# 15-410

***"...Everything old is new again..."***

## Scheduling
## Nov. 5, 2017

**Dave Eckhardt**

**Dave O'Hallaron**

**Roger Dannenberg**

# Outline

## Chapter 5 (or Chapter 7): Scheduling

- Scheduling-people/textbook terminology note
  - "Waiting time" means "time spent runnable but stuck in a scheduler queue"
    - Not "time waiting for the actual event to awaken you"!
  - "Task" means "something a scheduler schedules" (we say "thread" or sometimes "runnable")

# CPU-I/O Cycle

*Process* **view: 2 states**

- Running
- Blocked on I/O

Life Cycle:

  - I/O (loading executable), CPU, I/O, CPU, .., CPU (`exit()`)

*System* **view**

- Running
- Blocked on I/O
- Runnable (i.e. Waiting) – not enough processors right now

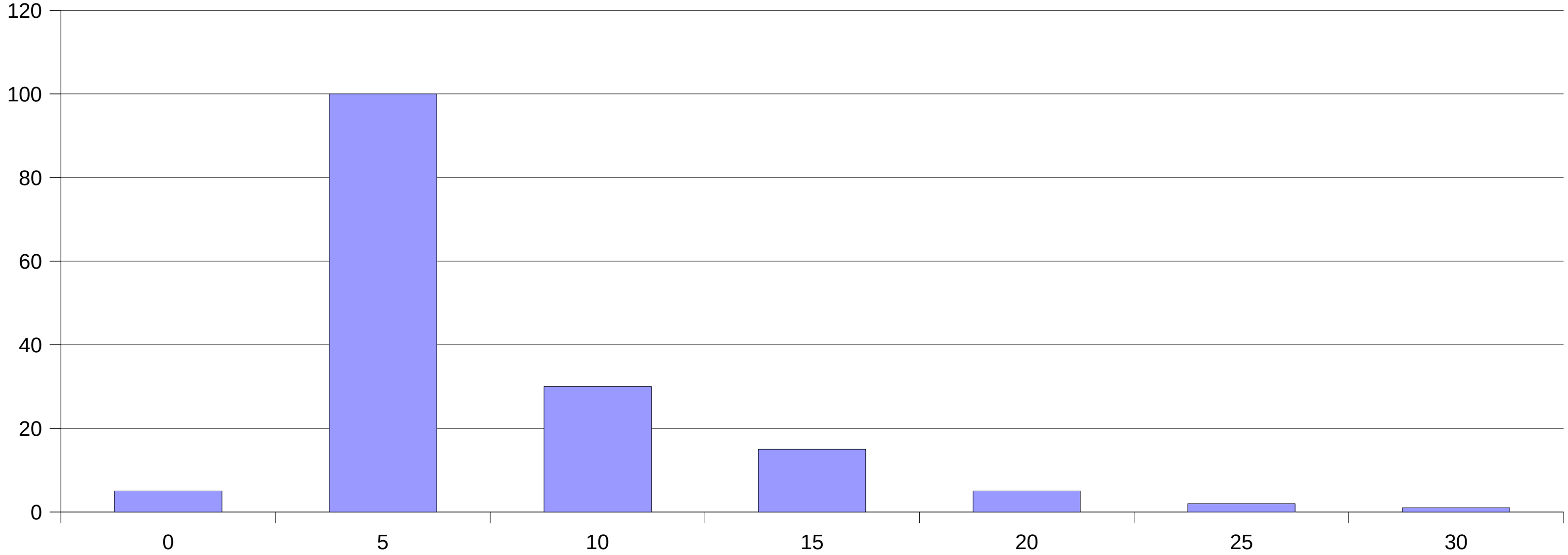
**Running ⇒ blocked mostly depends on program**

- How long do processes run before blocking?

# CPU Burst Lengths

## In general

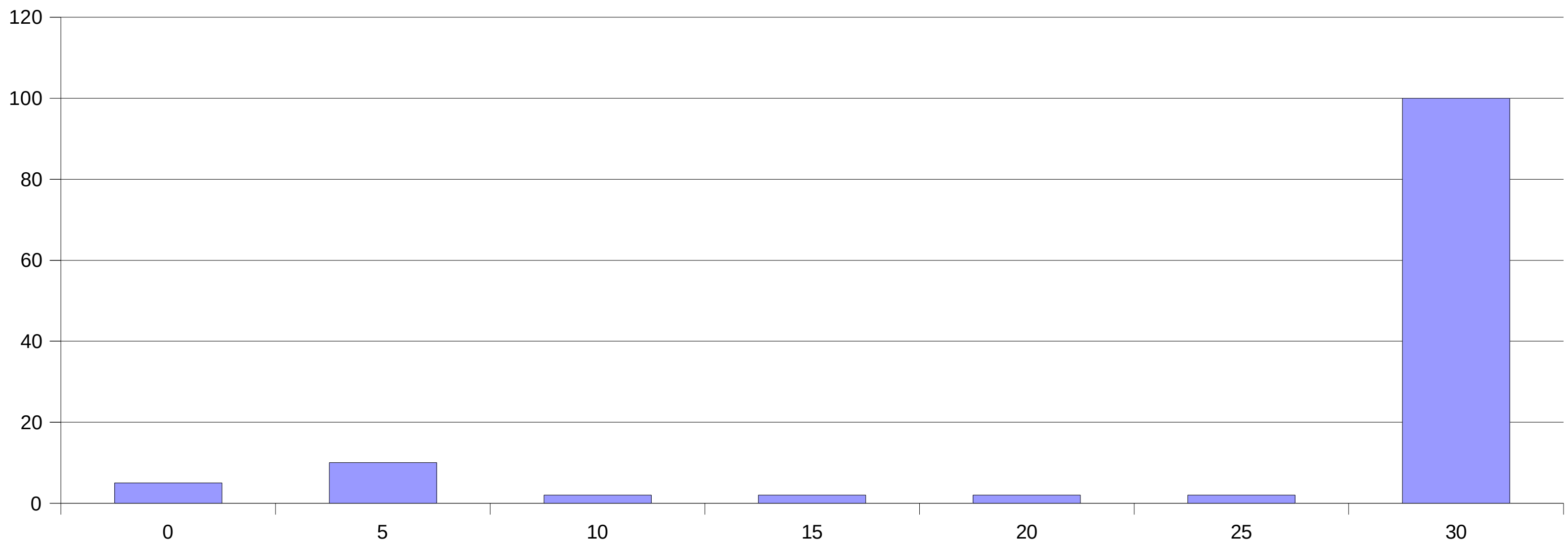
- Exponential fall-off in CPU burst length



# CPU Burst Lengths

## "CPU-bound" program

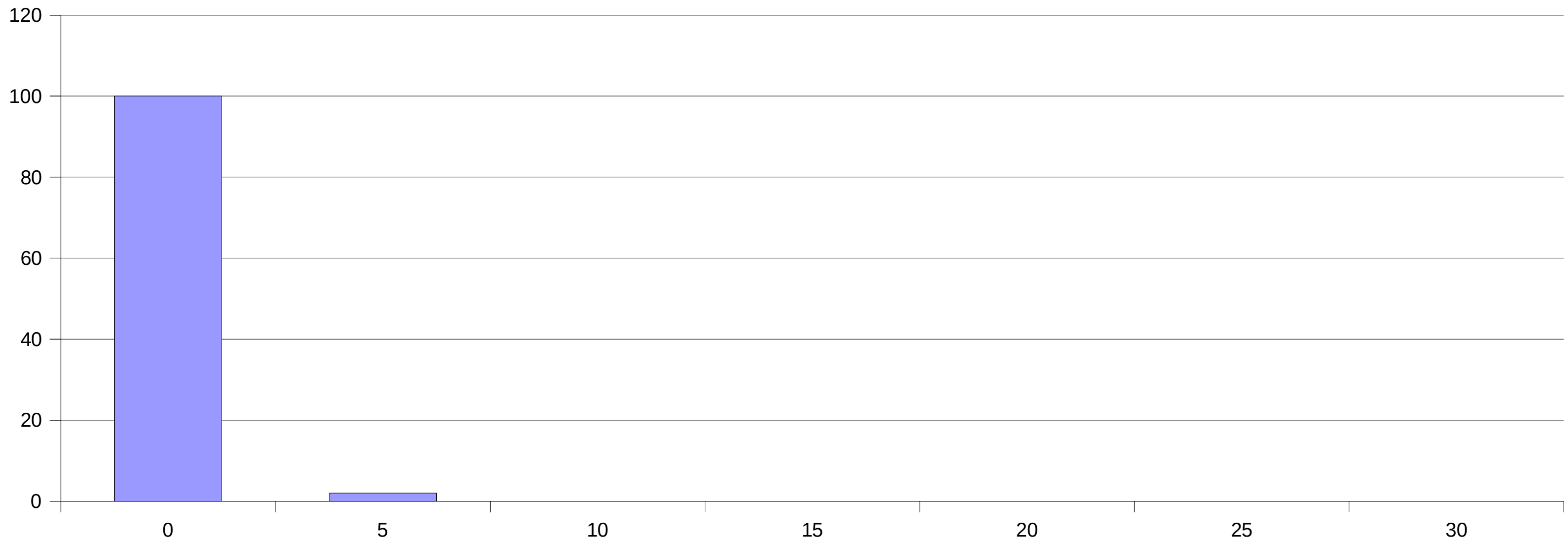
- **Batch job**
- **Long CPU bursts**



# CPU Burst Lengths

## "I/O-bound" program

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



# Why Scheduling?

**What if we let a CPU-bound program run to completion?**

- What happens to I/O-bound programs?

**What if we run an I/O-bound program whenever it is runnable?**

- What happens to CPU-bound programs?

# Preemptive?

## Four opportunities to schedule

- **A running process blocks (I/O, page fault, wait(), ...)**
- **A running process exits**
- **A blocked process becomes runnable (I/O done)**
- **Other interrupt (clock)**

## 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 and/or time passes**

- Current task blocks
- Clock interrupt

**Select next task**

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

**Switch (using "dispatcher")**

- Your term may vary

# Dispatcher

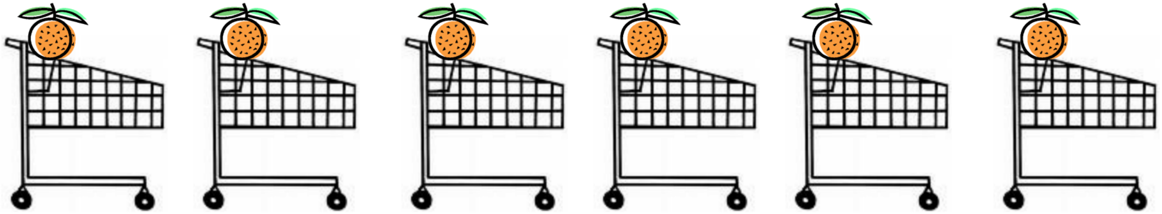
## Set down running task

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

## Pick up designated task

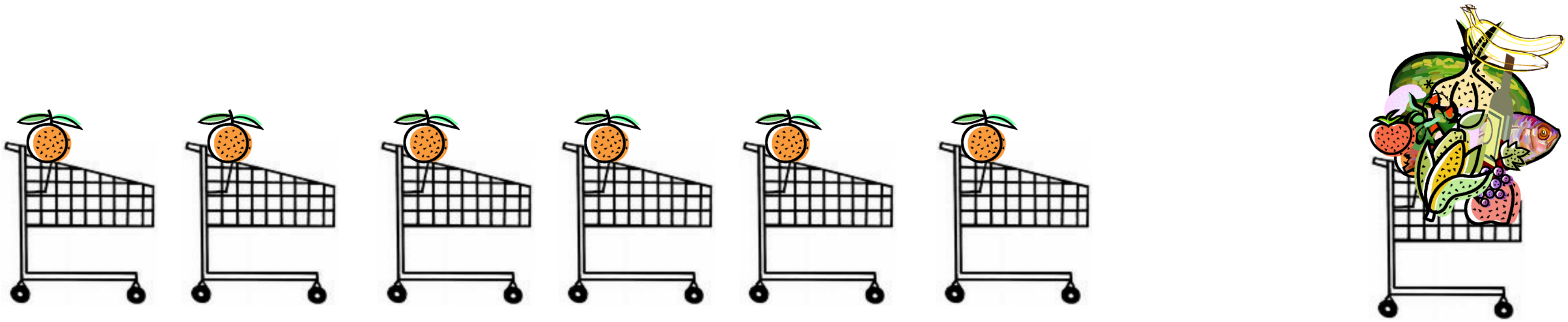
- Activate new task's memory
  - Protection, mapping
- Restore register state
- “Return” to whatever the task was previously doing

# Consider…



**Who goes first? Last?**

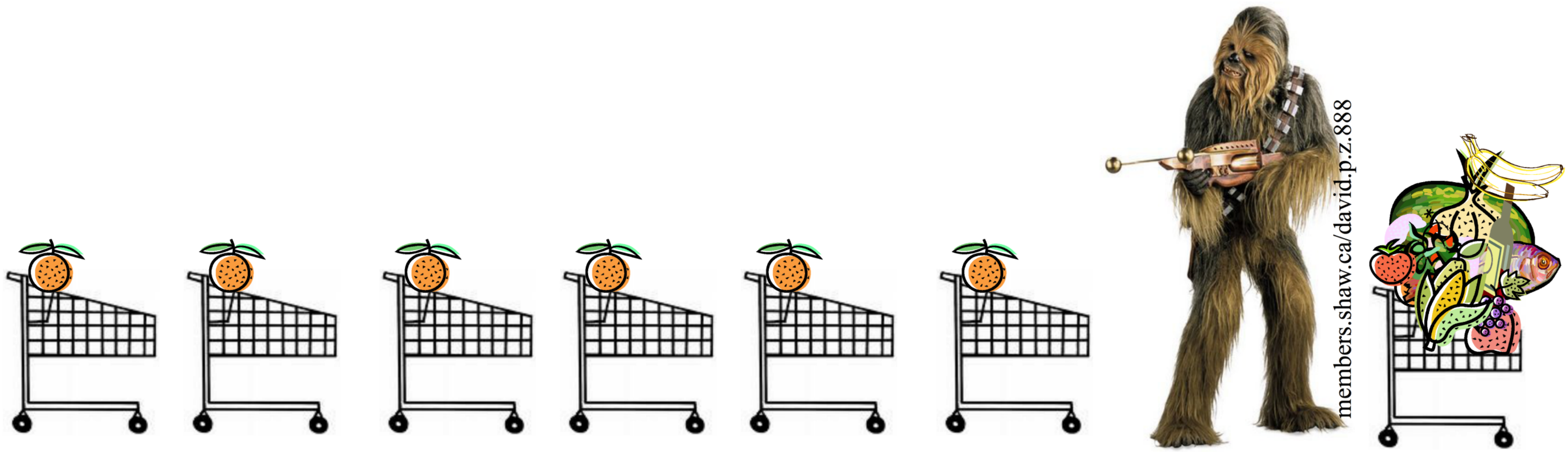
# Consider…



**Who goes first? Last?**

**Now who goes first? Last?**

# Consider…




**Who goes first? Last?**

**Now who goes first? Last?**

**Does this change things?**

# Scheduling Criteria

## System administrator view

- Maximize/trade off
  - CPU utilization (“busy-ness”)
    - Was important when buying computers was expensive
    - Now heat and power often cost more than silicon
  - Throughput (“jobs per second”)

## Process view

- Minimize
  - Turnaround time (everything, fork() to exit())
  - Waiting time (runnable but not running)

## User view (interactive processes)

- Minimize response time (input/output latency)
- *Predictable* response time (“Why is it slow today??”)

# Algorithms

## Don't try these at home

- FCFS
- SJF
- Priority

## Reasonable

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

## Multiprocessor

- Load balancing
- Processor affinity

## 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 (e.g., file copy)
- 1 task executes very long CPU burst
  - All I/O tasks become runnable during this time
- Lather, rinse, repeat
  - Result: N "I/O-bound tasks" can't keep I/O devices 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 suggests averaging recent burst lengths
      - Does not present evaluation (Why not?  Hmm...)

# 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 suggests averaging recent burst lengths
      - Does not present evaluation (Why not? Hmm...)
    - Sometimes applications ***can*** state their remaining work
      - Harchol-Balter et al., “Size-Based Scheduling to Improve Web Performance”, ACM TOCS 21:2, 5/2003

# 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 behaviour

## Big problem: *Starvation*

- "Most important" task gets to run often
- "Least important" task may *never* run
- Common hack: priority "ageing"

# 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"

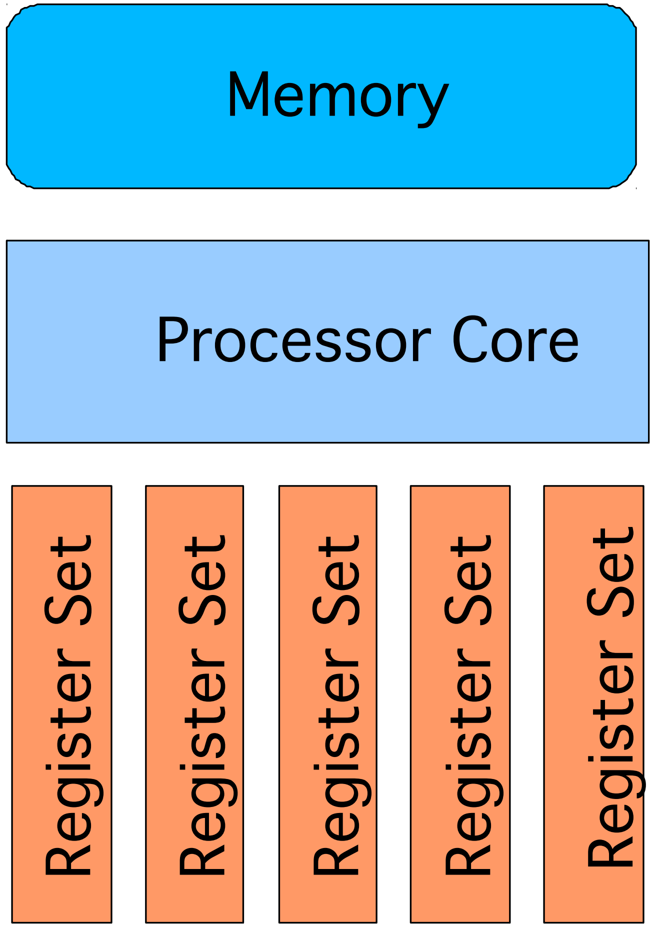
## CDC Peripheral Processors

## Memory latency

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

## Solution: round robin

- Quantum = 1 instruction



# *True* "Processor Sharing"

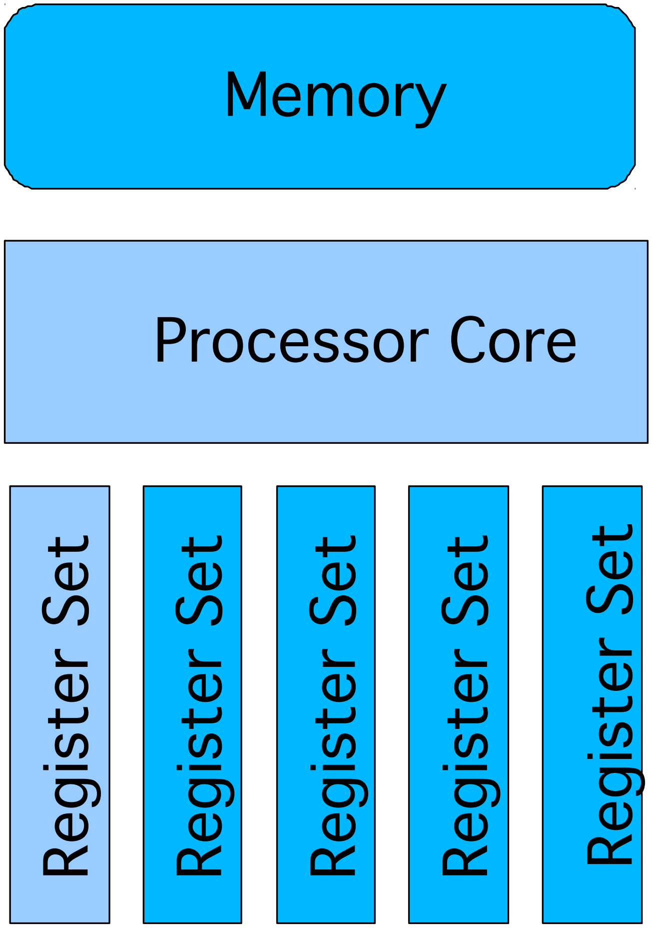
## CDC Peripheral Processors

## Memory latency

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

## Solution: round robin

- Quantum = 1 instruction
- One "process" running
- N-1 "processes" waiting on memory



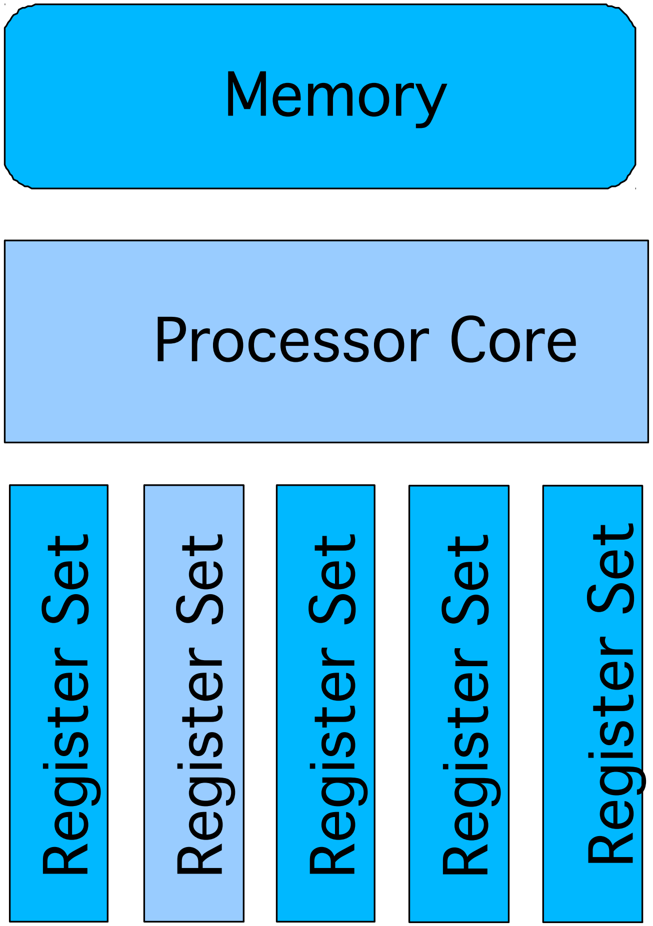
# *True* "Processor Sharing"

**Each instruction**

- "Brief" computation
- One load or one store
  - Sleeps process N cycles

**Steady state**

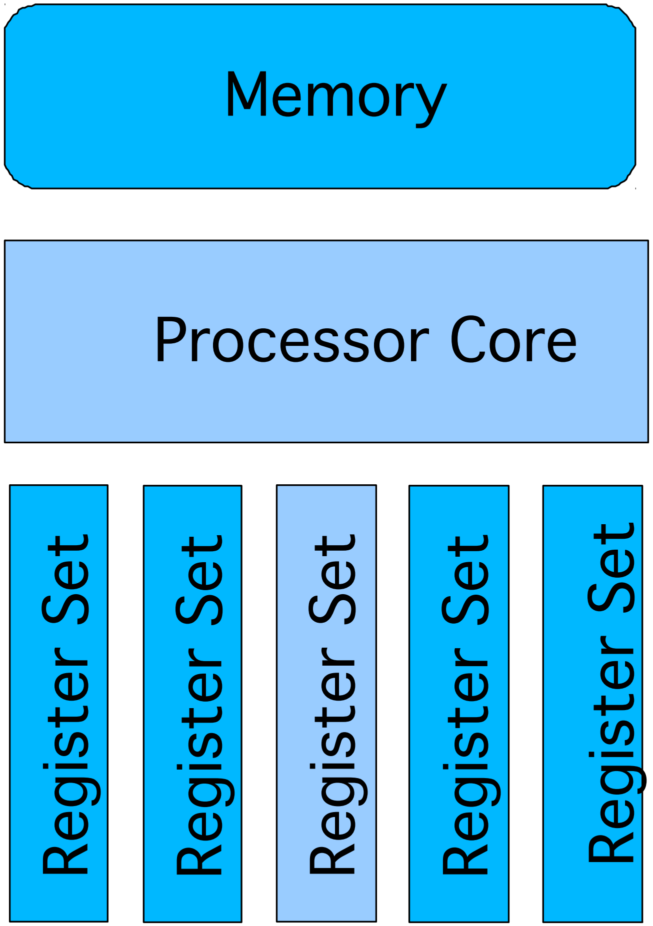
- Run when you're ready
- Ready when it's your turn



# Everything Old Is New Again

**Intel "hyperthreading"**

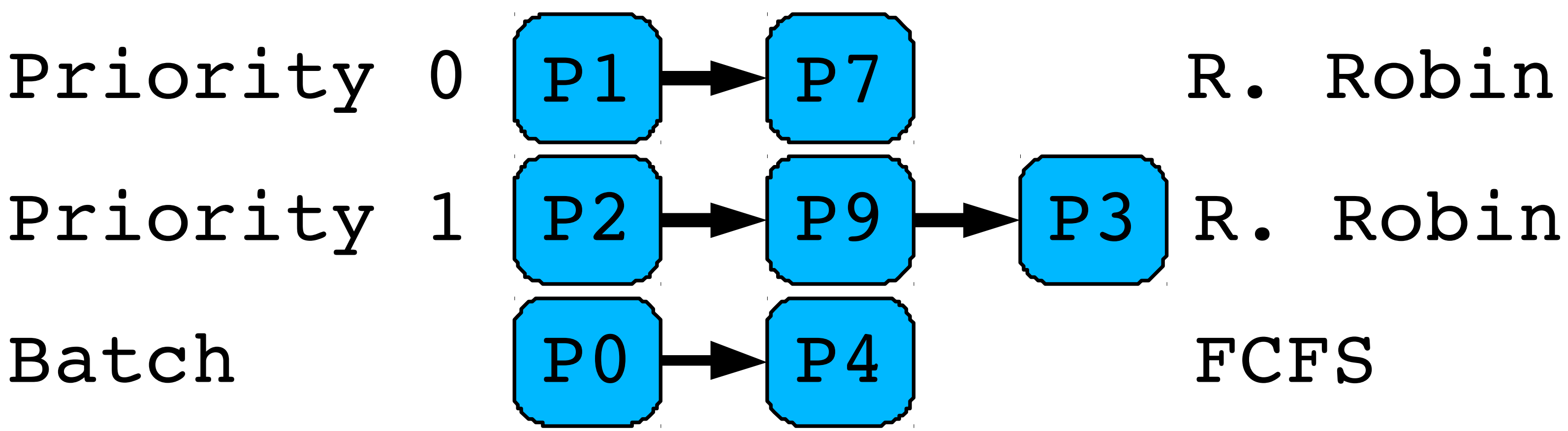
- N register sets
- M functional units
- Switch on long-running operations
- Sharing less regular
- Sharing illusion more lumpy
  - Good for some application *mixes*
  - *Awful* for others
  - "Hyperthreading Hurts Server Performance, Say Developers" - ZDNet UK, 2005-11-18



# 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 ("good runnable")

**Exhaust your quantum?**

- Demoted to slower queue ("bad runnable!")
  - 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)

## Goal: Load sharing / Load balancing

- "Easy": single global ready queue – no false idleness

# Multiprocessor Scheduling

## Common assumptions

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

## Goal: Load sharing / Load balancing

- "Easy": single global ready queue – no false idleness

## But!

- Single global ready queue is a contention "hot spot"
- "Processor Affinity": some processor may be more desirable or necessary
  - Special I/O device
  - Fast thread switch
  - Resuming onto most-recent CPU may find some stuff still cached
  - $1/N^{th}$ of memory may be faster - "NUMA"

# Scheduler Evaluation Approaches

## "Deterministic modeling"

- aka "hand execution"

## Queueing theory

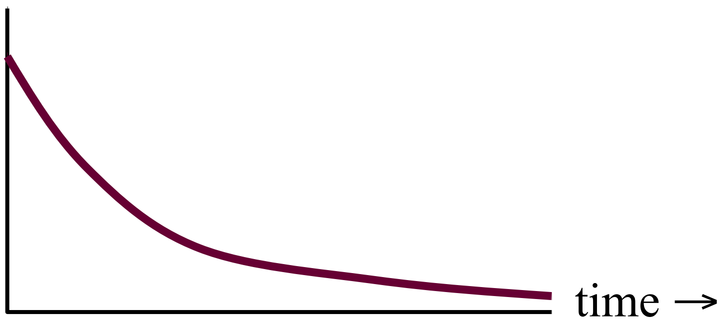
- Often gives fast and useful *approximations*
- Math gets big fast
- Math sensitive to assumptions
  - May be unrealistic (aka "wrong")

## Simulation

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

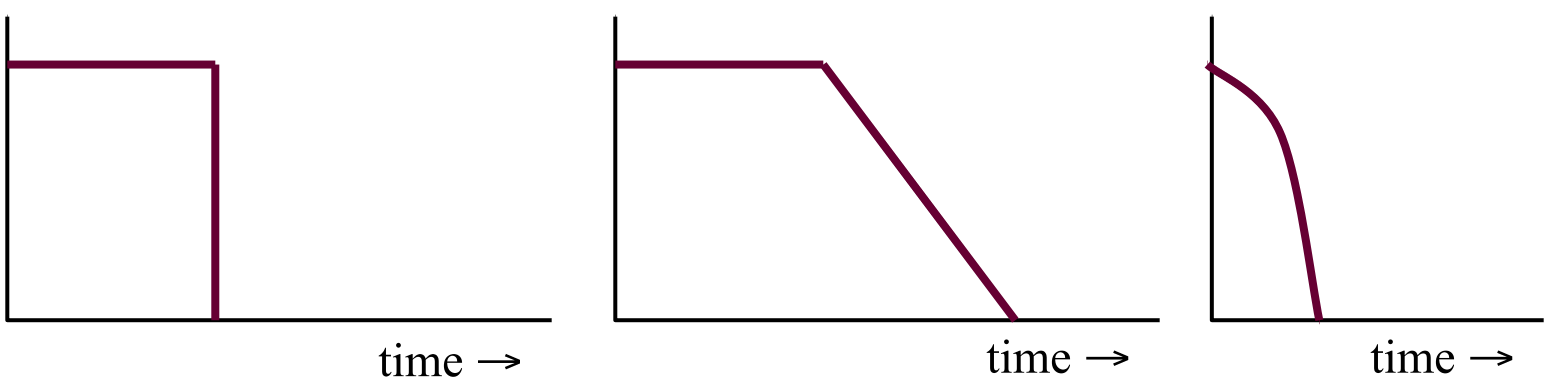
# Real-Time Scheduling

**What's a computation worth?**




**"Real Time": No (extra) value if early**

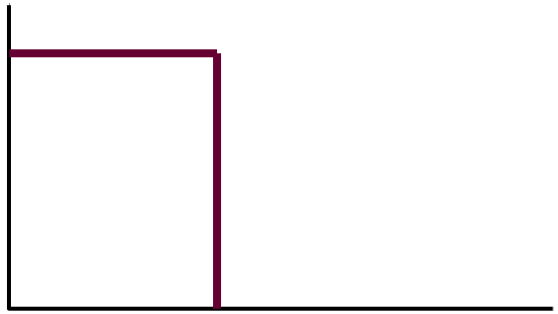
**(or in some cases, curve just falls off fast)**



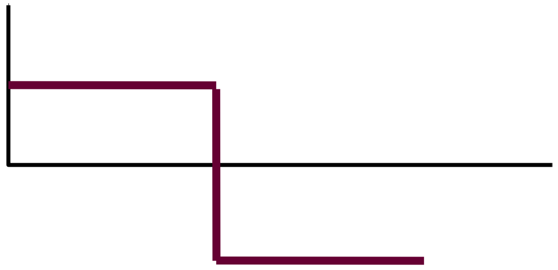
# "Hard Real-Time" = ?

**Multiple definitions are used**

- "Very fast response time" – 10$\mu$s?
- "No value" if results are late
- "Very costly" if late
- "Never" late



"No value"



"Very costly"

# Hard Real-Time Scheduling

## 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... Networks...

## Solutions often involve

- Simplified designs
- Over-engineered systems
- Dedicated hardware
- Specialized OS

# Soft Real-Time Scheduling

## Computation still has value after deadline

- Think "User Interface"
- Many control systems
  - (if the fly-by-wire system doesn't move the elevator within 50ms, probably still good to to it within 100ms)

## Performance is not critical (no one will die)

- YouTube video
- Skype
  - Note that late packets cause audio drop-out.
- CD-R writing software
  - Resulting CD can be corrupted

# Soft Real-Time Scheduling

**Now commonly supported in generic OS**

- POSIX real-time extensions for Unix

**Priority-based scheduler**

**Preemptible kernel implementation**

# 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
- Multi-processor scheduling is a big deal

## Real-Time Systems Concepts

- Terminology: soft, hard, deadline
- Key issue: "priority inversion" (see text)