

15-410

“...Everything old is new again...”

Scheduling
Mar. 20, 2006

Dave Eckhardt

Bruce Maggs

Synchronization

Project 3 suggestions

- **Three regular meeting times per week**
 - **Two hours or more at each meeting**
 - **Begin by asking questions about each other's code**
 - » **Requires having read code before meeting**
 - » **Requires “quiet time” between check-ins and meeting**
- **Source control**
 - **Frequent merges, not a single “big bang” at end**
- **Leave time at end for those multi-day bugs**

Synchronization

Checkpoint 2

- Friday
- In cluster, *mandatory*
- Again we will ask you to run demo programs

Synchronization

Checkpoint 3

- “End of third week”
- No cluster meeting – regular lecture
- Expect: code drop, milestone-estimation form
 - Bboard post today/tomorrow
 - Spending the time to really *plan* is worthwhile

Outline

Chapter 5: Scheduling

- Scheduling-people terminology note
 - “Waiting time” means “time spent runnable but stuck in a scheduler queue” - not “time waiting for an event to awaken you”
 - “Task” means “something a scheduler schedules” (we say “thread”)

CPU-I/O Cycle

Process view: 2 states

- Running
- Waiting for I/O
- Life Cycle
 - I/O (loading executable), CPU, I/O, CPU, ..., CPU (`exit()`)

System view

- Running, Waiting
- Runnable – not enough processors for you right now

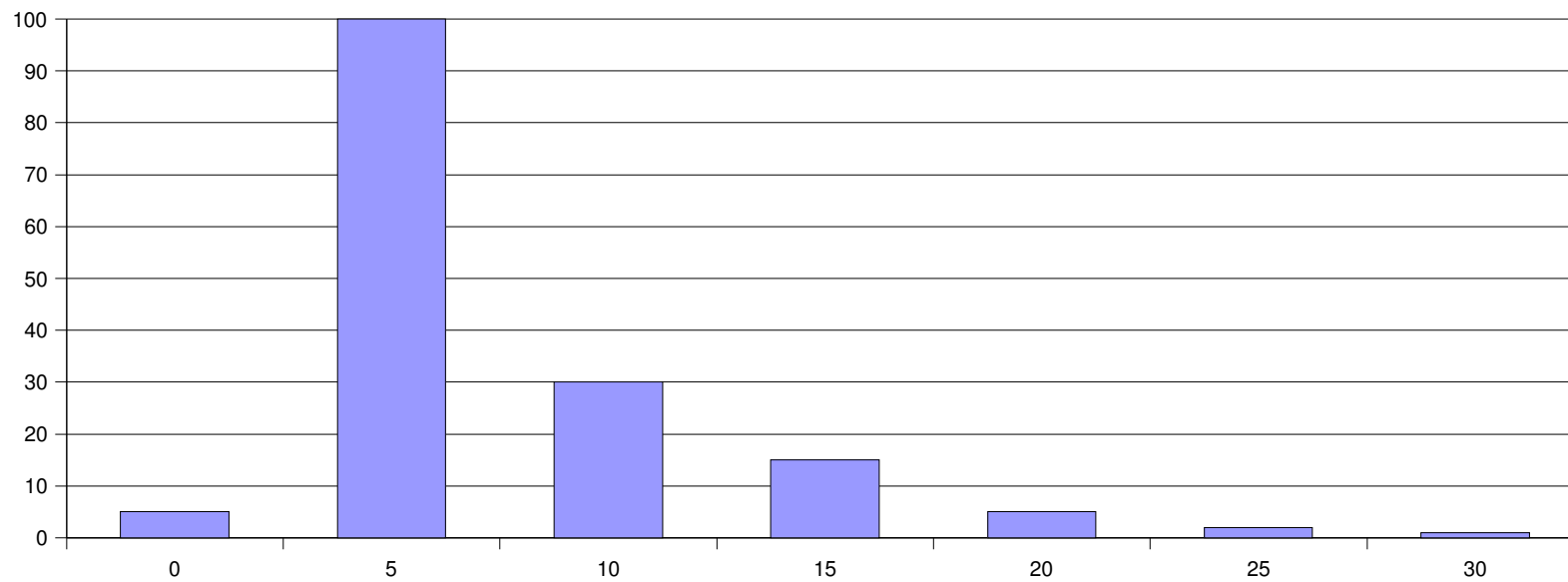
Running \Rightarrow waiting is mostly voluntary

- How long do processes choose to run before waiting?

CPU Burst Lengths

Overall

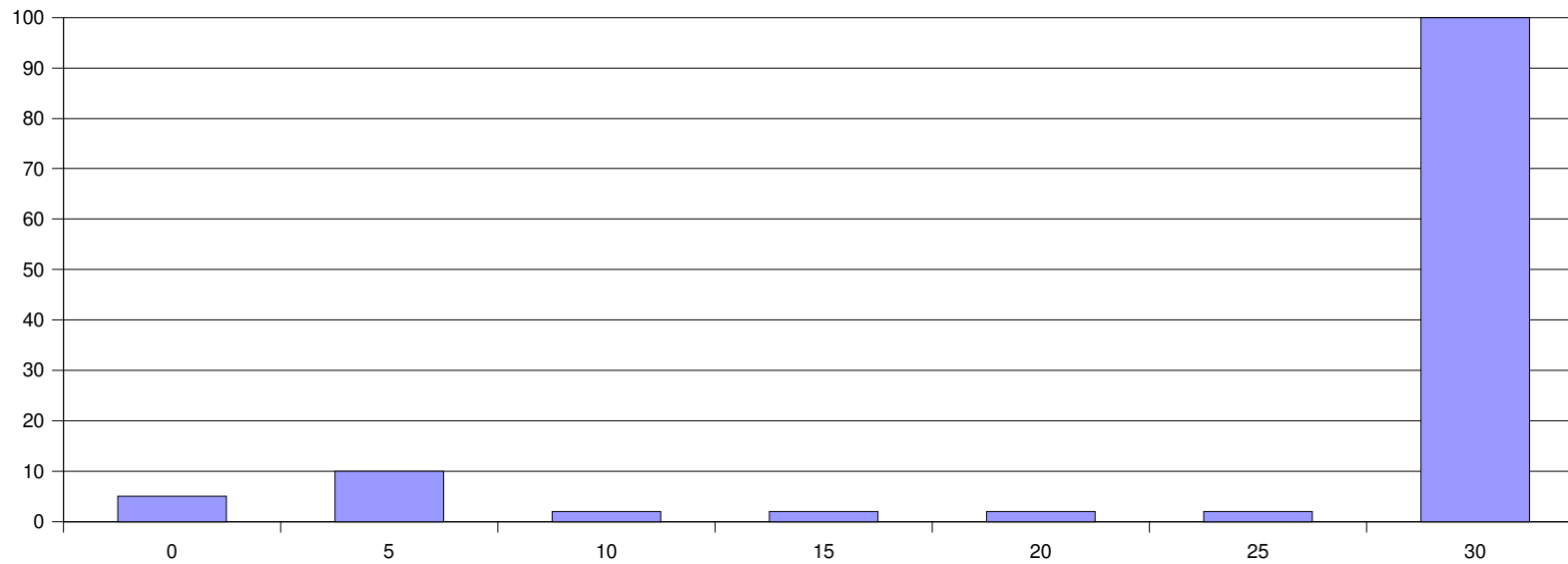
- Exponential fall-off in CPU burst length



CPU Burst Lengths

“CPU-bound” program

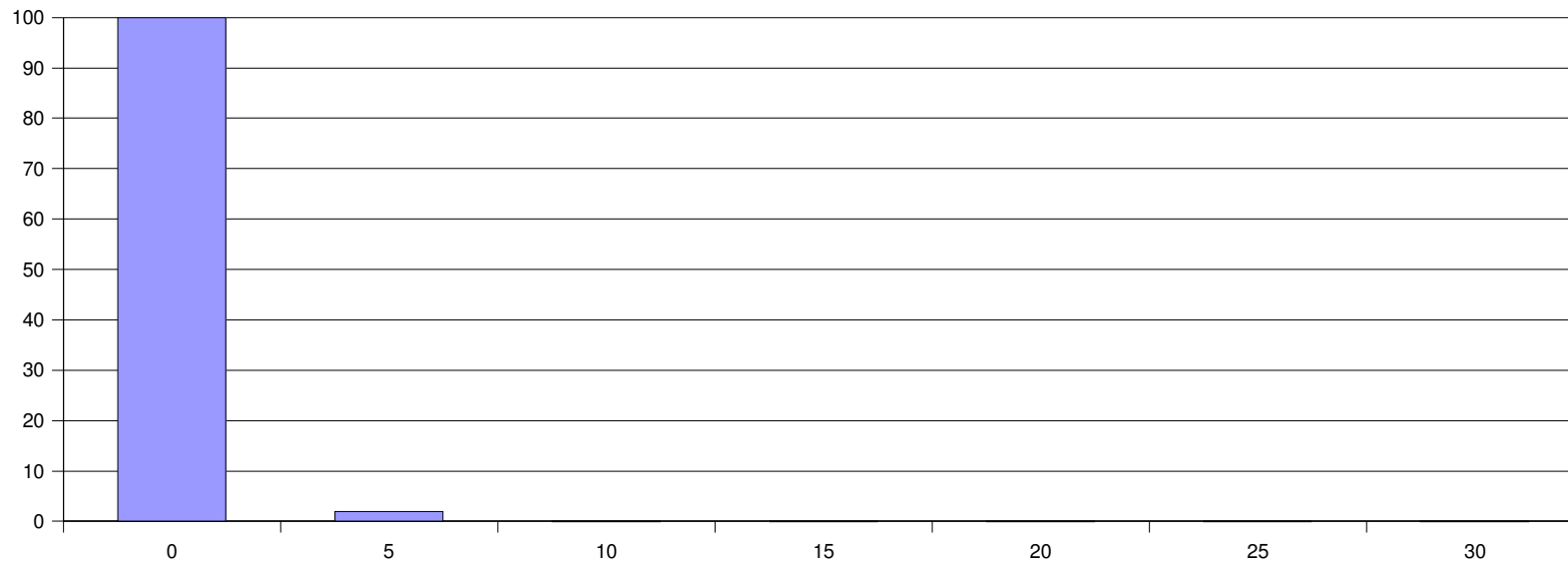
- Batch job
- Long CPU bursts



CPU Burst Lengths

“I/O-bound” program

- Copy, Data acquisition, ...
- *Tiny* CPU bursts between system calls



Preemptive?

Four opportunities to schedule

- A running process waits (I/O, child, ...)
- A running process exits
- A waiting process becomes runnable (I/O done)
- Other interrupt (clock, page fault)

Multitasking types

- Fully Preemptive: *All four cause scheduling*
- “Cooperative”: only first two

Preemptive *kernel*?

Preemptive multitasking

- All four cases cause context switch

Preemptive *kernel*

- All four cases cause context switch *in kernel mode*
- This is a goal of Project 3
 - System calls: interrupt disabling only when really necessary
 - Clock interrupts should suspend system call execution
 - » So fork() should *appear* atomic, but not *execute* that way

CPU Scheduler

Invoked when CPU becomes idle

- Current task blocks
- Clock interrupt

Select next task

- *Quickly*
- PCB's in: FIFO, priority queue, tree, ...

Switch (using “dispatcher”)

- Your term may vary

Dispatcher

Set down running task

- Save register state
- Update CPU usage information
- Store PCB in “run queue”

Pick up designated task

- Activate new task's memory
 - Protection, mapping
- Restore register state
- “Return” to whatever the task was previously doing

Scheduling Criteria

System administrator view

- Maximize/trade off
 - CPU utilization (“busy-ness”)
 - Throughput (“jobs per second”)

Process view

- Minimize
 - Turnaround time (everything)
 - Waiting time (runnable but not running)

User view (interactive processes)

- Minimize response time (input/output latency)

Algorithms

Don't try these at home

- FCFS
- SJF
- Priority

Reasonable

- Round-Robin
- Multi-level (plus feedback)

Multiprocessor, real-time

FCFS- First Come, First Served

Basic idea

- Run task until it relinquishes CPU
- When runnable, place at end of FIFO queue

Waiting time *very* dependent on mix

- Some processes run briefly, some much longer

“Convoy effect”

- N tasks each make 1 I/O request, stall
- 1 task executes very long CPU burst
- Lather, rinse, repeat
- N “I/O-bound tasks” can't keep I/O device busy!

SJF- Shortest Job First

Basic idea

- Choose task with shortest *next* CPU burst
- Will give up CPU soonest, be “nicest” to other tasks
- Provably “optimal”
 - Minimizes average waiting time across tasks
- *Practically impossible* (oh, well)
 - Could *predict* next burst length...
 - » Text presents exponential average
 - » Does not present evaluation (Why not? Hmm...)

Priority

Basic idea

- Choose “most important” waiting task
 - (Nomenclature: does “high priority” mean $p=0$ or $p=255$?)

Priority assignment

- Static: fixed property (engineered?)
- Dynamic: function of task behavior

Big problem: *Starvation*

- “Most important” task gets to run often
- “Least important “ task may *never* run
- Possible hack: priority “aging”

Round-Robin

Basic idea

- Run each task for a fixed “time quantum”
- When quantum expires, append to FIFO queue

“Fair”

- But not “provably optimal”

Choosing quantum length

- Infinite (until process does I/O) = FCFS
- Infinitesimal (1 instruction) = “Processor sharing”
 - A technical term used by theory folks
- Balance “fairness” vs. context-switch costs

True “Processor Sharing”

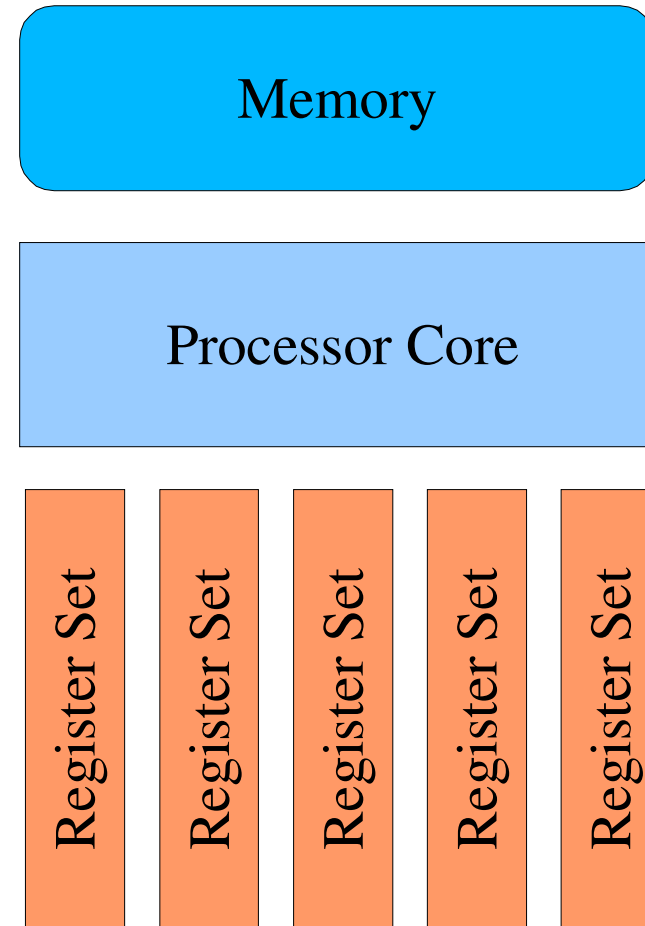
CDC Peripheral Processors

Memory latency

- *Long*, fixed constant
- Every instruction has a memory operand

Solution: round robin

- Quantum = 1 instruction



True “Processor Sharing”

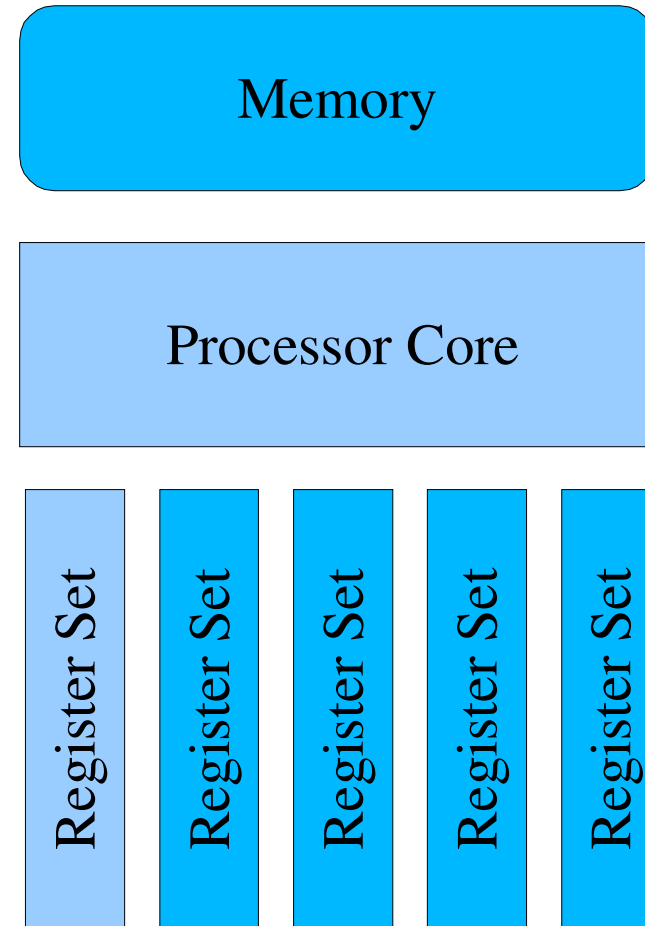
CDC Peripheral Processors

Memory latency

- *Long*, fixed constant
- Every instruction has a memory operand

Solution: round robin

- Quantum = 1 instruction
- One “process” running
- N-1 “processes” waiting



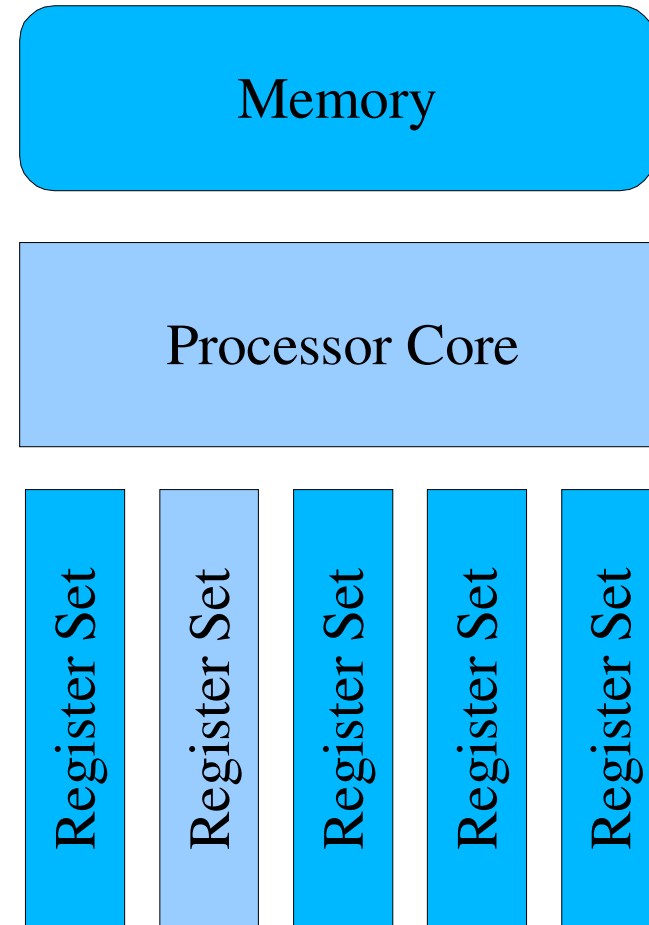
True “Processor Sharing”

Each instruction

- “Brief” computation
- One load xor one store
 - Sleeps process N cycles

Steady state

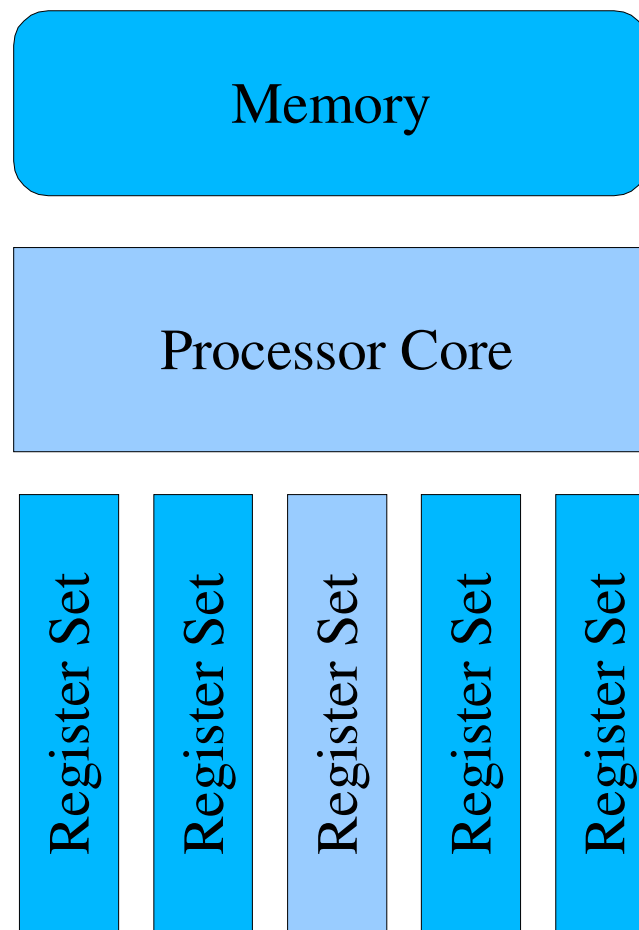
- Run when you're ready
- Ready when it's your turn



Everything Old Is New Again

Intel “hyperthreading”

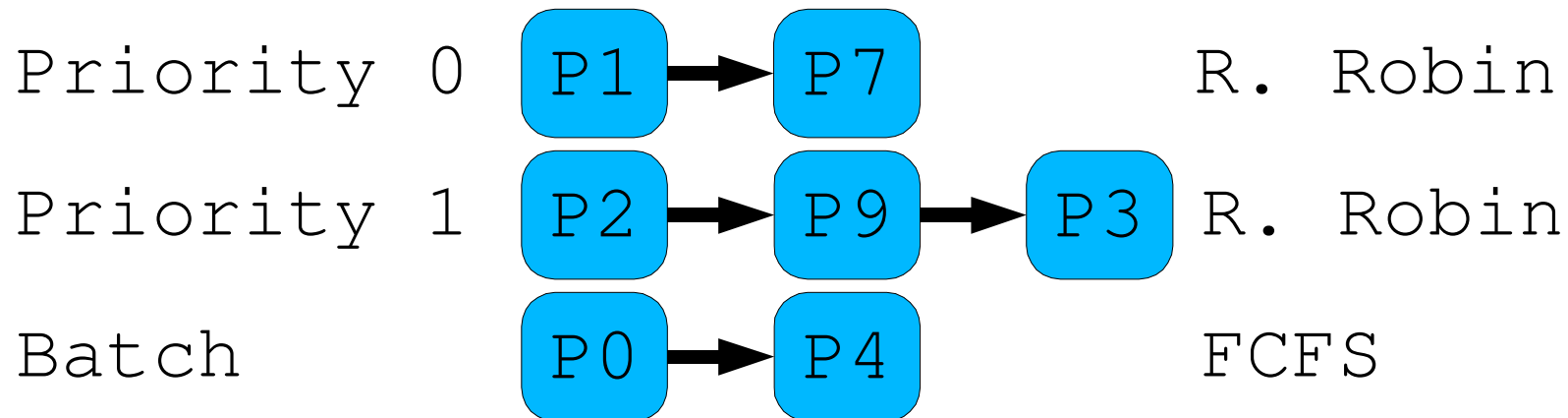
- N register sets
- M functional units
- Switch on long-running operations
- Sharing less regular
- Sharing illusion more lumpy
 - Good for some application *mixes*
 - *Awful* for others
 - “Hyperthreading Hurts Server Performance, Say Developers”



Multi-level Queue

N independent process queues

- One per priority
- Algorithm per-queue



Multi-level Queue

Inter-queue scheduling

- **Strict priority**
 - Pri 0 runs before Pri 1, Pri 1 runs before batch – *every time*
- **Time slicing (e.g., weighted round-robin)**
 - Pri 0 gets 2 slices
 - Pri 1 gets 1 slice
 - Batch gets 1 slice

Multi-level *Feedback* Queue

N queues, different quanta

Block/sleep before quantum expires?

- Added to end of your queue

Exhaust your quantum?

- Demoted to slower queue
 - Lower priority, typically longer quantum

Can you be promoted back up?

- Maybe I/O promotes you
- Maybe you “age” upward

Popular “time-sharing” scheduler

Multiprocessor Scheduling

Common assumptions

- Homogeneous processors (same speed)
- Uniform memory access (UMA)

Load sharing / Load balancing

- Single global ready queue – no false idleness

Processor Affinity

- Some processor may be more desirable or necessary
 - » Special I/O device
 - » Fast thread switch
 - » $1/N^{\text{th}}$ of memory may be faster

Multiprocessor Scheduling - “SMP”

Asymmetric multiprocessing

- Also known as “master/slave”
- One processor is “special”
 - Executes all kernel-mode instructions
 - Schedules other processors
- “Special” aka “bottleneck”

Symmetric multiprocessing - “SMP”

- “Gold standard”
- Tricky

Real-time Scheduling

Hard real-time

- System must *always* meet performance goals
 - Or it's *broken* (think: avionics)
- Designers must describe task requirements
 - Worst-case execution time of instruction sequences
- “Prove” system response time
 - Argument or automatic verifier
- Cannot use indeterminate-time technologies
 - Disks!

Real-time Scheduling

Soft real-time

- “Occasional” deadline failures tolerable
 - CNN video clip on PC
 - DVD playback on PC
- *Much* cheaper than hard real-time
 - Real-time extension to timesharing OS
 - » POSIX real-time extensions for Unix
 - Can estimate (vs. prove) task needs
- Priority scheduler
- Preemptible kernel implementation

Scheduler Evaluation Approaches

“Deterministic modeling”

- aka “hand execution”

Queueing theory

- Math gets big fast
- Math sensitive to assumptions
 - » May be unrealistic (aka “wrong”)

Simulation

- Workload model or trace-driven
- GIGO hazard (either way)

Summary

Round-robin is ok for simple cases

- Certainly 80% of the conceptual weight
- *Certainly* good enough for P3
 - Speaking of P3...
 - » Understand preemption, don't evade it

“Real” systems

- Some multi-level feedback
- Probably some soft real-time