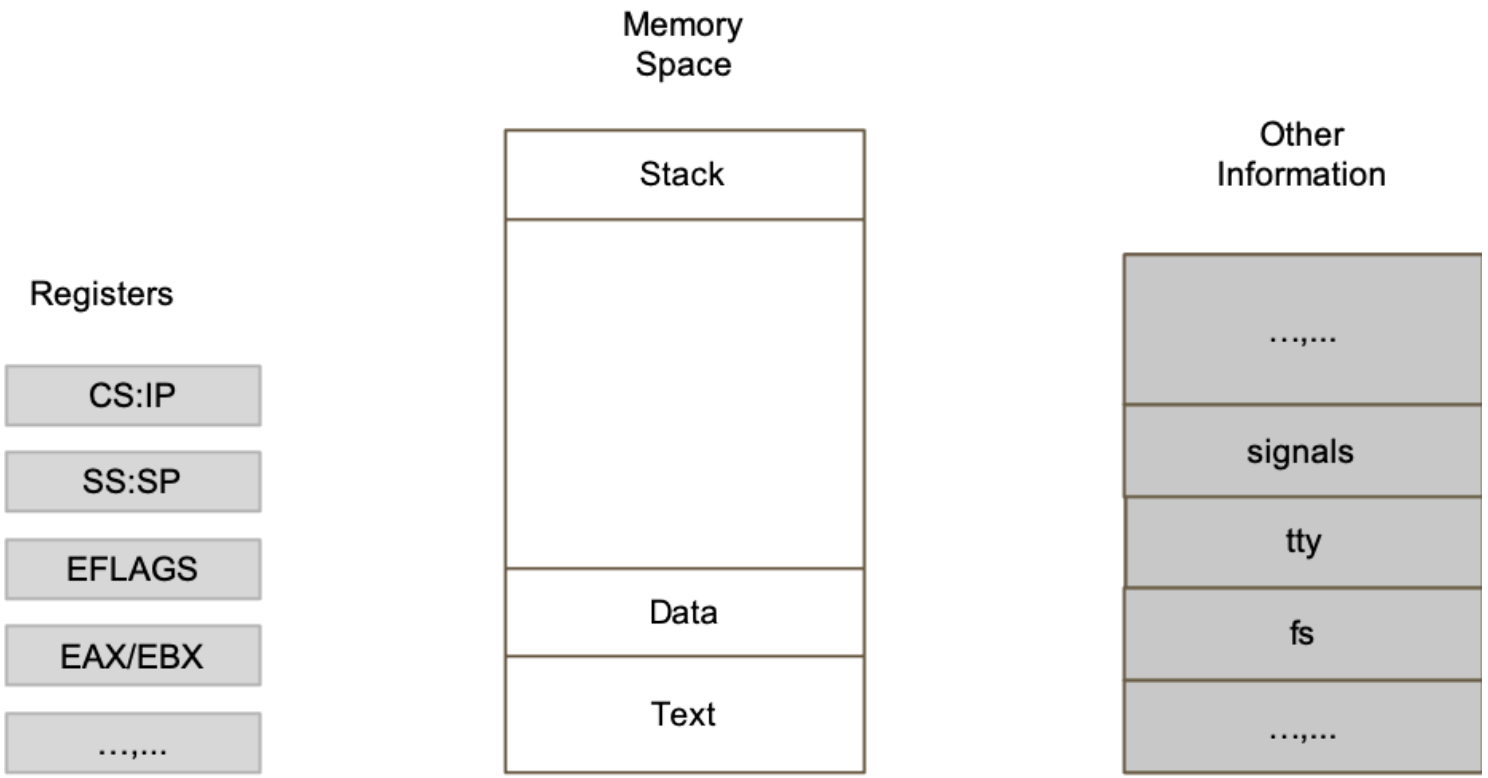
A **process** is an instance of a program in execution:

- Running with limited rights (protection)
- Isolated from other processes (address spaces)

It contains at least three parts:

- **Memory**: code + data (+ stack/heap)
- **Registers**: PC, SP, and many others
- **Access to devices/resources**: files, I/O, network, credentials

Process Abstraction

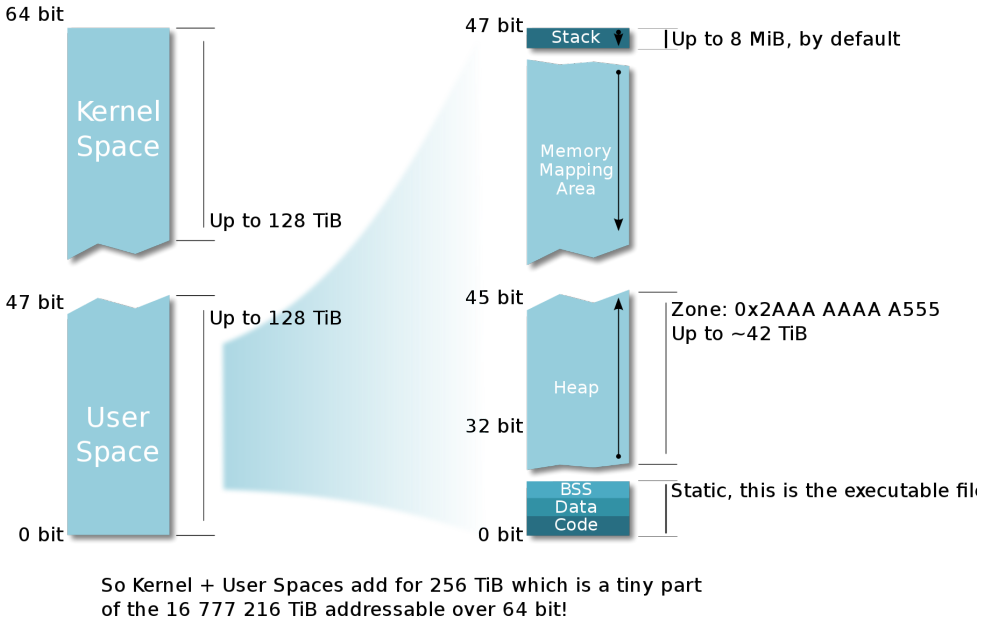


What constitutes a process? (machine state)

A process is more than “code running”. The OS must track (at least):

- **CPU register state**: PC/IP, SP, FP, general registers
- **Address space**: memory mappings (code, data, heap, stack), page tables
- **Open files**: file descriptors + offsets
- **Other context**: current directory, credentials, signal handlers, etc.

Address space (conceptual)



Key idea: **isolation + virtualization**. Each process *believes* it owns a private, contiguous memory.

OS data structures: PCB (`task_struct`) + queues

The kernel keeps per-process metadata in a **process control block (PCB)**:

- process state + accounting
- saved register context (for context switching)
- pointers to memory mappings/page tables
- scheduling info (priority/weight, runnable vs blocked)

PCBs are linked into kernel **queues/lists** (ready/run queue, wait queues for I/O, etc.).

How to Create a Process (high level)

Typical steps:

- Allocate and initialize the process descriptor (PCB / Linux `task_struct`)
- Allocate memory (address space + user stack + kernel stack)
- Load program code/data into memory (and set up page tables/mappings)
- Initialize user stack (`argc/argv/envp`) and default file descriptors (0/1/2)
- Return to user mode and start executing at the entry point (`_start` → `main`)

Key trick: the new process must be created such that it **looks like** it was interrupted and is now "returning" back to user mode.

Process API (Unix): a cheat sheet

Goal	Common syscalls
Create a process	`fork()` (and variants like `clone()`)
Run a new program	`execve()` / `execvp()`
Wait/reap	`wait()` / `waitpid()`
Exit/terminate	`exit()` / `_exit()` / `kill()`
Identify	`getpid()` / `getppid()`

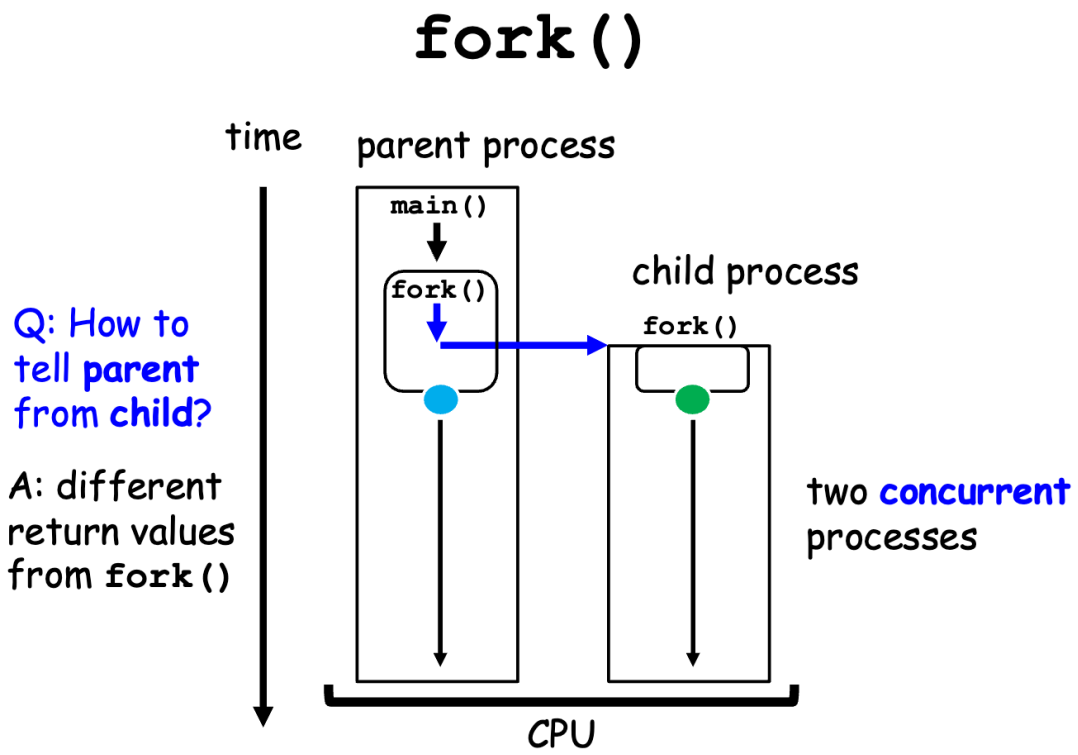
We'll use these mental models even when we later zoom into kernel internals.

`fork()`: create an (almost) exact copy

```
pid_t rc = fork();
if (rc < 0) {
  // fork failed
} else if (rc == 0) {
  // child
} else {
  // parent (rc == child's PID)
}
```

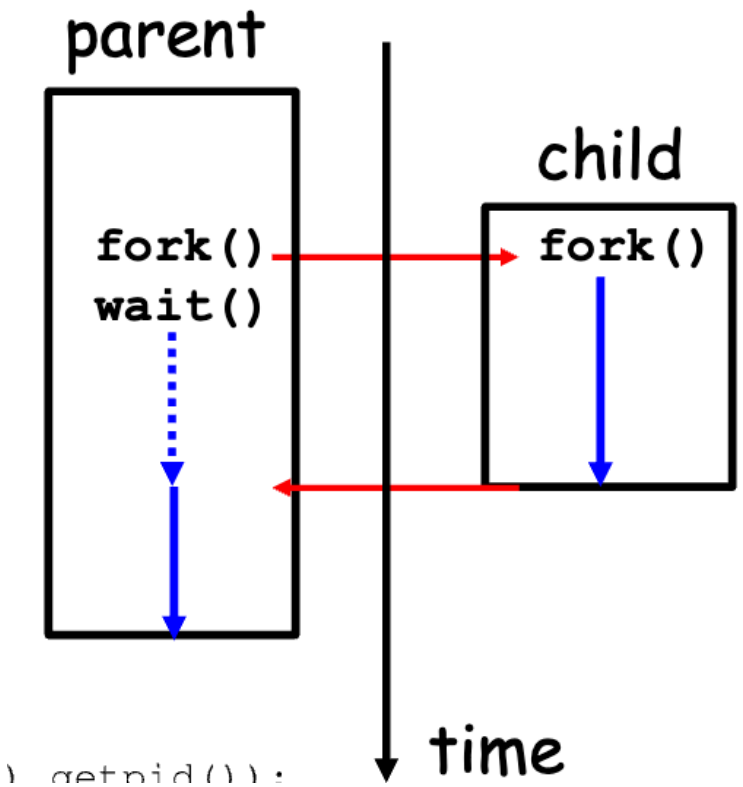
- Both parent and child return from `fork()`
- Parent returns **child PID**, child returns **0**
- On a single CPU, “who runs next” is **not deterministic**

`fork()`: create an (almost) exact copy



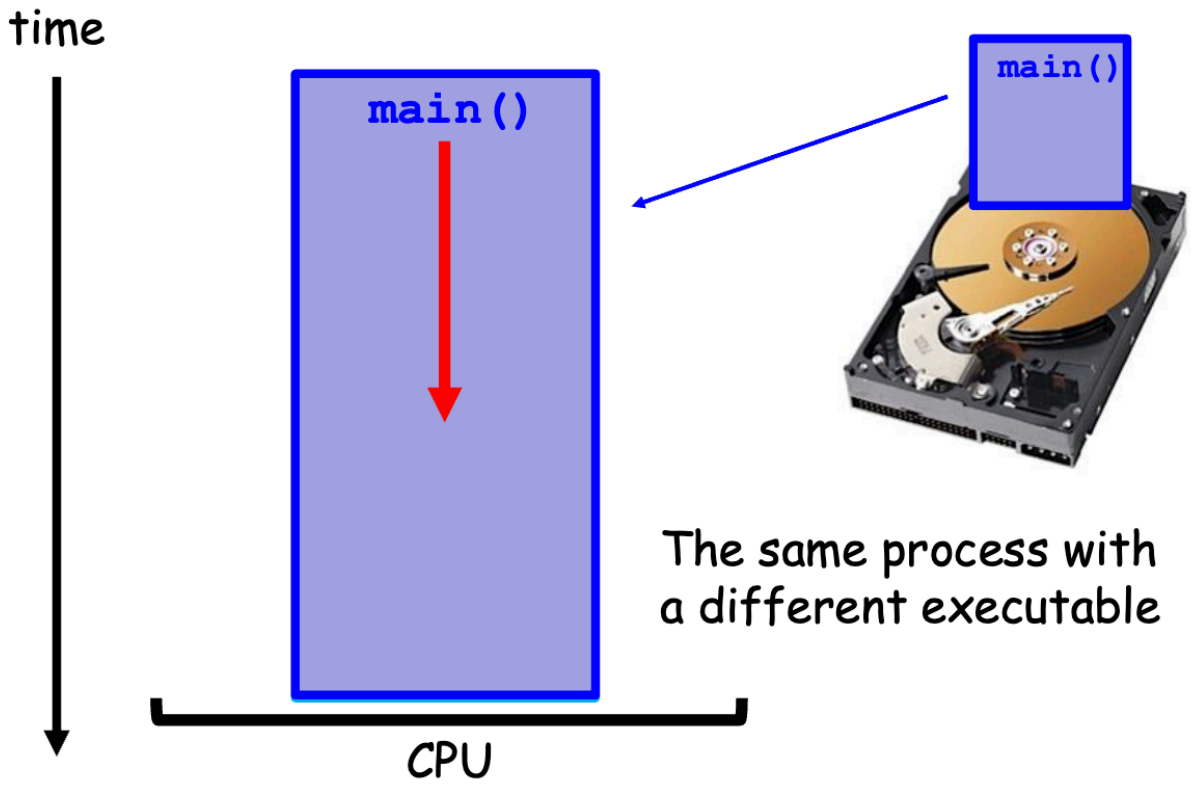
`wait()`: ordering + zombies

- Parent can call `wait()` to block until a child exits
- If a child exits but the parent hasn't waited yet, the child becomes a **zombie**
  - (resources mostly freed; exit status retained until reaped)



`exec()`: replace the current program

- Does **not** create a new process
- Overwrites the process's code/data and reinitializes stack/heap
- On success, **never returns** (control transfers to the new program)



Why `fork()` + `exec()` split? (shell redirection trick)

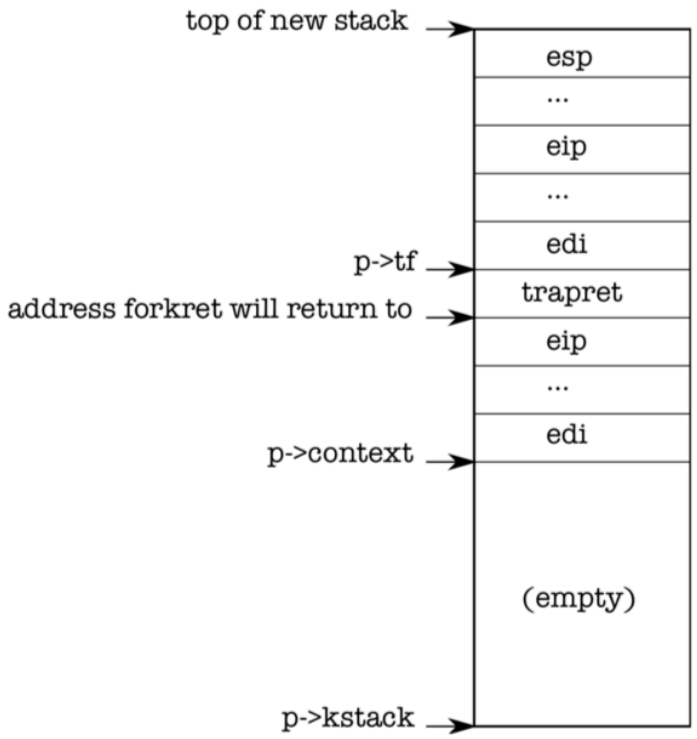
Process Creation Example (xv6)

```
  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++)
    if(p->state == UNUSED)
      goto found;

  release(&ptable.lock);
  return 0;

found:
  p->state = EMBRYO;
  p->pid = nextpid++;
```

- Check process table to locate a free entry
- Set status and assign a unique PID

Process Creation: “fork returns twice” (xv6)

```
// Allocate kernel stack.
if((p->kstack = kalloc()) == 0){
  p->state = UNUSED;
  return 0;
}
sp = p->kstack + KSTACKSIZE;

// Leave room for trap frame.
sp -= sizeof *p->tf;
p->tf = (struct trapframe*)sp;

// Set up new context to start executing at forkret,
// which returns to trapret.
sp -= 4;
*(uint*)sp = (uint)trapret;

sp -= sizeof *p->context;
p->context = (struct context*)sp;
memset(p->context, 0, sizeof *p->context);
p->context->eip = (uint)forkret;

return p;
```



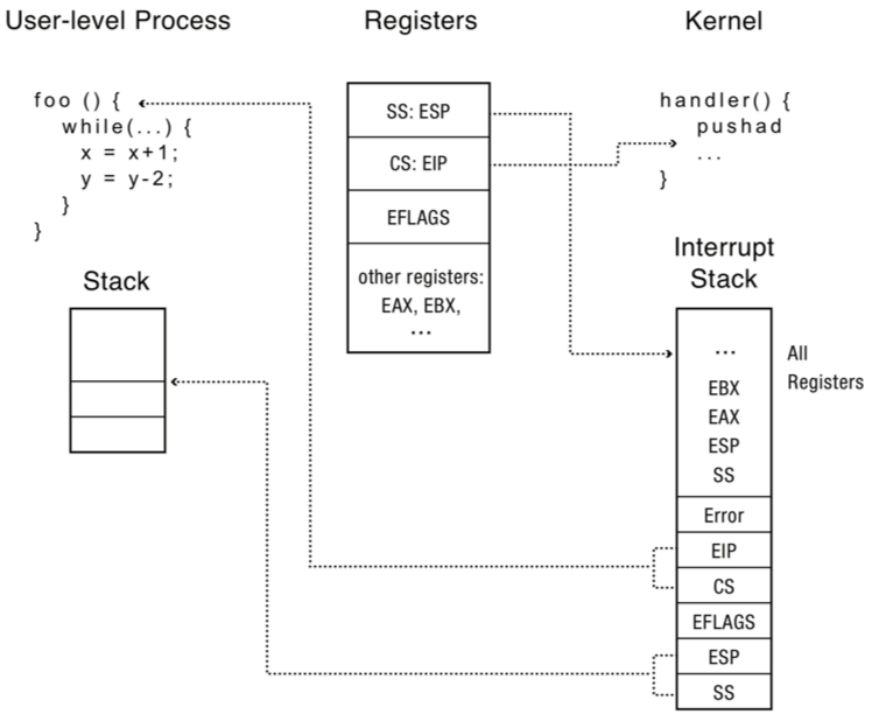
- `p->context->eip = forkret`
- `forkret` → `trapret`
- By constructing the kernel stack carefully, the child will later “return” into user mode

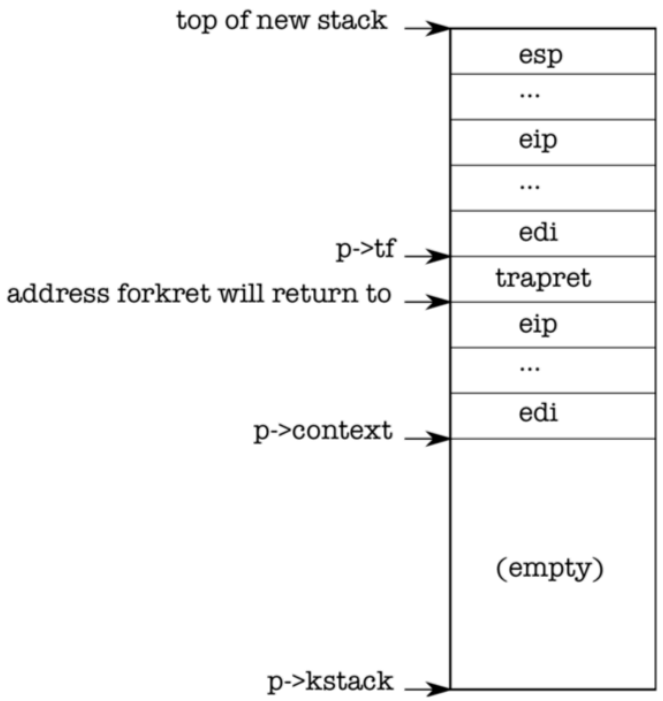
Note: Why this stack trick works

- `p->context->eip = forkret` causes the kernel to start executing `forkret` for the new process
- `forkret` returns to `trapret`
- `trapret` restores user registers from the trap frame, then jumps into user code (`eip`)

Result: **process creation** and **context switching** can share the same return-from-trap path.

Recap: Interrupts/Traps (xv6 example)





Recap: Interrupts/Traps (what gets saved)

A process transfers into the kernel via an **interrupt/trap** mechanism:

- hardware + kernel code save user registers on the process's **kernel stack**

When creating a new process:

- write values at the top of the new kernel stack *as if* it had entered the kernel via an interrupt
- then reuse ordinary return-from-trap code to start it

Process Context Switch: what must be true

Context switch should be **transparent**:

- usually triggered by a timer interrupt

Context switch must handle **contention**:

- multiple CPUs switching concurrently
- interrupts arriving during switching

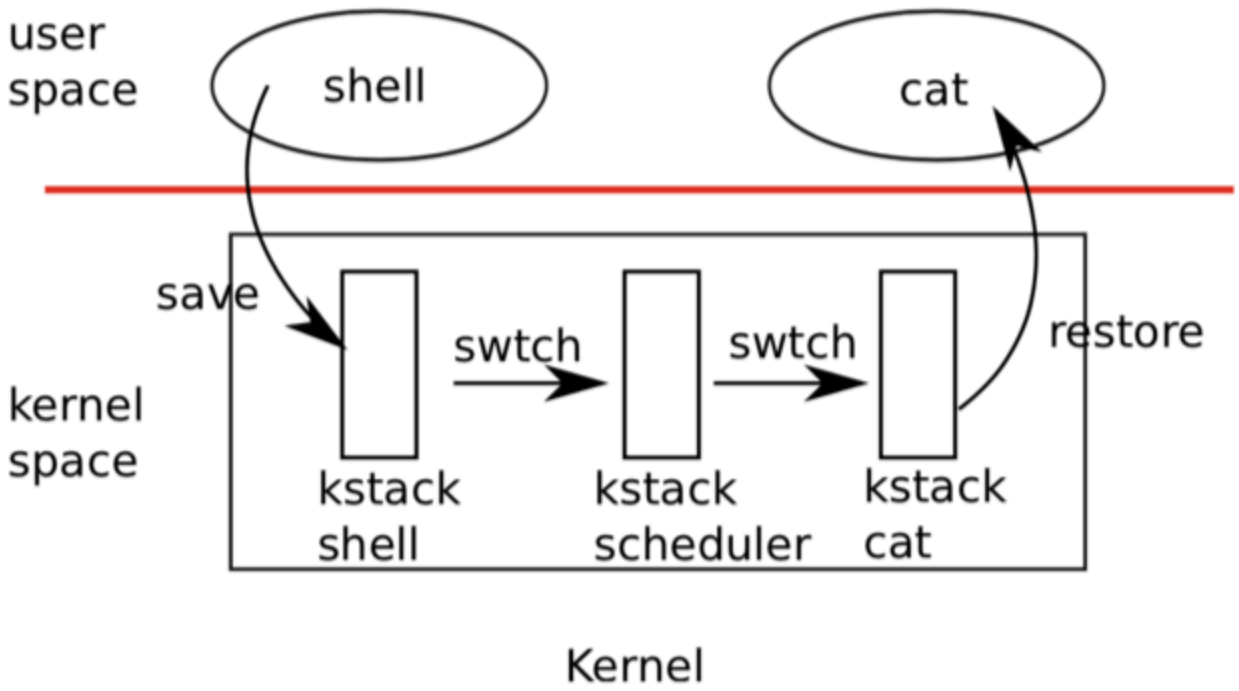
Context switch must handle **process exit** safely:

- free resources without races/use-after-free

Process Context Switch: a 4-step view

1. transition `user → kernel` on the old process's kernel stack
2. switch to the **CPU's scheduler context** (scheduler runs on its own stack)
3. switch to the new process's kernel context
4. `trapret` back to user mode

Process Context Switch (xv6 picture)



Scheduler runs on its own kernel stack.

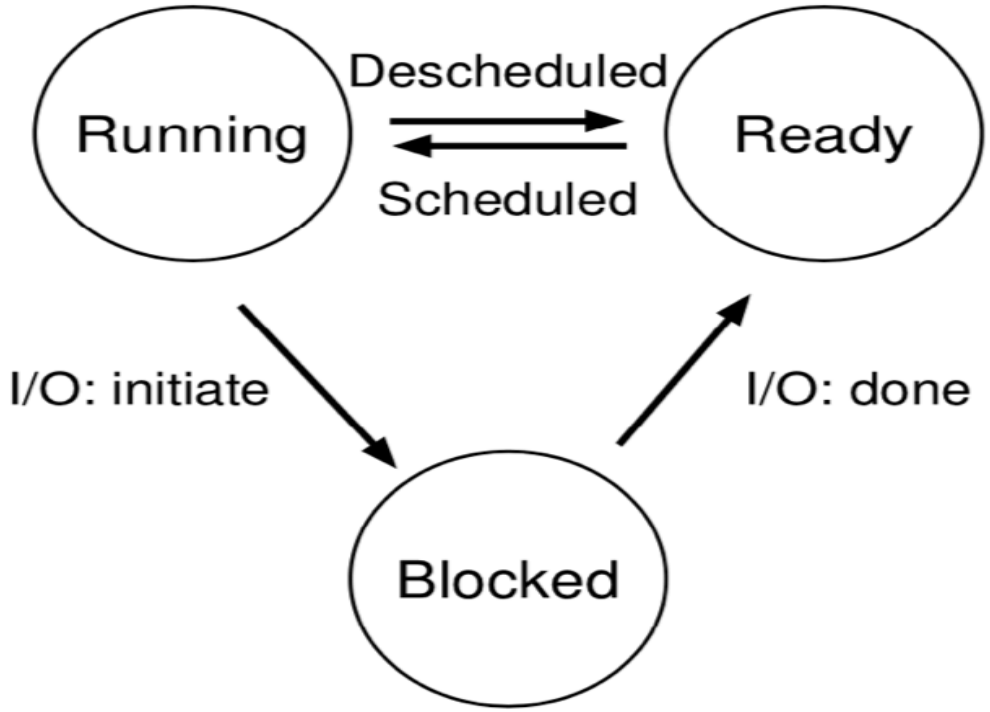
`swtch(old, new)` (xv6 intuition)

`context switch` implementation typically:

- saves registers of old context on old stack
- stores old stack pointer into `old->sp`
- loads new stack pointer from `new->sp`
- restores registers and returns into the new context

```
# Context switch
#
#   void swtch(struct context **old, struct context *new);
#
# Save the current registers on the stack, creating
# a struct context, and save its address in *old.
# Switch stacks to new and pop previously-saved registers.

.globl swtch
swtch:
  movl 4(%esp), %eax
  movl 8(%esp), %edx

  # Save old callee-save registers
  pushl %ebp
  pushl %ebx
  pushl %esi
  pushl %edi

  # Switch stacks
  movl %esp, (%eax)
  movl %edx, %esp

  # Load new callee-save registers
  popl %edi
  popl %esi
  popl %ebx
  popl %ebp
  ret
```

Process State Diagram



Thread Concept

A **thread** is a single execution sequence that represents a separately schedulable task.

- Single execution sequence: each thread executes instructions sequentially
- Separately schedulable: OS can run/suspend it at any time

Question to think about: **Is a kernel interrupt handler "a thread"?** (conceptually similar, but implemented differently)

Motivation for Threads

OS often needs to handle multiple things at the same time:

- process execution
- interrupts
- background activities

Threads are an abstraction that helps humans write concurrent programs:

- write each task as a sequential stream
- the streams interleave in time

Process vs. Lightweight Process vs. Thread (Linux view)

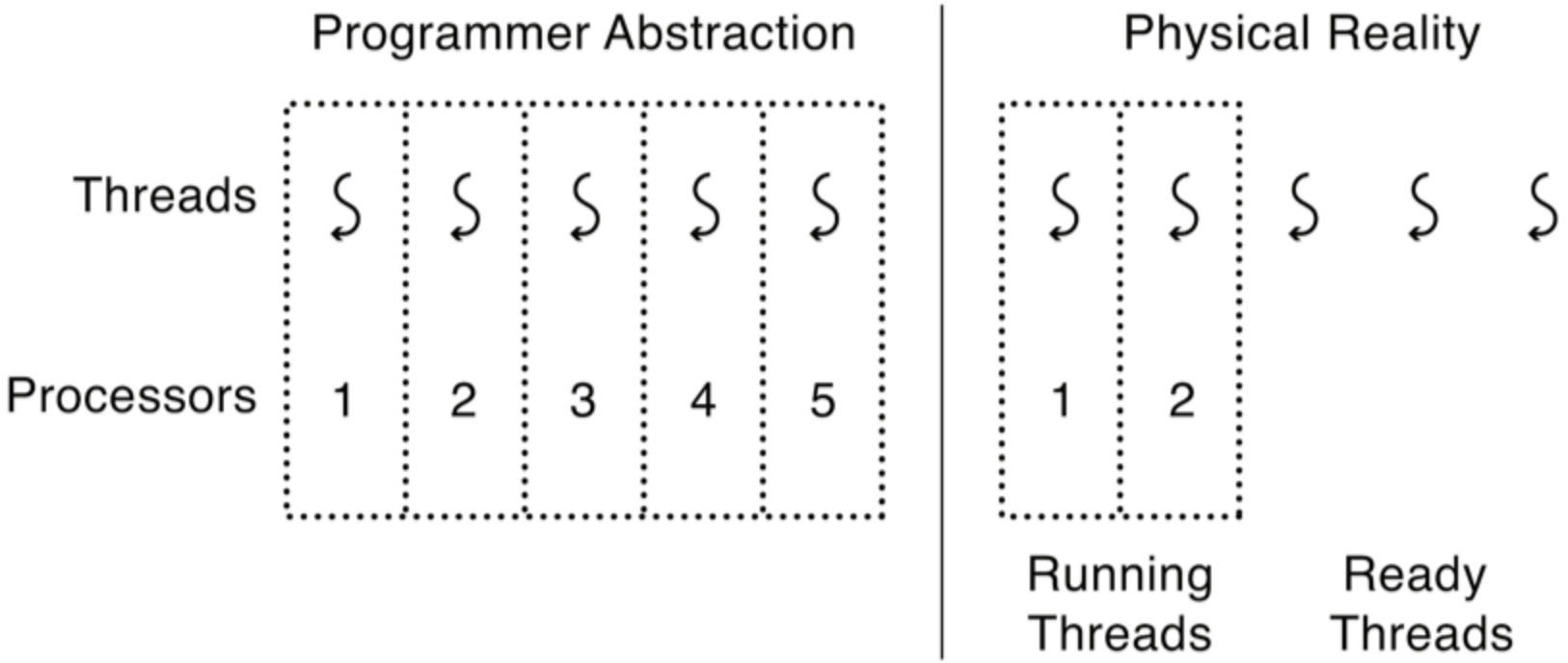
Historical note:

- Older systems could implement “threads” purely in user mode
- Problem: **blocking system calls** block the entire process

Modern Linux (and most Unix-like OSes):

- threads are implemented as **lightweight processes** (each has a kernel scheduling entity)
- threads share address space/files/etc, but can be scheduled independently

Thread Abstraction (picture)

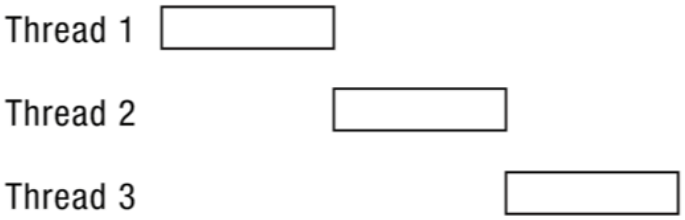


Programmer vs. Processor View

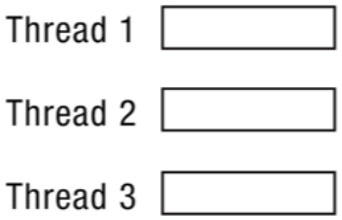
Programmer's View	Possible Execution #1	Possible Execution #2	Possible Execution #3
.	.	.	.
.	.	.	.
.	.	.	.
x = x + 1;	x = x + 1;	x = x + 1;	x = x + 1;
y = y + x;	y = y + x;	.....	y = y + x;
z = x + 5y;	z = x + 5y;	Thread is suspended.	.....
.	.	Other thread(s) run.	Thread is suspended.
.	.	Thread is resumed.	Other thread(s) run.
.	.	.....	Thread is resumed.
		y = y + x;	.....
		z = x + 5y;	z = x + 5y;

Possible Executions (interleavings)

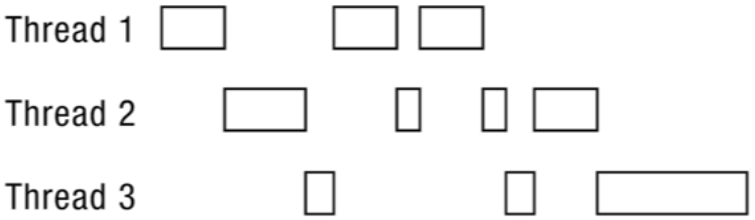
One Execution



Another Execution



Another Execution



POSIX Threads: key operations

- `pthread_create(thread, attr, func, args)`
  - create a new thread running `func(args)`
- `pthread_join(thread, retval)`
  - wait for thread termination

`pthread_exit()` terminates only the calling thread; `exit()` terminates the entire process.

POSIX Threads

```c
#include <stdio.h>
#include <pthread.h>

void* f(void* p) {
  printf ("%s\n", p);
  return NULL;
}

int main () {
  pthread_t teach, student[50];
  char pm[] = "Hello, my name is Dong.";
  char sm[] = "Hello Dong!";

  pthread_create(&teach, NULL, f, pm); //create a new thread
  pthread_join (teach, NULL); //wait for completion

  //create 50 threads
  for (int i=0; i < 50; ++i)
    pthread_create(&student[i], NULL, f, sm);

  //wait for the 50 threads to complete
  for (int i=0; i < 50; ++i)
    pthread_join(student[i], NULL);
  return 0;
}
```

Fork/Join Parallelism

Create children threads and wait for completion:

- common in servers: one worker per connection/request
- common in algorithms: merge sort, parallel memory copy

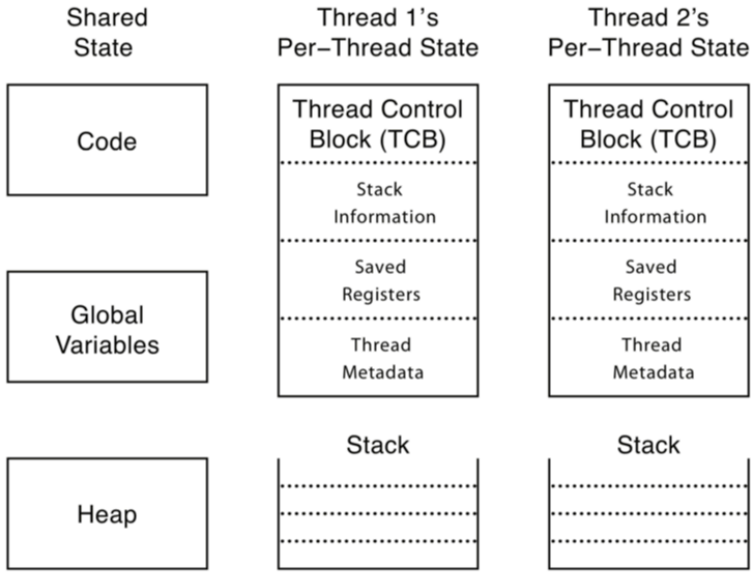
```
// Zero a block of memory using multiple threads.
void blockzero (unsigned char *p, int length) {
    int i, j;
    thread_t threads[NTHREADS];
    struct bzeroparams params[NTHREADS];

// For simplicity, assumes length is divisible by NTHREADS.
    assert((length % NTHREADS) == 0);
    for (i = 0, j = 0; i < NTHREADS; i++, j += length/NTHREADS) {
        params[i].buffer = p + i * length/NTHREADS;
        params[i].length = length/NTHREADS;
        thread_create_p(&(threads[i]), &go, &params[i]);
    }
    for (i = 0; i < NTHREADS; i++) {
        thread_join(threads[i]);
    }
}
```

Thread Implementation (what's inside a thread system)

Typical per-thread state:

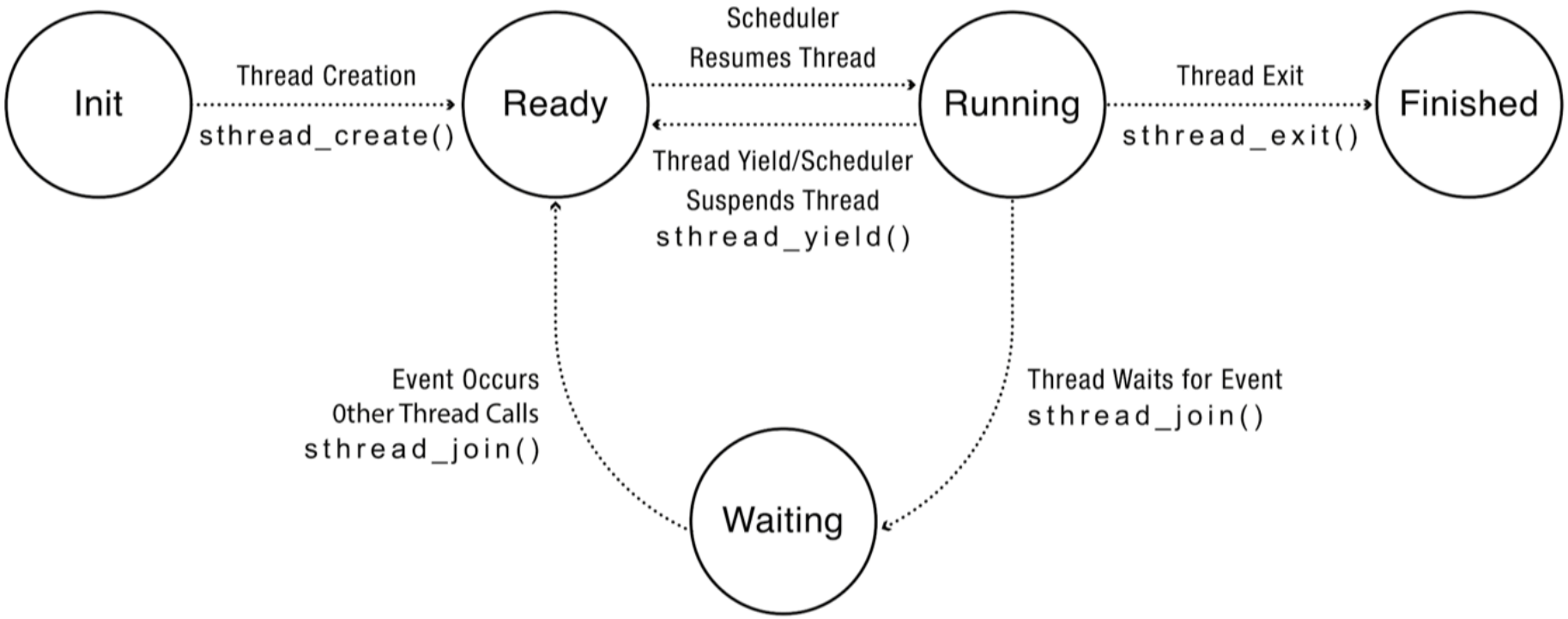
- registers / saved context
- stack
- TCB (thread control block)
- ready/wait queues

- Thread Control Block (TCB)
  - State of the computation
  - Management metadata
- Per-thread Computation state
  - Stack
  - Processor Registers
- Per-thread Metadata
  - Thread ID
  - Scheduling Priority
  - Thread Status



- Share state
  - Program code
  - Global variables
  - Heap
- Other pre-thread state
  - Scope spans like global variables
  - Each thread has its own copy
  - **Errno**
    - Thread-local variable
  - **Heap internals**
    - Subdivide the heap
    - Improve cache performance

Thread Lifecycle



Kernel threads vs user-level threads

- **Kernel threads**: thread abstraction managed by OS kernel
- **User-level threads**: implemented in a library without syscalls
  - fast switching
  - but without kernel help, blocking syscalls block the entire process

Hybrid model: user-level scheduling on top of a pool of kernel threads (e.g., Go runtime).

Thread Context Switch

A thread can switch between RUNNING and READY:

- **Voluntary**: `yield`, waiting in `join`, blocking on a lock
- **Involuntary**: preemption via timer interrupt, higher-priority thread becomes runnable

Context switch details (two important facts)

1. A context switch starts executing in the **old** thread, and returns in the **new** thread
2. The return PC might not be “the yield() call”

- a thread could have been blocked in `join` and is now waking up
- it resumes from wherever it was stopped

This is why shared invariants / atomicity is critical.

Involuntary Thread Switch

- a timer or I/O interrupt arrives
- interrupt handler decides to schedule a different task

Naïve implementation can save/restore context twice:

- once in interrupt entry/exit
- once in the context switch routine

Other concurrency abstractions (big picture)

Besides threads, common abstractions include:

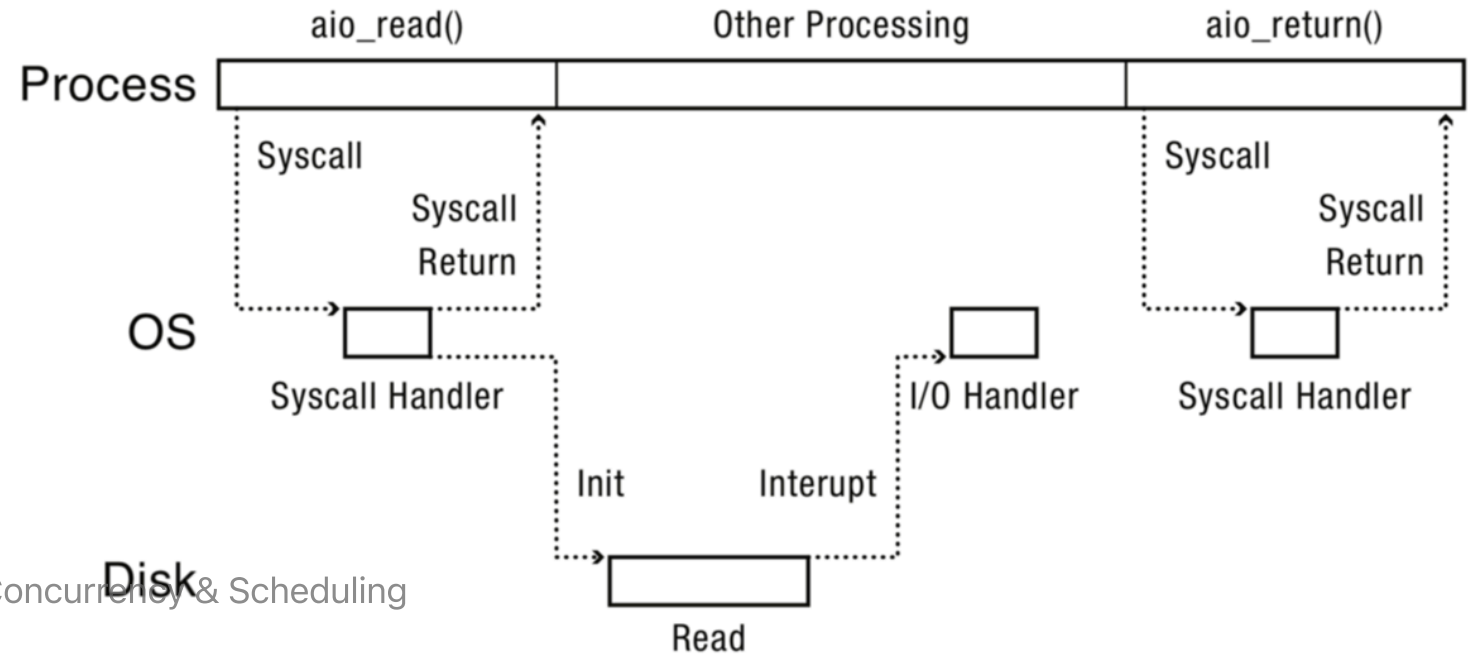
- asynchronous I/O + event loop (e.g., Node.js)
- data-parallel models (SIMD, OpenMP)

You'll see these again later in the course (containers, network dataplane, runtime systems).

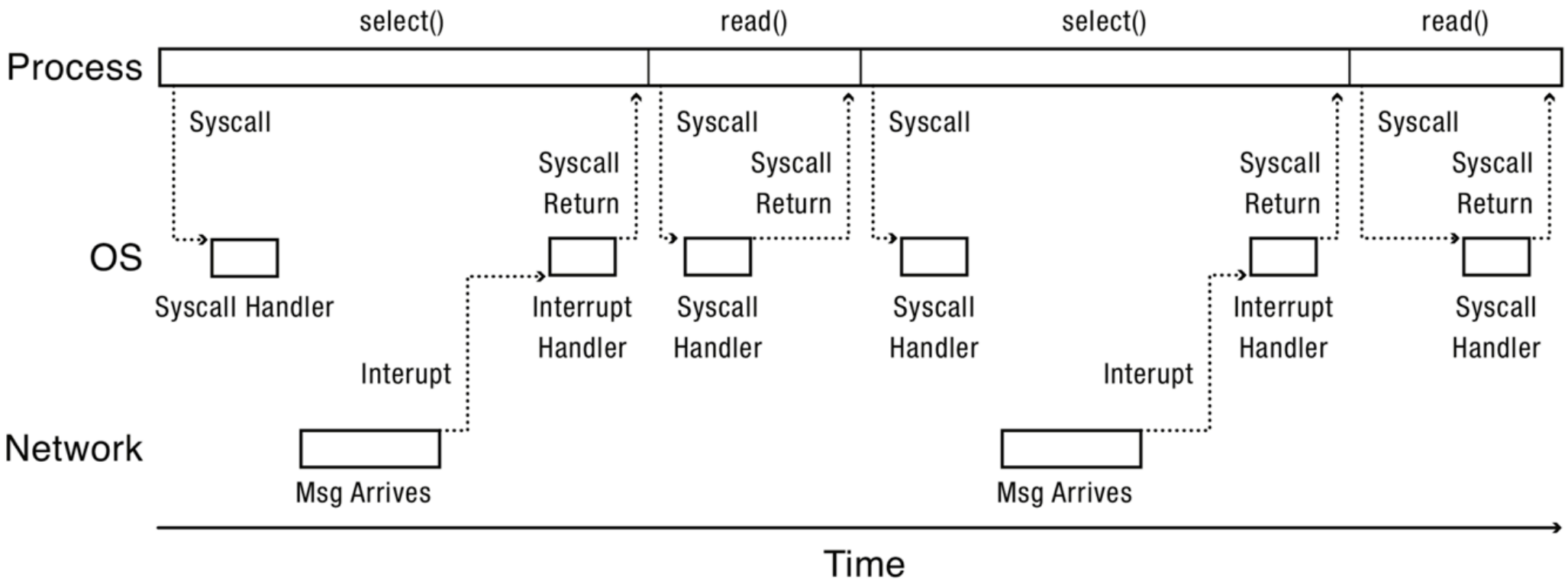
Async I/O (concept)

- issue multiple I/O requests concurrently
- call returns immediately
- completion delivered later via:
  - signals/callbacks
  - shared queue
  - separate syscall to fetch results

Linux AIO example diagram:



I/O multiplexing: select/poll/epoll



Part 1: Scheduler Fundamentals

From toy policies to practical latency experiments

What is a scheduler trying to optimize?

Common (and conflicting) goals:

- **response time** (latency to first run)
- **turnaround time** (time to completion)
- **fairness** (no starvation)
- **overhead** (minimize context switches / decision cost)
- **throughput** (work done per unit time)

Schedulers are fundamentally **policy + mechanism**.

Uniprocessor Scheduling Policies

Classic policies:

- First-in-first-out (FIFO / FCFS)
- Shortest Job First (SJF)
- Round-Robin (RR)
- Case study: Multi-level Feedback Queue (MLFQ)

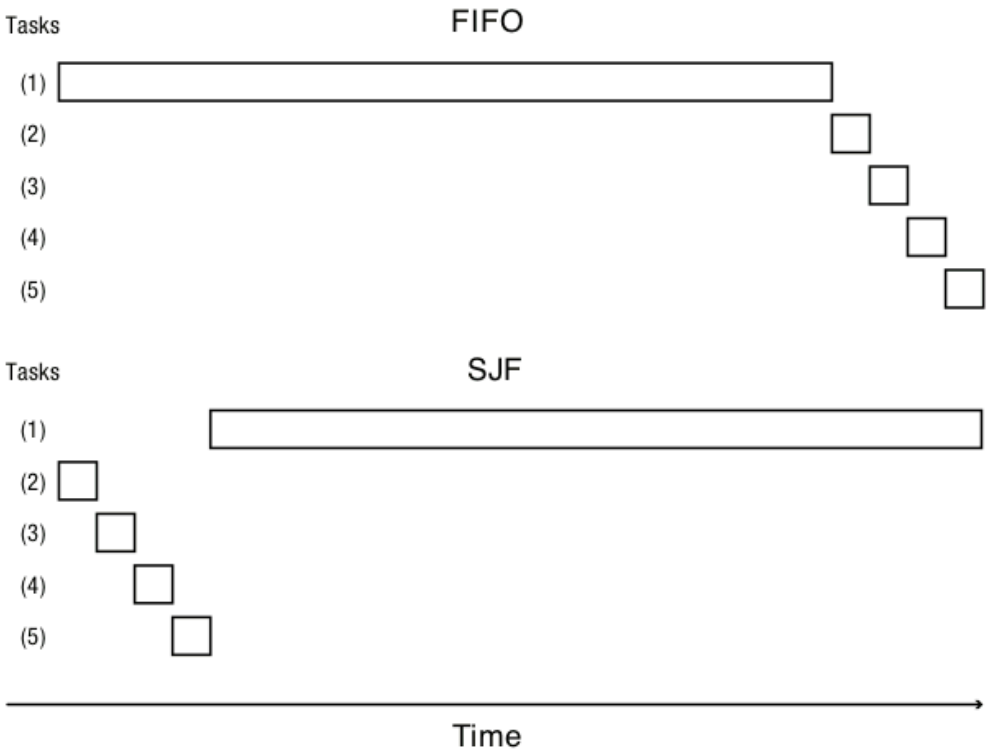
These are the right mental models even if Linux is more complex.

FIFO / FCFS

- run each task in the order it arrives

Clear problem:

- a tiny job can get stuck behind a huge job → bad average response time



When FIFO is OK

Even “simple” policies can fit:

- when requests are small and similar in size
- when minimizing scheduling overhead matters

Example: some caching servers can approximate FIFO under certain workloads.

Shortest Job First (SJF)

- schedule the shortest job first
- optimal for average response time **if** you know job length

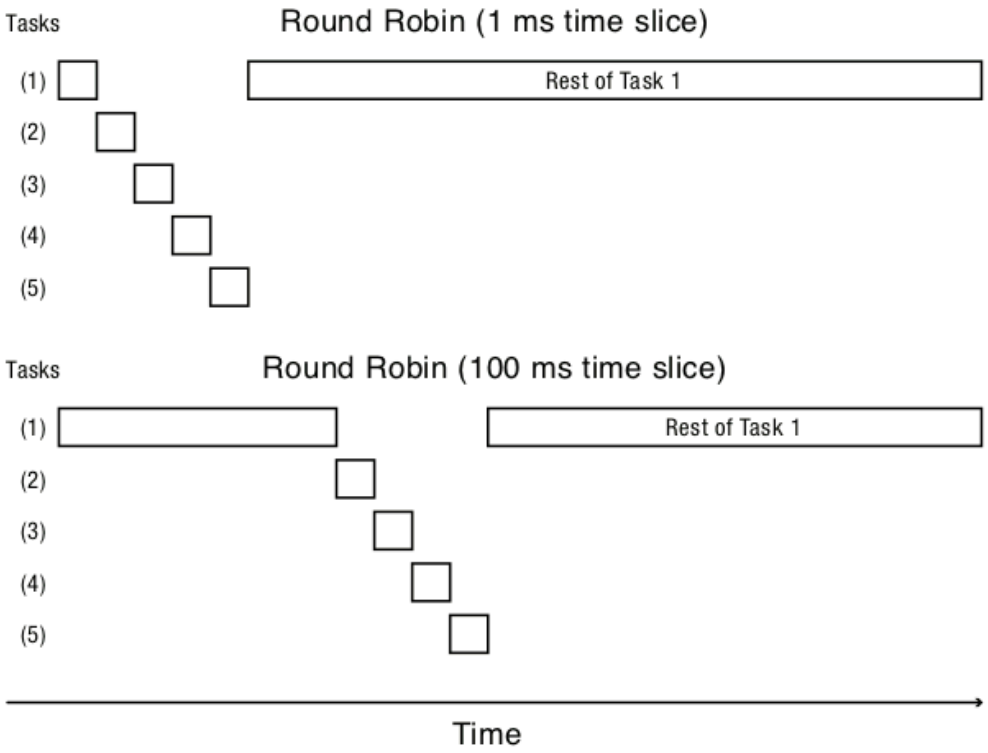
Downsides:

- can starve long jobs
- can increase variance in response time

Round-Robin (RR)

Addresses starvation by time slicing:

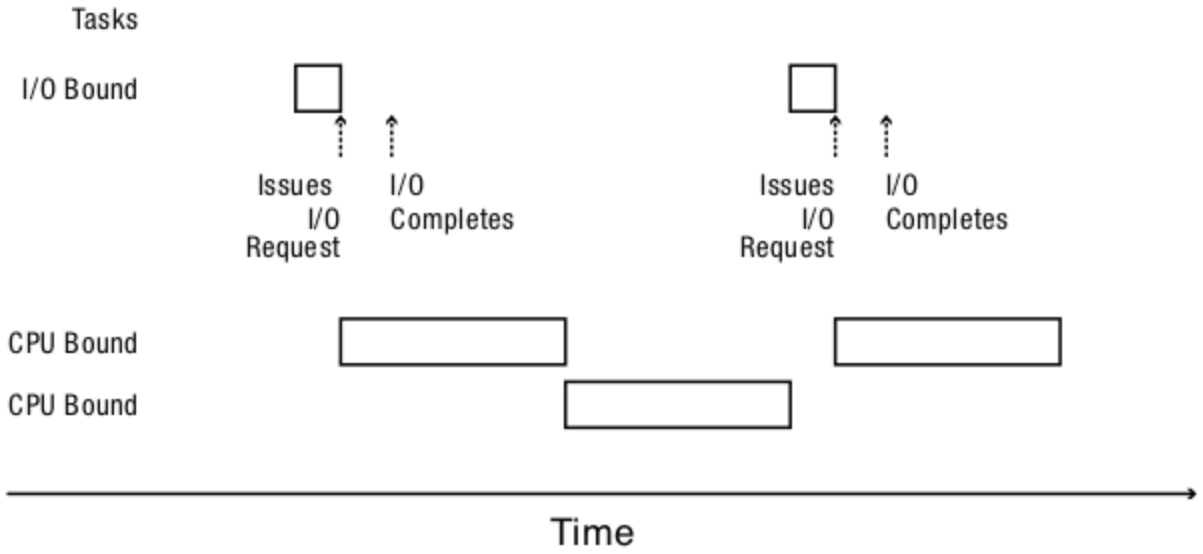
- each runnable task runs for a **time quantum**
- at quantum end, preempt and put it back to ready queue



RR can be bad for mixed workloads

I/O-bound tasks need short bursts of CPU to issue the next I/O.

With a large quantum, an I/O-bound task may wait behind CPU-bound tasks for a long time.



Case Study: Multi-Level Feedback Queue (MLFQ)

Most commercial OS schedulers incorporate MLFQ ideas:

- responsiveness (like SJF)
- low overhead (avoid too many preemptions)
- starvation freedom (like RR)
- fairness / background tasks

MLFQ is an extension of RR

- multiple RR queues with different priority levels + quanta
- higher priority preempts lower priority
- tasks move between levels based on behavior

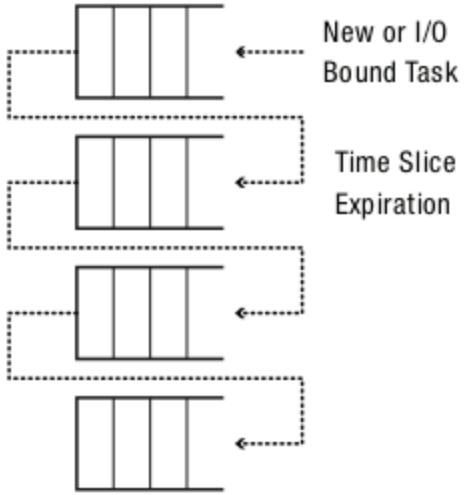
Rules-of-thumb:

- new task enters at high priority
- uses up quantum → drops a level
- yields/blocks quickly → stays high (interactive/I/O-bound)

MLFQ (picture)

Priority	Time Slice (ms)
1	10
2	20
3	40
4	80

Round Robin Queues



Per-CPU run queues (scales to many cores)

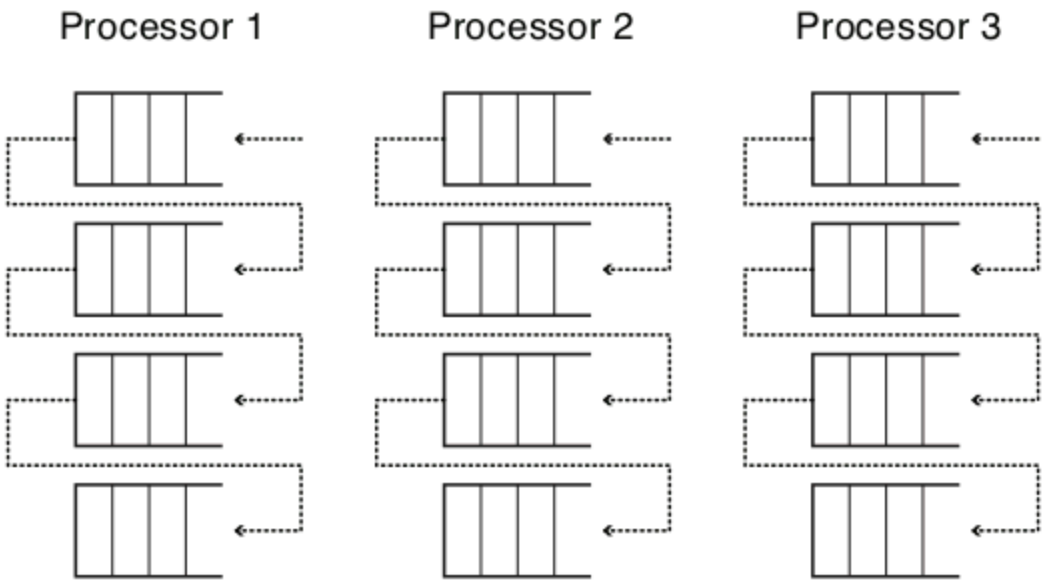
Modern kernels maintain per-CPU runnable queues:

- each CPU has runnable entities
- load balancing may migrate tasks
- CPU affinity constrains where tasks may run

Tail latency can improve with isolation (pinning) and worsen with migrations.

Multi-processor scheduling: per-processor queues

A typical approach: keep a separate runqueue per CPU.



Properties:

- affinity scheduling improves cache reuse
- load balancing only when queue length differences persist

Practical knobs we will use (Lab 3)

To create/mitigate contention and see tail latency change, we need a few **control knobs**:

- **Background load**: create runnable-queue pressure
- **nice**: bias CPU share between “important” vs “background” work
- **CPU affinity**: avoid/force contention on a core; reduce migrations

We will focus on *observable effects* first; Linux’s detailed policy (CFS, weights) is the focus of next week.

Nice values (practical effect)

nice	Meaning	Typical use
-20	highest priority (needs privilege)	critical system tasks
0	default	normal work
+19	lowest priority	background batch jobs

In this week's lab, we'll use **+19** for background CPU hogs so the latency-sensitive task wins more often.

Bridge to Week 4: make scheduler decisions visible

Next week we will connect the experiment to concrete Linux events:

- tracepoints: `sched:sched_wakeup`, `sched:sched_switch`
- metrics: per-task runtime, scheduling latency distributions, migrations
- tool: eBPF (low-overhead continuous observation)
- note: on modern kernels (Linux ≥ 6.6; our class uses ~6.14), the CFS class uses **EEVDF** internally

Week 3: *I can reproduce the tail.*
Week 4: *I can explain the tail from kernel events.*

Part 2: Modern Context

Tail latency, runtimes, and how scheduling shows up in production

Thread States

Typical high-level states:

- **Running**: executing on a CPU
- **Runnable**: ready, waiting for CPU time (in run queue)
- **Sleeping / Blocked**: waiting on I/O, timer, futex, etc.

Scheduling latency is mainly time spent in **Runnable** state.

Where does scheduling happen?

A simplified timeline:

1. A thread becomes runnable (wakeup)
2. It is placed in a run queue (enqueue)
3. Scheduler chooses next runnable thread (pick-next)
4. Kernel performs a context switch

We will map each of these to concrete Linux events and measurements.

What is a context switch?

A context switch typically includes:

- save registers of the outgoing thread
- switch kernel stack
- update scheduler bookkeeping
- switch memory context (sometimes cheap, sometimes expensive)
- restore registers of the incoming thread

Context switches are not free; *too many* can destroy tail latency.

Atomicity & critical sections

When scheduler structures change (run queues, wait queues):

- operations must be atomic
- otherwise preemption/interrupts can observe inconsistent state

Kernel approach:

- spinlocks + disabling preemption/interrupts in critical paths

Scheduling latency

Scheduling latency = time from “thread becomes runnable” to “thread starts running”.

Linux tracing approximation:

- start: `sched:sched_wakeup` (or wakeup from sleep)
- end: `sched:sched_switch` where that task becomes `next`

This is OS-level queueing delay.

Why tail latency is a scheduling problem

Even if average CPU usage looks fine, p99 can explode because:

- runnable queue builds up briefly (microbursts)
- lock contention causes threads to sleep/wake frequently
- CPU migrations blow away cache locality
- background work steals time slices

Tail latency is usually about *queueing somewhere*.

User-level runtimes still matter

Many systems implement scheduling above the kernel:

- Go goroutines (G/M/P model)
- JVM (executors / virtual threads)
- async runtimes (tokio, libuv)

But our focus this week: **measuring and explaining kernel scheduling latency under real contention**.

How to measure scheduling effects

If your environment allows it:

- `perf sched record` + `perf sched timehist/latency`
- `perf stat -e context-switches,cpu-migrations`
- `/proc/schedstat`, `/proc/<pid>/sched`

If perf PMU events are unavailable in VMs (common in VirtualBox/VMware), you can still:

- measure user-visible wakeup lateness (our lab)
- use `/proc` scheduler stats as supporting evidence

A minimal “scheduling latency experiment”

We run a periodic latency-sensitive loop:

1. sleep until an absolute timestamp (1ms period)
2. record how late we actually wake up
3. repeat thousands of times

Under CPU contention, wakeups get delayed → p99 increases.

Part 3: Case Study

“CPU usage is fine, but p99 is terrible”

Incident

A service reports:

- CPU utilization ~60%
- Average latency stable
- **p99 latency spikes** during busy hours

Question: how can p99 explode if CPU isn’t at 100%?

Hypothesis

Even at 60% average, the system can have short windows where:

- many threads become runnable at once
- a few get scheduled late

This creates “micro-queueing” on the run queue.

Tail latency is dominated by these worst moments.

Evidence chain you should look for

Two independent signals:

1. **Tail of wakeup latency** (p99 of “late by X us”)
2. Scheduler pressure indicators:
    - high `context-switches`
    - high `cpu-migrations`
    - (optional) `perf sched latency` shows run-queue delays

How to fix?

1. Reduce background CPU work (or nice it down)
2. Pin latency-sensitive work to a core (affinity)
3. Reduce lock contention (less sleep/wakeup churn)
4. Use cgroups/quotas to isolate tenants

In the lab you will validate at least one fix with before/after p99.

Part 4: Lab Preview

Lab 3: Scheduling Latency Under CPU Contention

Lab 3 Goal

Create controlled CPU contention and show:

1. Wakeup latency distribution has a long tail (p50/p90/p99)
2. Under contention, p99 increases significantly
3. A mitigation (nice / CPU affinity / cgroup) improves the tail

Workload 1: latency probe (prints one sample per iteration)

```
make -C week3/lab3_sched_latency
./week3/lab3_sched_latency/wakeup_lat --iters 20000 --period-us 1000 --cpu 0 > baseline.log
```

Output format:

```
iter=123 latency_us=847
```

Workload 2: controllable CPU load generator

```
# In another terminal
./week3/lab3_sched_latency/cpu_hog --threads 4 --cpu 0
```

Then rerun the probe and compute percentiles.

Evidence you must show

1. Tail percentiles from the samples (baseline vs contended)
2. At least one supporting OS signal:

- `perf stat -e context-switches,cpu-migrations ...` (if allowed)
- `/proc/<pid>/sched` or `/proc/schedstat` snapshot

One required mitigation

Pick one and verify improvement:

- **Nice** background hogs: `nice -n 19 ./cpu_hog ...`
- **Affinity**: pin probe to an isolated CPU vs shared CPU
- **(Optional advanced)** cgroup CPU control (`cpu.max` or `cpu.weight`)

Deliverables

Submit:

1. `report.md` (use template)
2. `baseline.csv` and `contended.csv` (or a combined CSV)
3. One paragraph: “mechanism chain” from contention → runnable queue → p99

References

- OSTEP: Scheduling + Concurrency
  - Scheduling intro (Ch. 7), MLFQ (Ch. 8)
  - Concurrency intro (Ch. 26)
- `man 1 nice`, `man 1 taskset`
- `perf sched` docs: https://perf.wiki.kernel.org/

(Backup) Beyond a single CPU / single machine

If we have time, these examples connect scheduling ideas to clusters:

- Gang scheduling (parallel jobs)
- Batch scheduling and backfilling (HPC)

Gang Scheduling (concept)

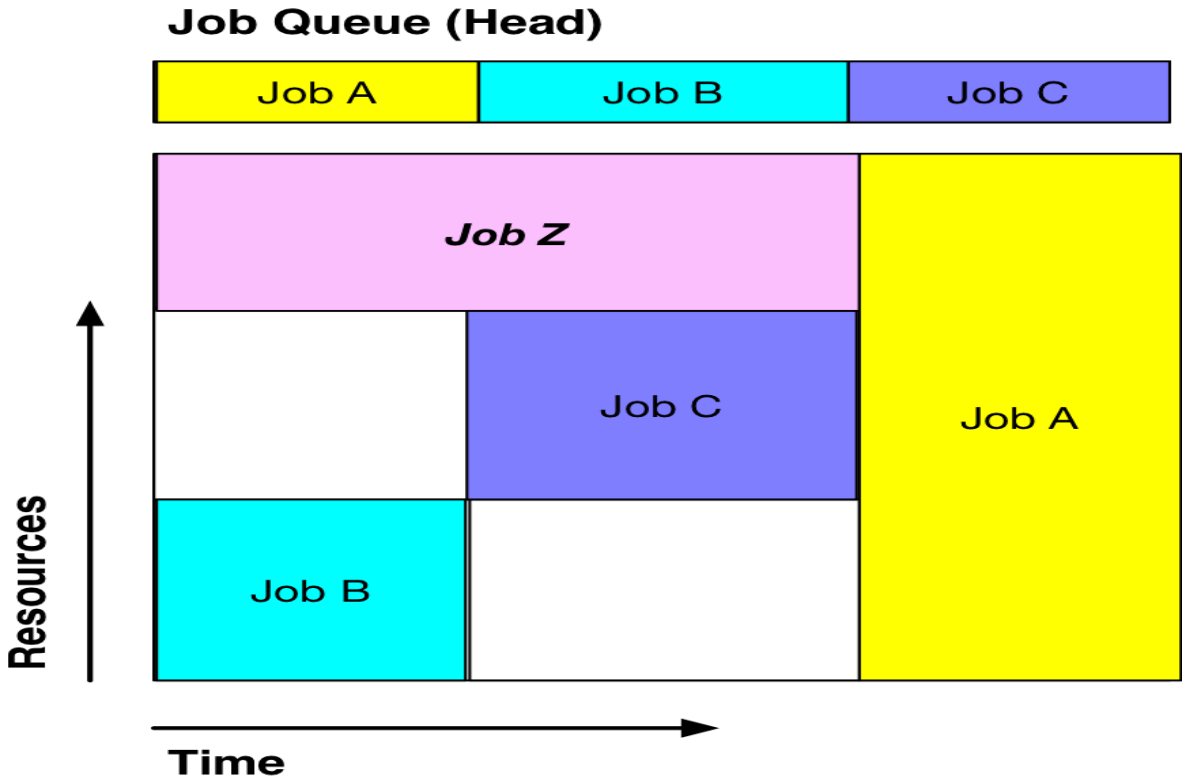
- schedule all tasks/threads of a parallel program together
- if OS switches to a different application, it preempts the whole gang if needed

Useful in some parallel workloads; can be inefficient for general multiplexing.

Backfilling (HPC batch scheduling)

If the highest-priority job can’t run due to insufficient resources:

- scheduler may run smaller jobs as long as it doesn’t delay the reserved start time of the big job



Questions?

Then we start Lab 3.