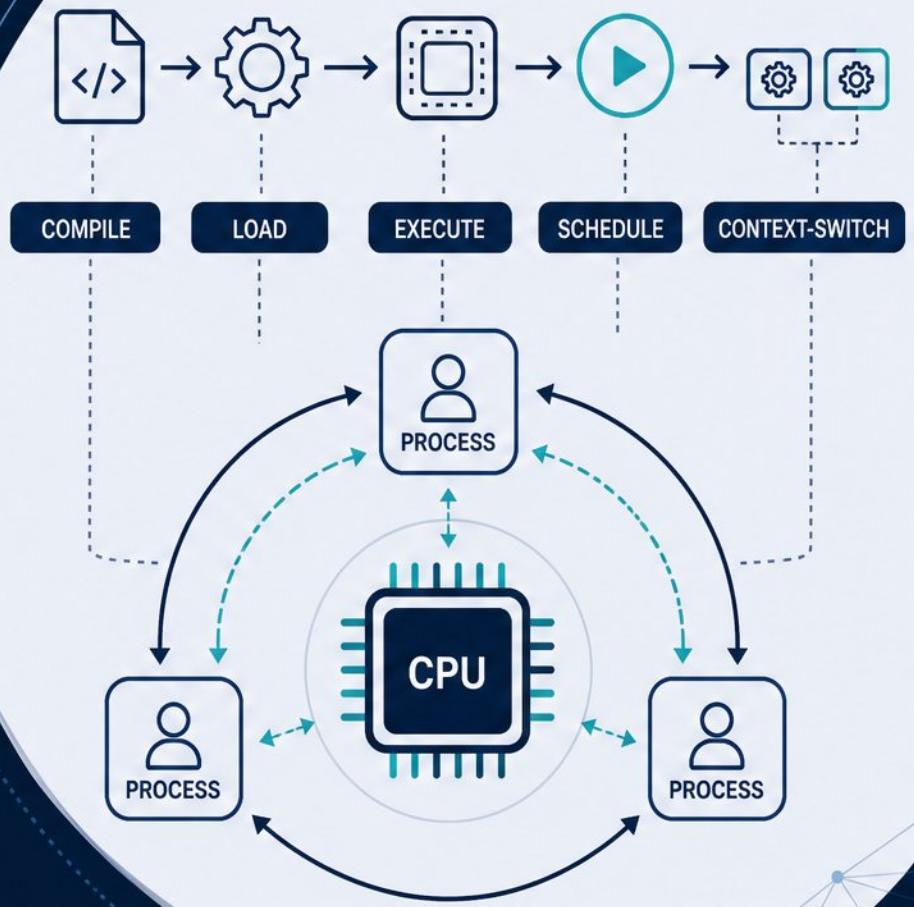
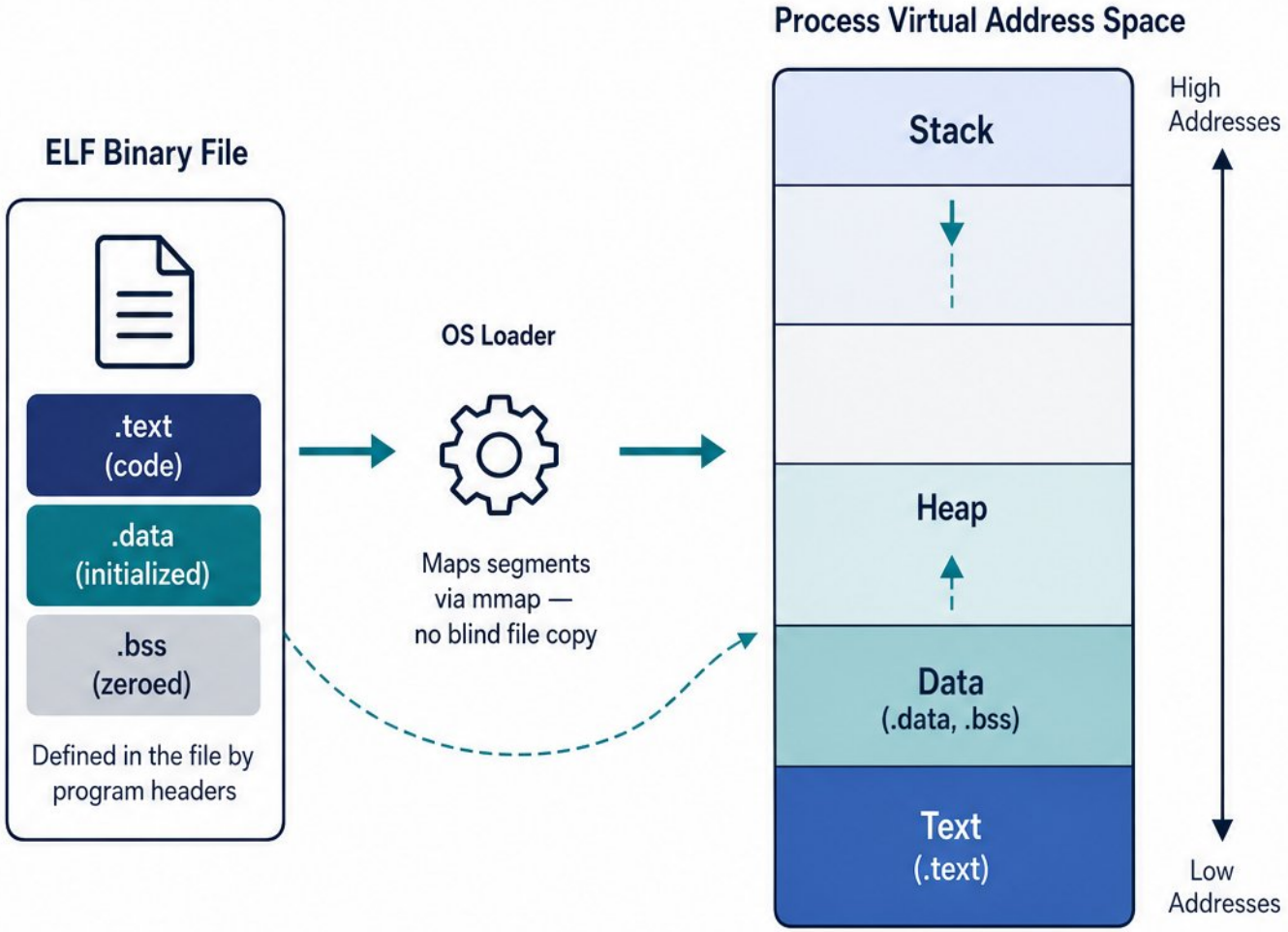
How Operating Systems Run Programs: From Code to Concurrent Execution

- Trace the journey from a program file on disk to multiple running processes
- Explore how the OS creates, schedules, and switches between processes
- Overview of the lifecycle: compile, load, execute, schedule, context-switch
- Process vs. program: CPU virtualization, scheduling, context switching, memory isolation

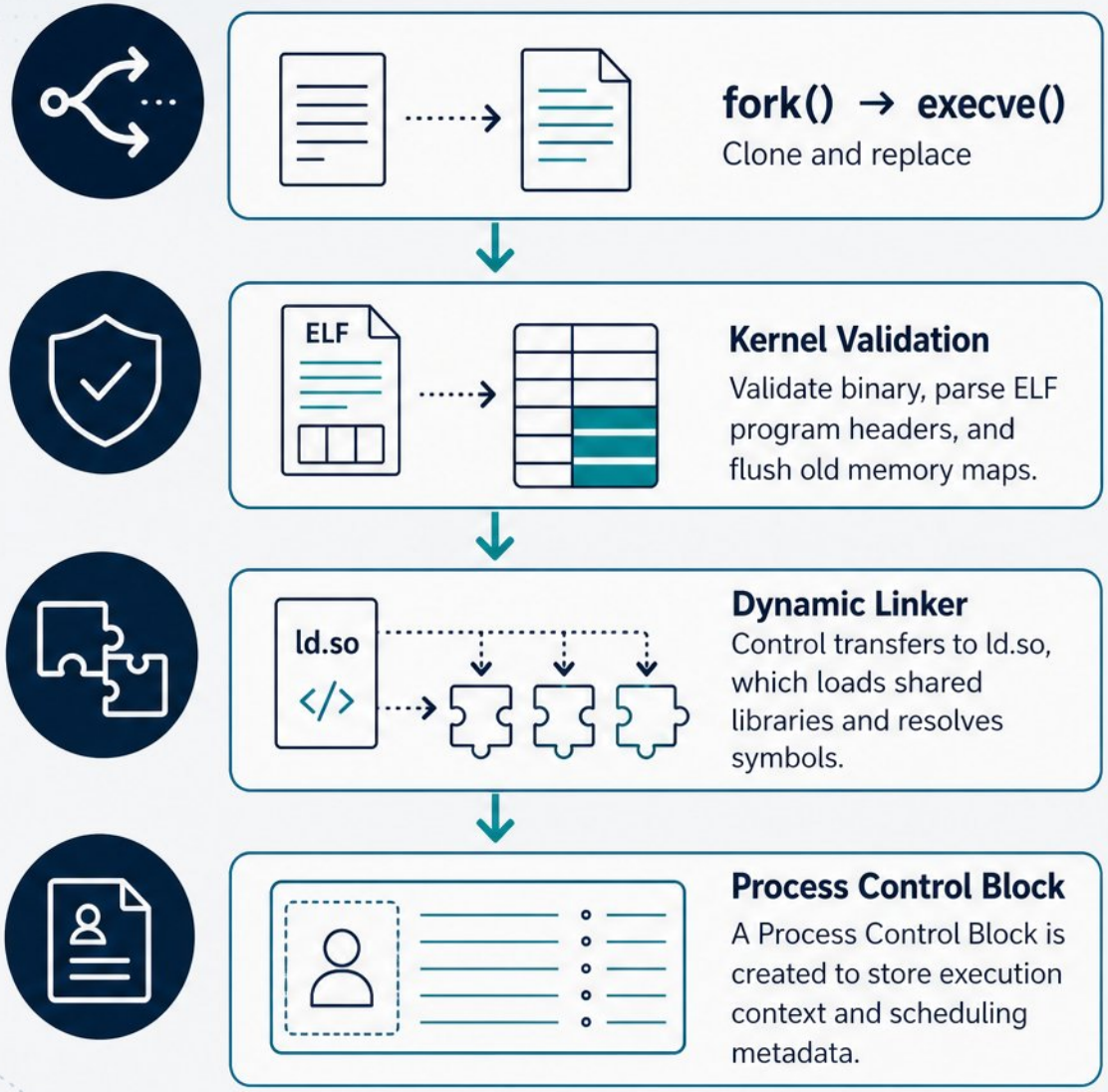


Deconstructing a Program: From Binary File to Address Space

- ELF executable segments: .text (code), .data (initialized), .bss (zeroed)
- Program headers, not section headers, tell the kernel how to map memory
- The OS loader maps segments via mmap — no blind file copy
- Virtual memory + MMU give each process a private, isolated address space
- Resulting layout: Text, Data, Heap → ← Stack



The Spark of Life: How the OS Creates a Process



Process Control Block (PCB)

Execution Context
- Program Counter
- CPU Registers
- Stack Pointer
- Process State

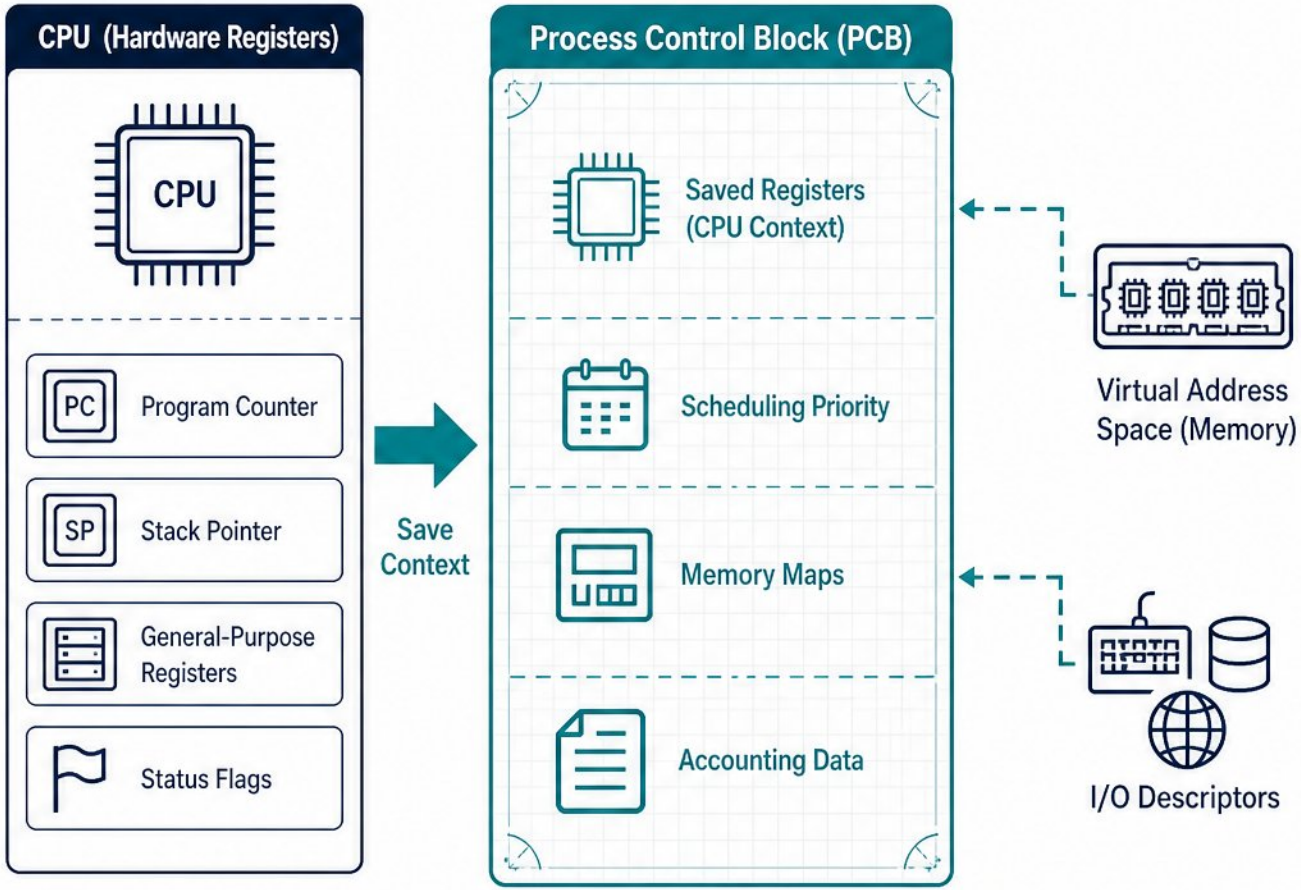
Scheduling Metadata
- Scheduling Information
- Priority
- Queue Links
- Time Accounting

Memory Management
- Memory Map Pointer
- Page Tables
- Address Space Info

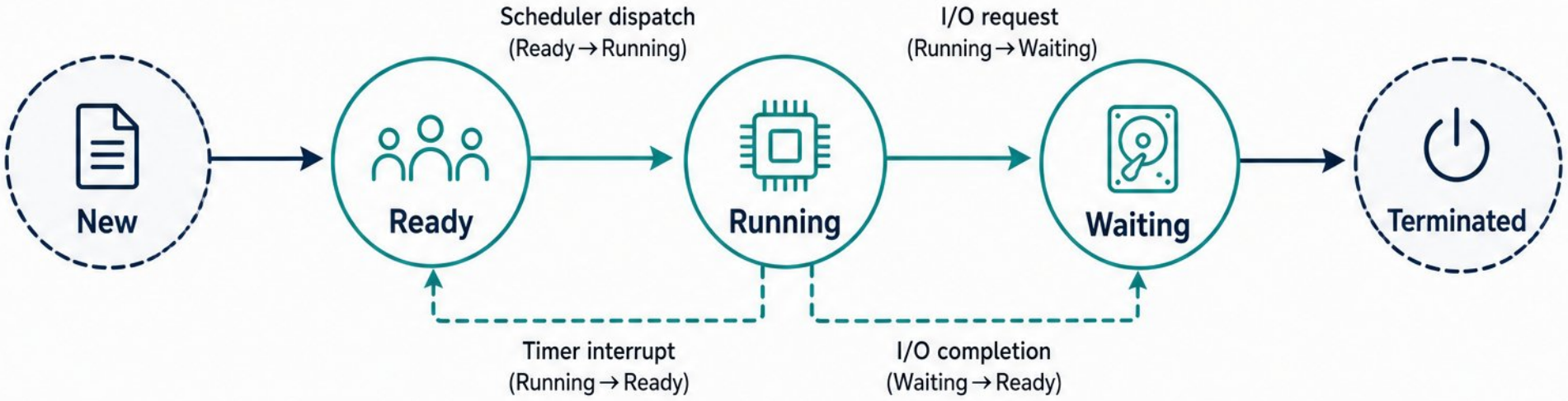
I/O and Resources
- Open Files
- File Descriptors
- Other Resources

Inside a Process: The CPU, Memory, and State Footprint

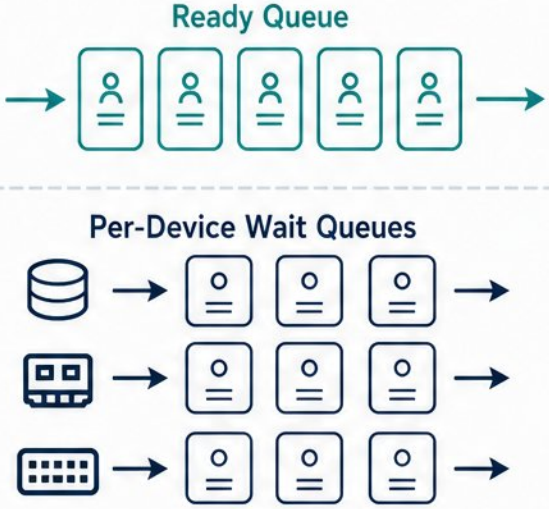
- Every process owns a virtual address space, CPU context, and I/O descriptors
- Hardware execution context: Program Counter, Stack Pointer, general-purpose registers, status flags
- The Process Control Block (PCB) stores saved registers, scheduling priority, memory maps, and accounting data
- Saving and restoring context atomically is critical for transparent process switching



The Rules of the Game: Process States and Lifecycle Transitions

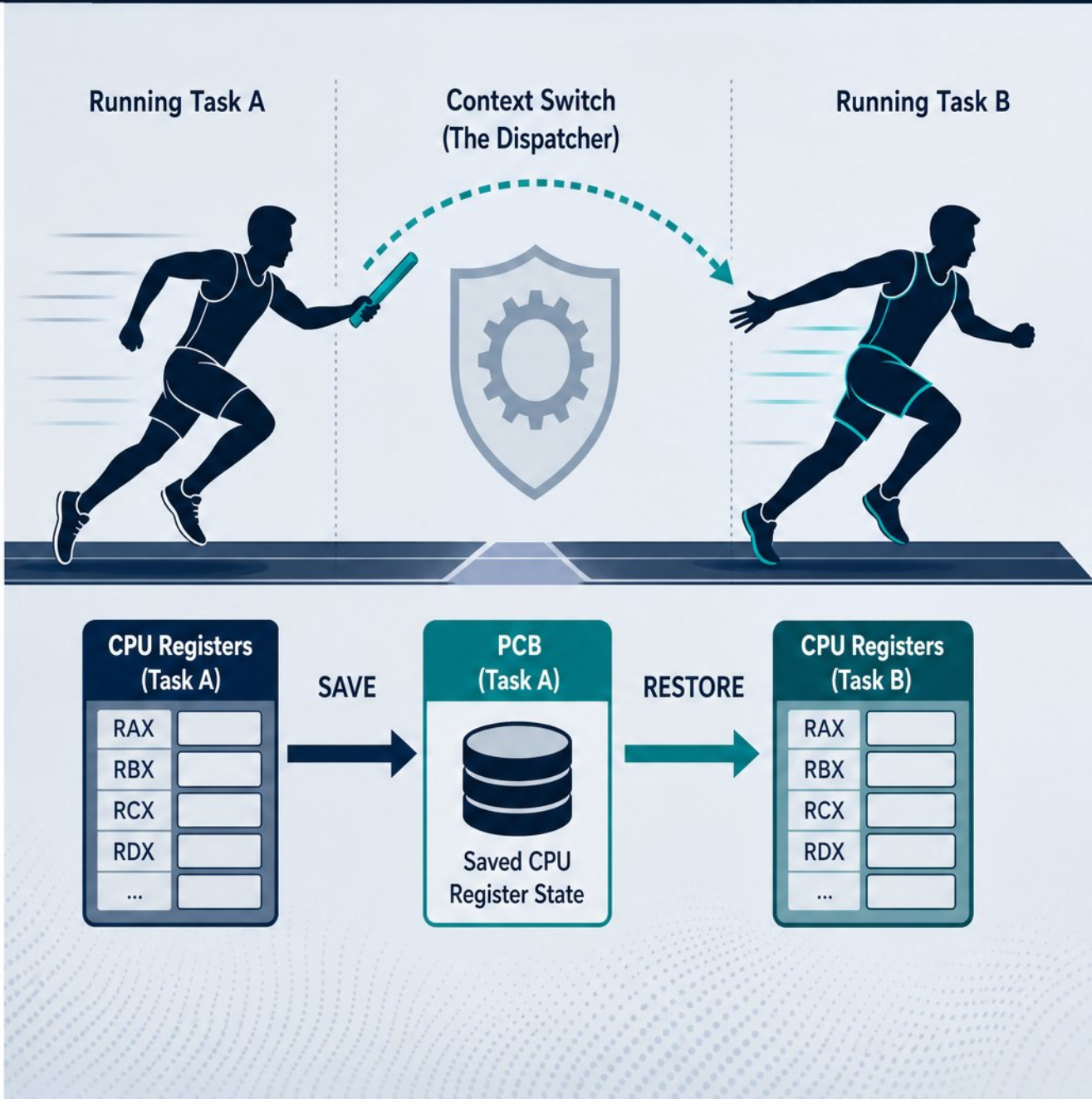


- Five-state model: New → Ready → Running → Waiting → Terminated
- Scheduler dispatch (Ready → Running), timer interrupt (Running → Ready)
- I/O request (Running → Waiting), I/O completion (Waiting → Ready)
- I/O-bound processes block frequently; CPU-bound use full time slices
- Ready queue and per-device wait queues organize PCBs by state



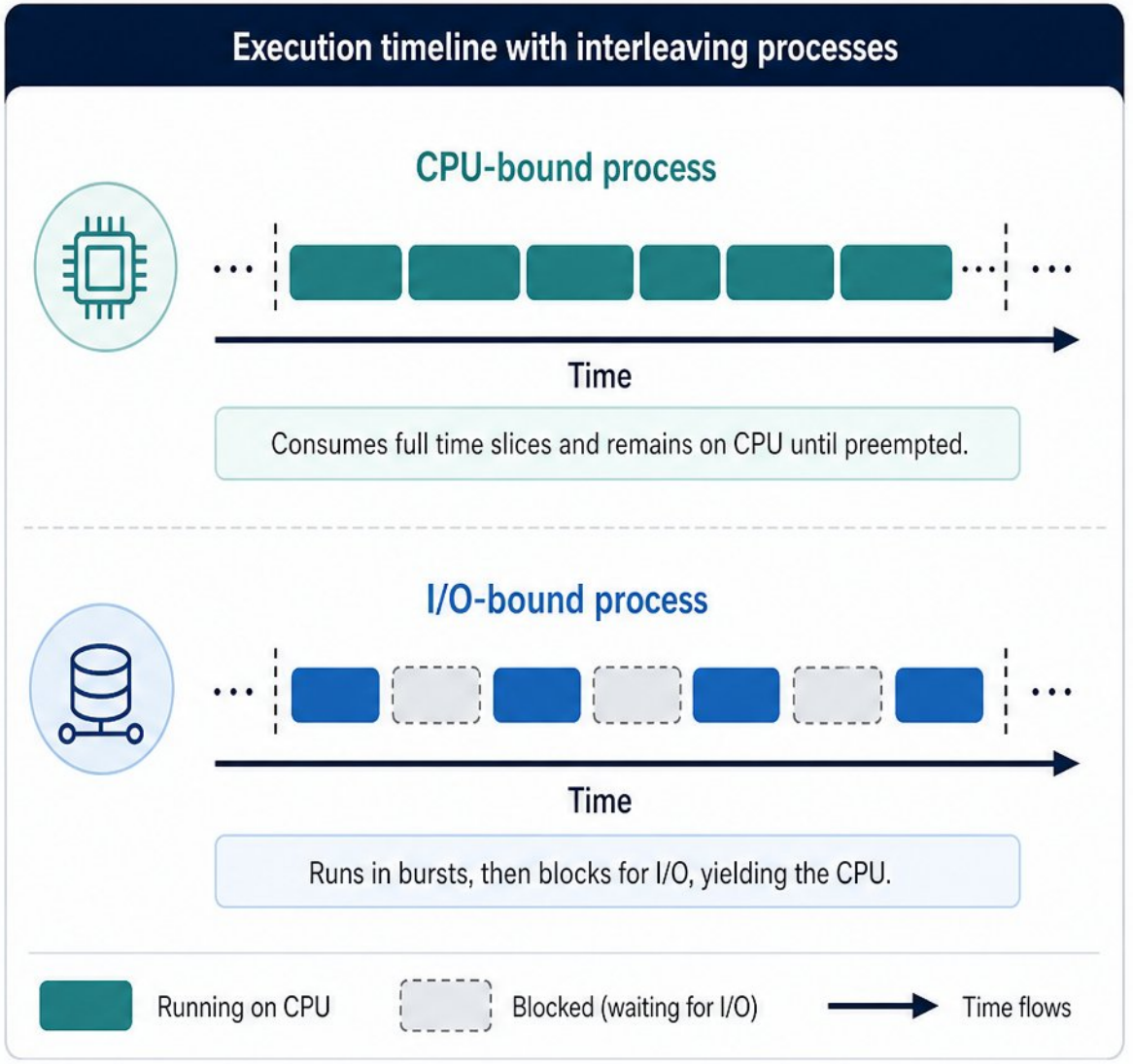
The Dispatcher: Context Switching Mechanics Step-by-Step

- Enter kernel via system call or interrupt, save CPU registers to PCB
- Voluntary switch: task blocks on I/O or lock. Involuntary: timer preempts
- Timer interrupt drives preemption, preventing CPU monopolization
- Indirect costs of cache/TLB perturbation dominate direct register save cost



Introduction to Scheduling: Goals, Metrics & Process Profiles

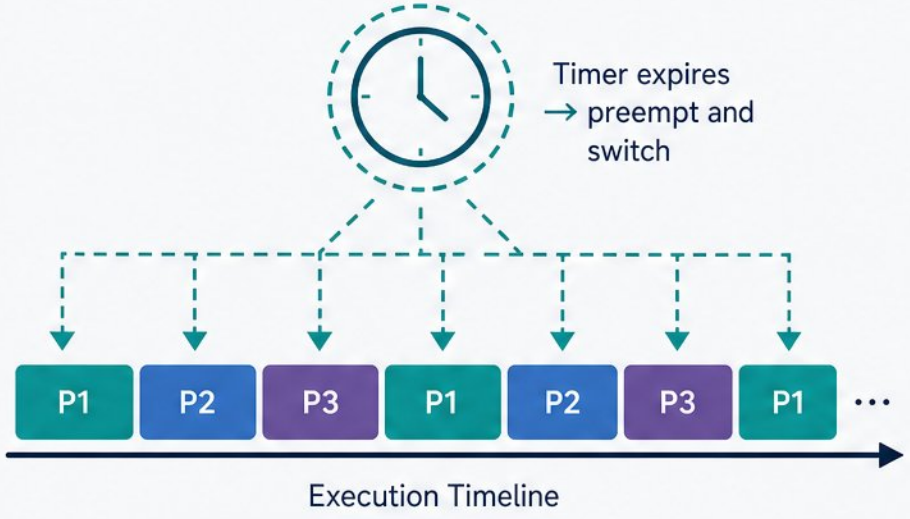
- **Goals:** fair CPU sharing, high throughput, low latency & response time
- Turnaround time = completion time – arrival time
- **Response time =** first schedule time – arrival time
- CPU-bound processes consume full time slices; I/O-bound processes block often
- **Core challenge:** optimize mixed workloads without knowing future behavior



Scheduling Algorithm Foundations: Round-Robin and Priority

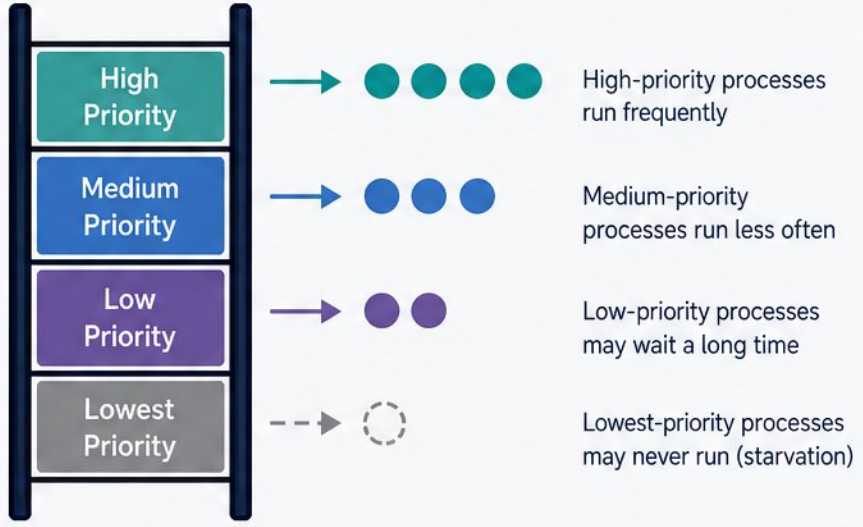
ROUND-ROBIN

Quantum Timer for Fair CPU Sharing



FIXED-PRIORITY SCHEDULING

Starvation Risk and Aging Solution



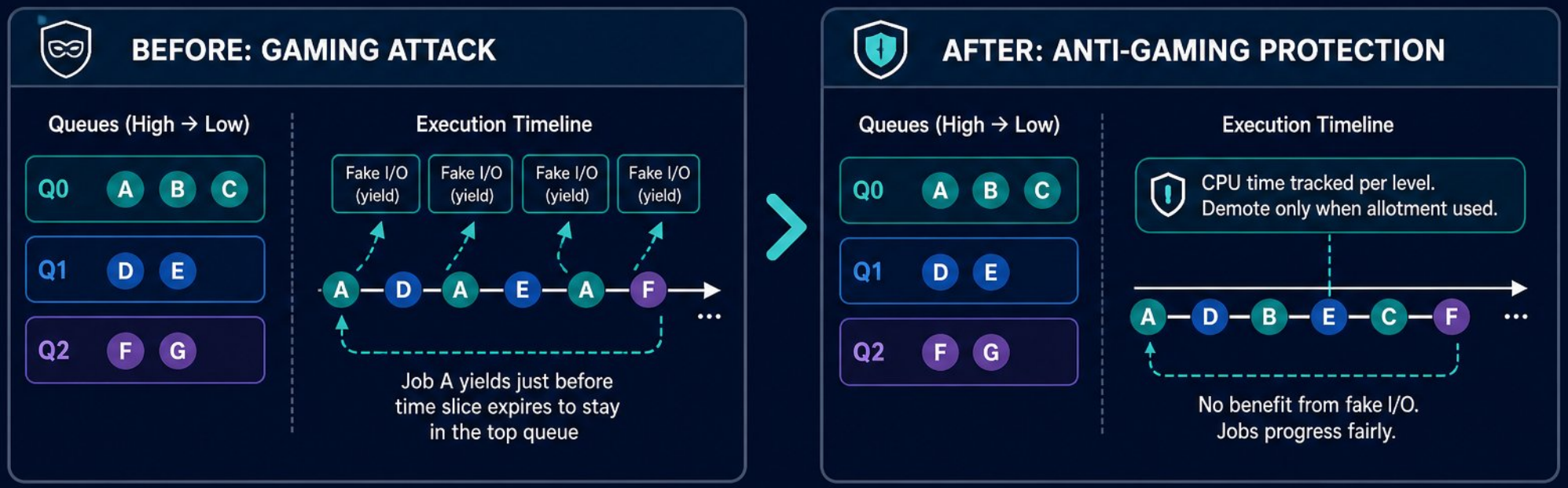
- Round-Robin: fixed time quantum with timer-based preemption for fair CPU sharing
- Quantum size trade-off: too small = excessive switches; too large = FCFS behavior
- Fixed-priority scheduling risks starvation—low-priority processes may never run
- Aging prevents starvation by dynamically boosting the priority of waiting processes

The MLFQ: An Adaptive, Learning Scheduler

- MLFQ learns from past process behavior to predict the future
- New processes start at the highest priority level
- Full time slice usage leads to demotion; I/O blocks lead to promotion
- Longer time slices for lower queues favor batch throughput
- Periodic priority boost prevents starvation of low-priority processes

MLFQ Refinements:
Priority Boost and Anti-Gaming

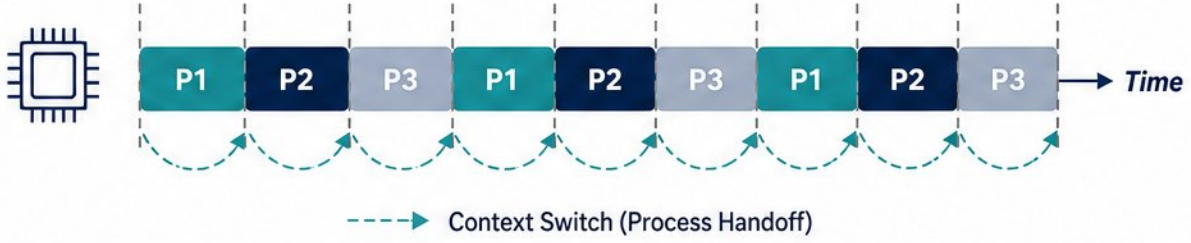
- Basic MLFQ vulnerabilities: starvation and scheduler gaming
- Priority boost periodically moves all jobs to the top queue
- Gaming: fake I/O just before time slice expires
- Anti-gaming rule: track total CPU time per level, demote only when allotment used



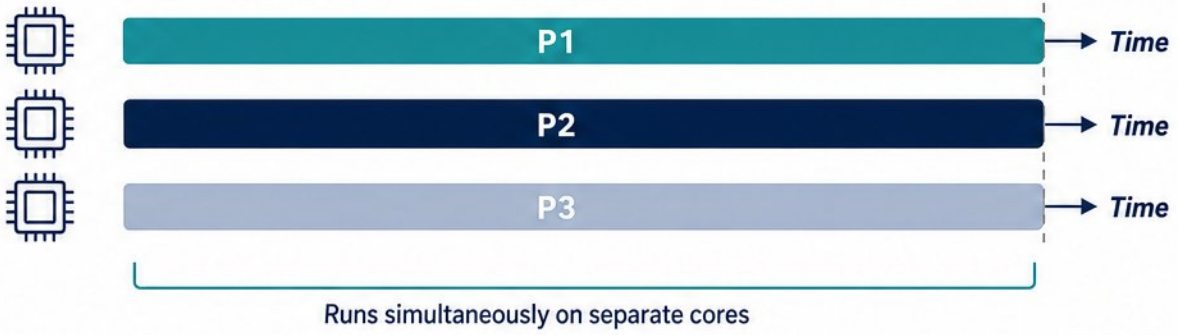
Concurrency in Action: Visualizing Context Switches Over Time

- Execution timelines show how multiple processes interleave on a single CPU, creating the illusion of parallelism
- Single-core interleaving alternates processes in time slices; multicore systems truly run them simultaneously
- A Round-Robin scheduler divides time into fixed quanta, switching to the next process when a quantum expires
- Scheduler decision points occur at every timer interrupt or I/O event, producing a preemptive pattern

SINGLE-CORE INTERLEAVED EXECUTION (TIME-SLICED)



MULTI-CORE PARALLEL EXECUTION (TRUE CONCURRENCY)



CONTEXT SWITCHING: THE RELAY RACE

Scheduling Artifacts: Starvation, Priority Inversion, and Latency

Starvation:

low-priority processes perpetually denied CPU by a flood of high-priority jobs.

Priority Inversion:

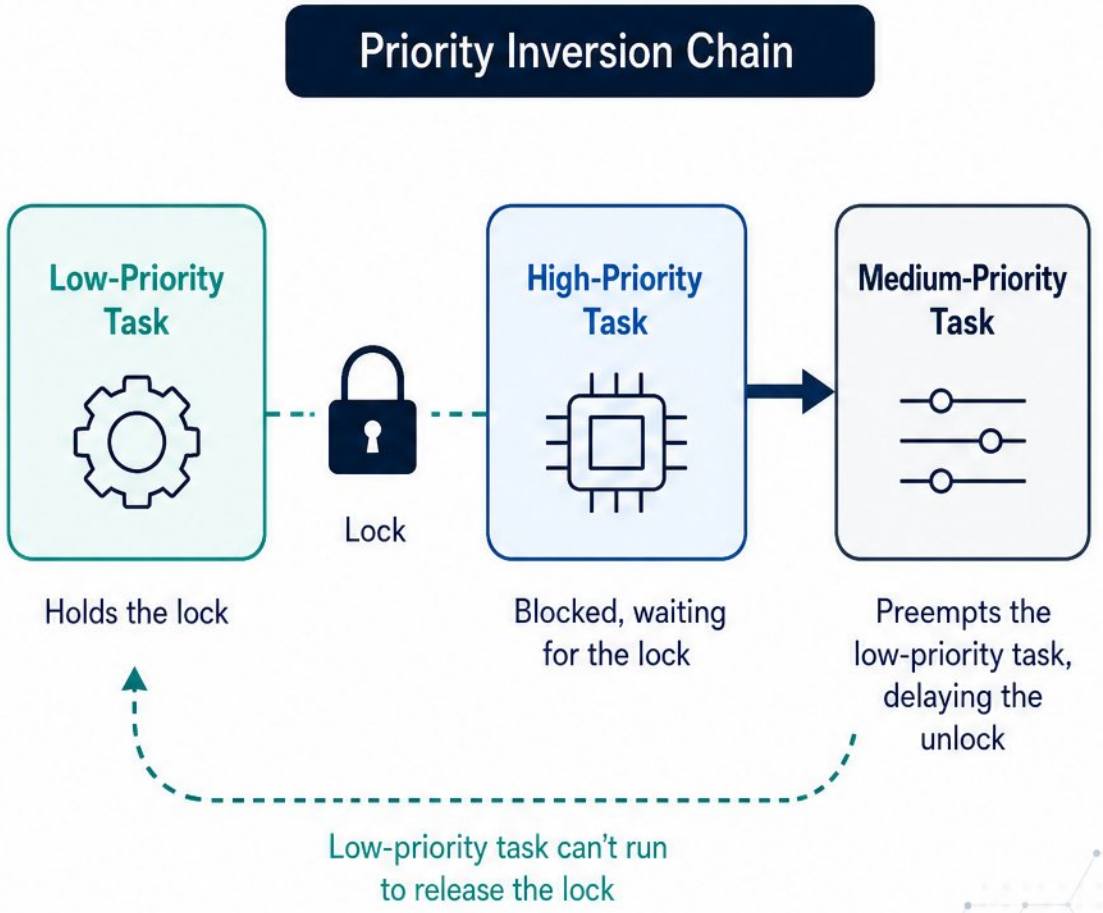
a high-priority task waits on a lock held by a low-priority task.

Real-world symptoms:

unresponsive UI, audio glitches, and unexplained high CPU usage.

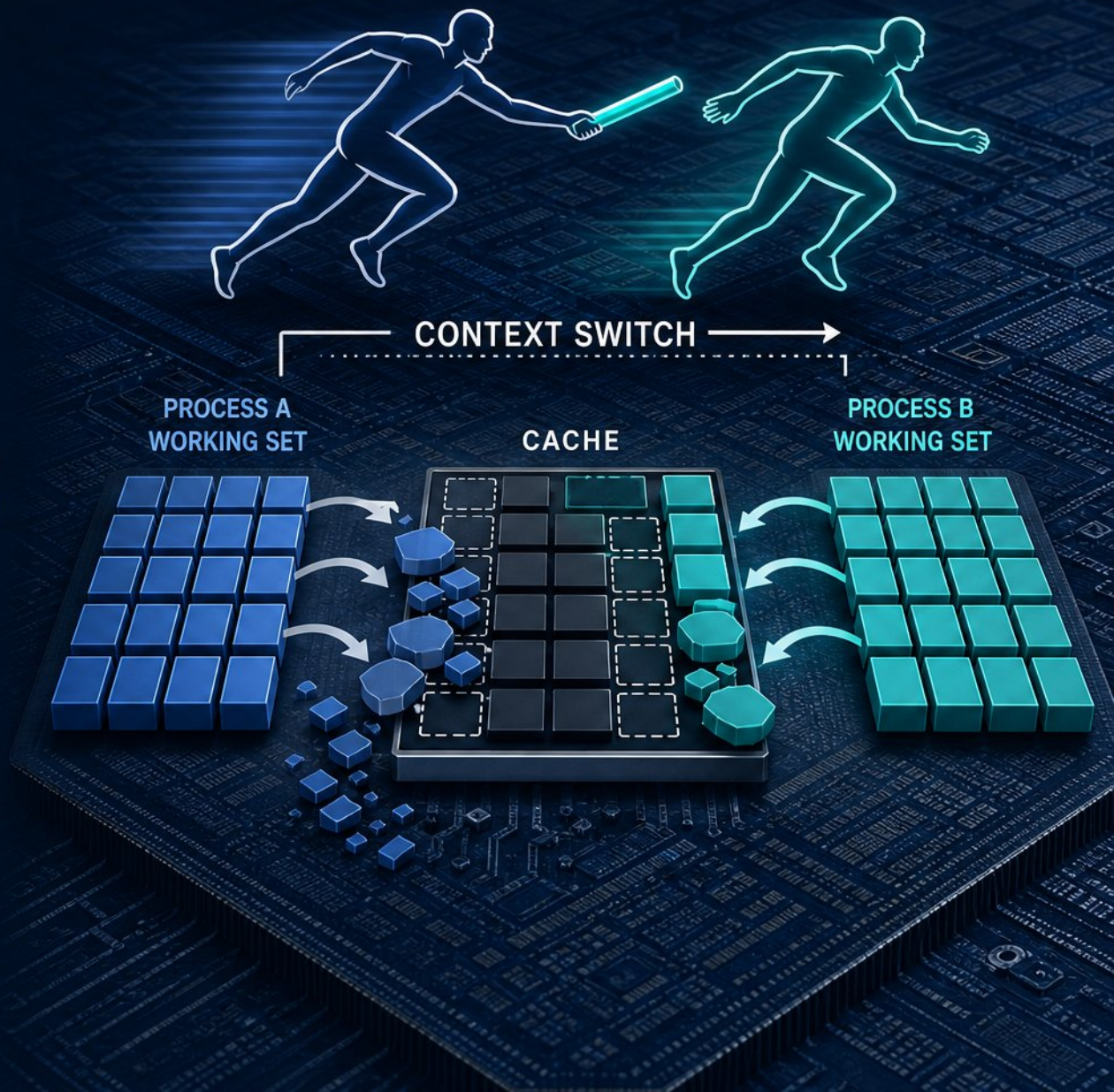
Mitigation:

priority inheritance temporarily boosts the lock-holder's priority to unlock faster.



The Hidden Cost: Cache Thrashing and Context-Switch Overhead

- After switching, a cache re-warming burst causes CPU cycles-per-instruction to spike
- Indirect overhead from cache/TLB misses often dwarfs the direct cost of saving registers
- A process near the cache size suffers most; its working set is fully displaced by the next process
- Larger associative caches paradoxically increase reordered misses: lines shift, not just replaced
- The penalty often reaches thousands of cycles, comparable to receiving a network packet



Practical Observation: Inspecting Processes and Scheduler Behavior

```
>_

$ ps -eo pid,ppid,stat,comm
  PID    PPID  STAT COMMAND
 1234       1   R   bash      → R: Running/ready
 1235    1234   S   vim       → S: Sleeping
 1236       2   D   worker    → D: Uninterruptible I/O

$ htop
  [ ||||||||||||||||||||      ] Tasks: ...
  [ ||||||||||||||||||||      ] Load average: ...
  [ ||||||||||||||||||||      ] Uptime: ...

$ vmstat 1
 r   b  swpd  free  buff cache  si  so  bi  bo  in  cs us sy id wa st
--- ---  ---   ---   ---   ---  ---  --- --- --- --- --- --- --- --- ---
```

r: run queue (vs. cores)

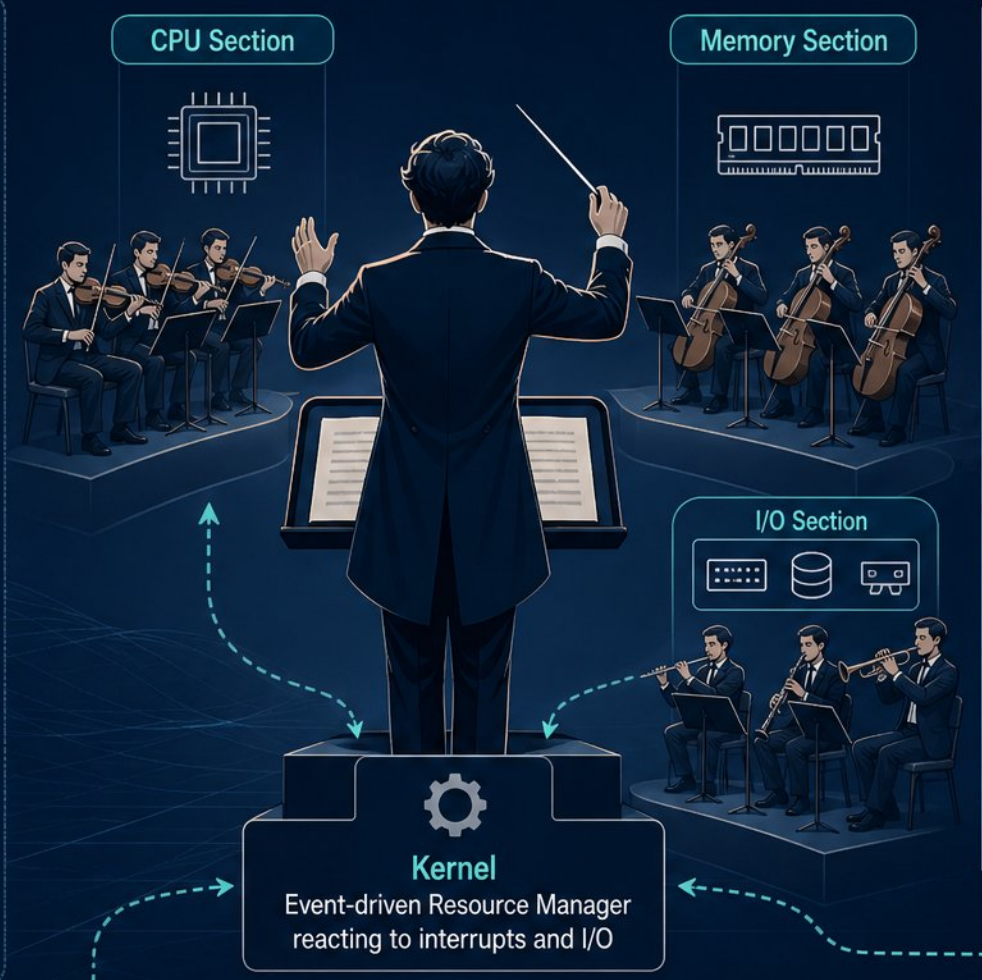
b: blocked (uninterruptible)

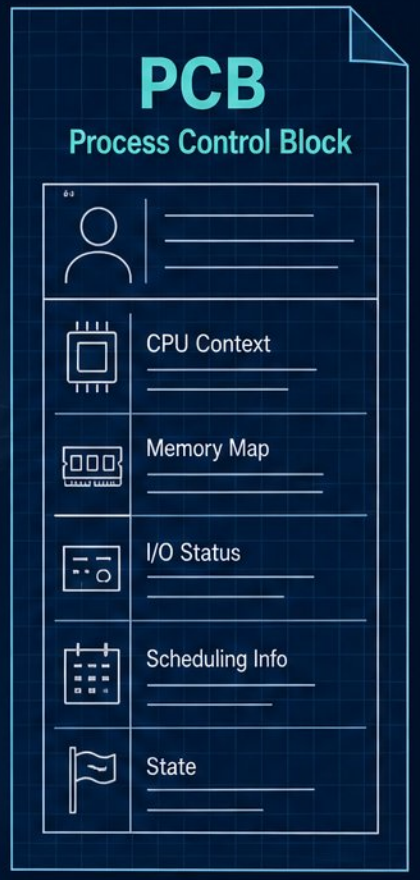
cs: context switches/s

- `vmstat` 1: watch `r` (run queue vs. cores), `b` (blocked), `cs` (switches/s)
- Process states: R (running/ready), S (sleeping), D (uninterruptible I/O)
- `r` > CPU cores = CPU bottleneck; `b` > 0 = I/O bottleneck
- High `cs`: excessive switches waste CPU on overhead and cache thrashing
- `pidstat -w`: drill into per-process voluntary vs. involuntary switch counts

Summary: The Orchestration of a Running Program

- Loader maps binary into address space, dispatcher starts, scheduler multiplexes on CPU
- OS virtualizes CPU and memory for safe, fair hardware sharing
- Kernel as event-driven resource manager reacting to interrupts and I/O
- Context switching and MLFQ scheduler deliver responsive interactivity and throughput





Loader
Maps binary into address space

Dispatcher
Starts execution

Scheduler
Multiplexes on CPU

Context Switching
Saves and restores process state

PersonWise.

PersonWise • AI digital-human course platform

Scan the code or click anywhere to watch this course live, with voice Q&A

personwise.ai/t/f6d1fab0/public