CS 475
Operating Systems

Department of Mathematics
and Computer Science

Lecture 5
CPU Scheduling

What Is CPU Scheduling?

- Roadmap:
  - Policy: We want concurrency among threads
  - Mechanism: We can now context switch between threads

- CPU Scheduling Policies concern the selection of the next thread to run
  - When does the scheduler get invoked?
  - Which thread should run next?
  - How long does the thread run for?
  - How to ensure "fairness" among threads?
  - How to prevent "starvation?"

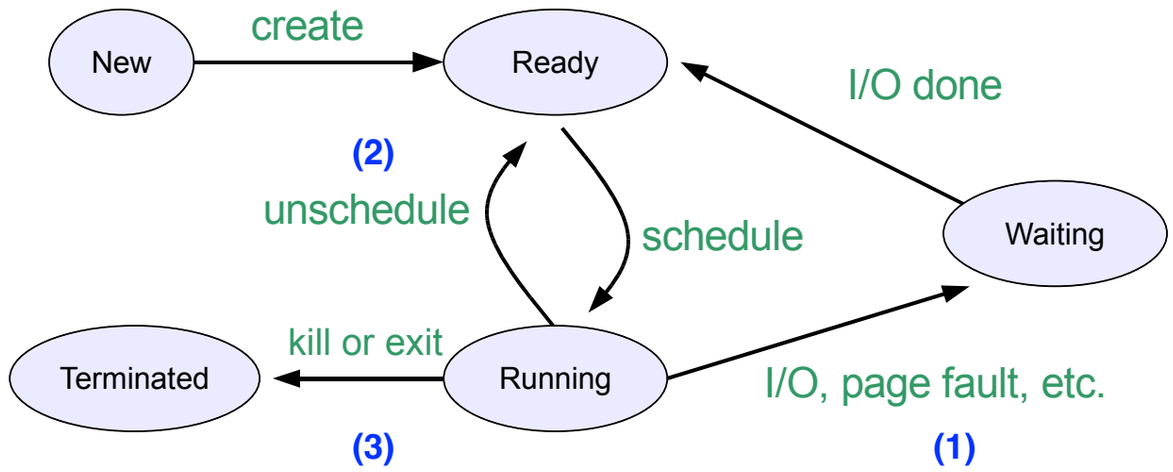
Cooperative (or Non-preemptive) Scheduling

Run scheduler when....

(1) a thread blocks on I/O

(2) the running thread *yields the CPU*

(3) a thread terminates



In the transitions listed above, the OS has no choice but to schedule a new thread.

Windows 3.1, Mac pre-OSX

Preemptive Scheduling

Run scheduler when....

(1) a thread blocks on I/O

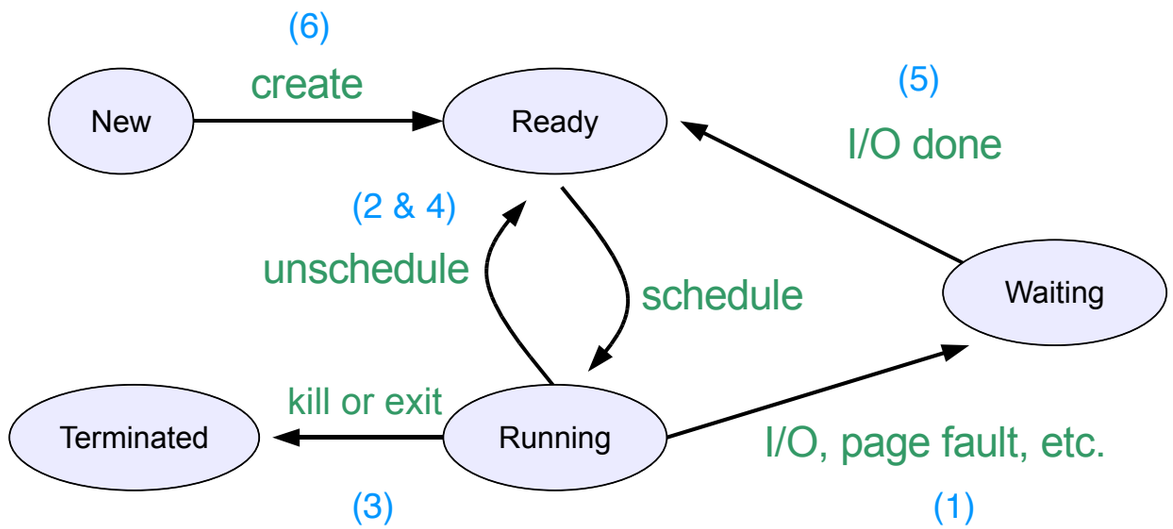
(2) the running thread yields the CPU

(3) a thread terminates

(4) a thread is interrupted

(5) I/O completes for some thread

(6) a new thread is created



Preemptive scheduling: In 4, 5, 6 the OS now gets a *choice.* If it chooses a new thread to run, then the current thread is "preempted"

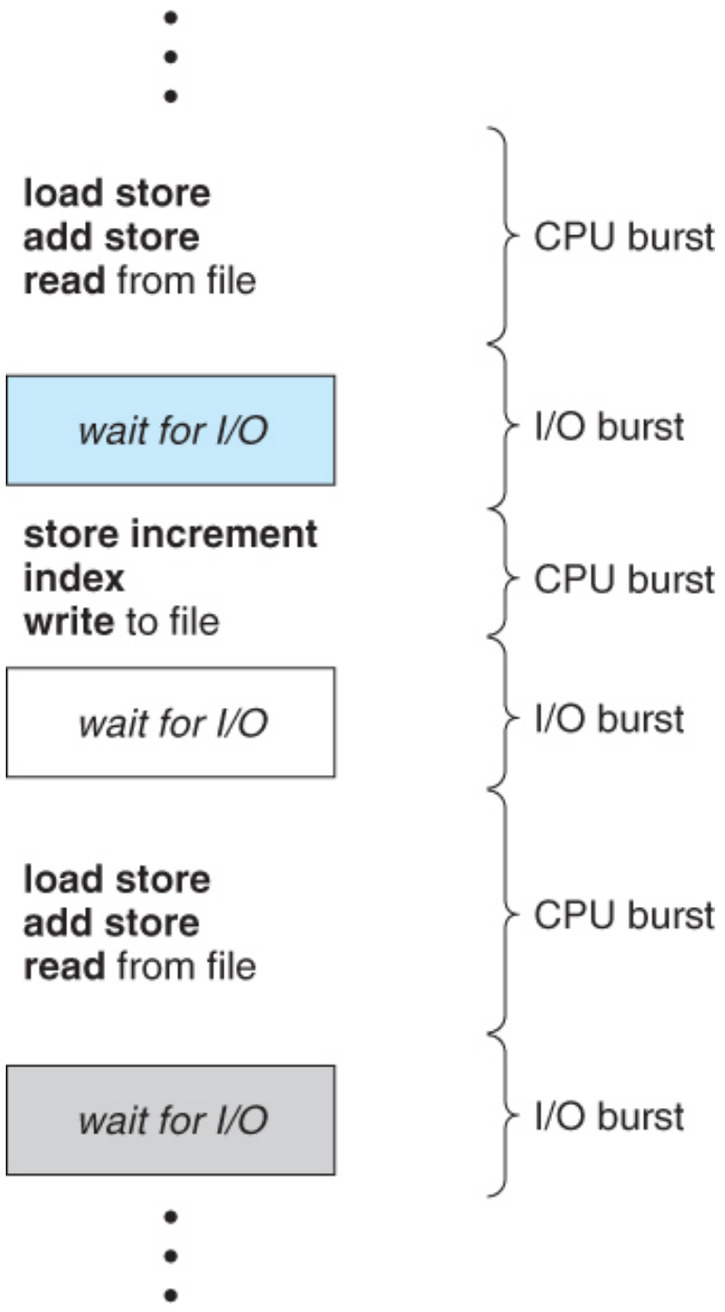
Windows 95+, Mac OS X, Unix, Linux

Goals for This Lecture

- Objectives
  - Characterizing Thread/Process Execution
  - Scheduling Metrics and Goals
- Scheduling Policies
  - Preemptive
  - Non-preemptive
  - Evaluation
- Multiprocessing Considerations
- Conclusion

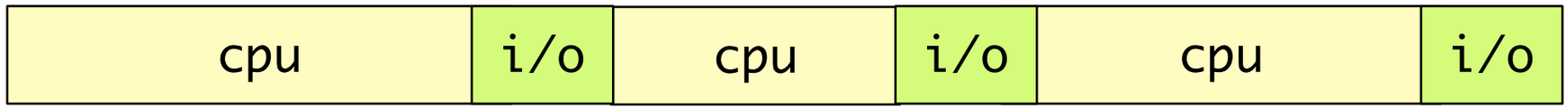
Simple Model of Thread Execution

- Let's observe how a few common programs tend to execute:
  - Chrome
  - git
  - vscode
  - bash
  - gcc
  - eclipse
  - *etc.*



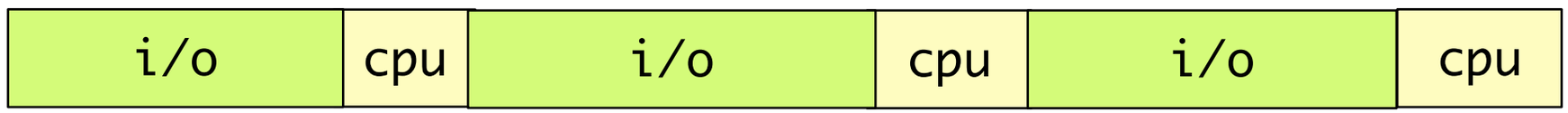
Execution Model (Cont.)

- We characterizing threads to be either:
  - *CPU-Bound (or CPU-Intensive)* is when CPU bursts are long



    - (*e.g.*, graphics algorithms, AI/ML algorithms, gcc, ...)

  - *I/O-Bound (or I/O-Intensive)* is when CPU bursts tend to be short



    - Characteristic of both *interactive* and/or *data-intensive* applications (*e.g.*, shells, git, vs code, SQL databases, web browsers)

Scheduling Policy Metrics

- Wait Time = $scheduled\_time - arrival\_time$
  - "Arrival" refers to when the thread (re)enters the ready queue
- Turnaround Time
  - The time between a thread arrival to its completion:
    $execution\_time + wait\_time$
- Scheduling Overhead
  - The time required for the OS to select the next thread to run
- Fairness (winners and losers)
  - How fair is the scheduler being to all threads?
- Starvation
  - Do any thread risk never getting picked to run?

Goals for This Lecture

- Objectives
  - Characterizing Thread/Process Execution
  - Scheduling Metrics and Goals
- Scheduling Policies
  - FCFS, SJF, RR, SRTF, MLFQ
- Multiprocessing Considerations
- Conclusion

FCFS Policy

- The earliest scheduling policy (used in batch processing)
  - Seems fair: whichever thread has been waiting the longest gets picked
  - Threads are scheduled in the order they arrive

FCFS Policy

Grab the thread from the head of the Ready Queue when:

- A thread terminates, or
- A thread blocks on I/O or self-yields
- *(So, it's a cooperative, or non-preemptive, policy)*

- Mechanism: Store pointers to TCBs in a FIFO queue.

FCFS Policy Evaluation

Arrival into ready queue means the thread either:
(1) Just got created or
(2) It just finished a previous I/O routine

Job (Arrival Time)	P3 (t=0)	P2 (t=1)	P1 (t=2)
Burst Length	2	3	24

- **[Draw the gantt diagram representing the FCFS schedule]**
- Evaluate the policy:
  - Assume context-switches and process creation/termination overheads are zero cost.
  - What is the *average wait time* and *average turnaround time* of this schedule?

FCFS: Arrival Order Matters!

(CPU)

"LOL slow pokes.
I'll have a *Hello Kitty Macchiato*!"

Adam (long job)

Brad (short job)

David (short job)

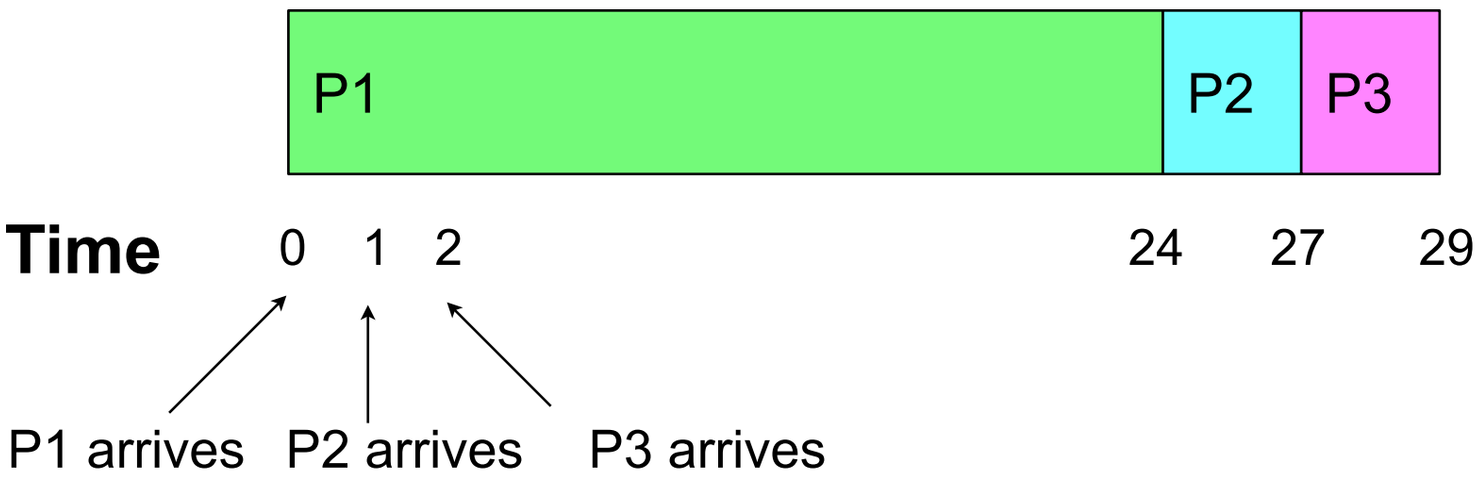
Job (Arrival Time)	P1 (t=0)	P2 (t=1)	P3 (t=2)
Burst Length	24	3	2

FCFS Schedule

Time 0 1 2 ... 24 27 29

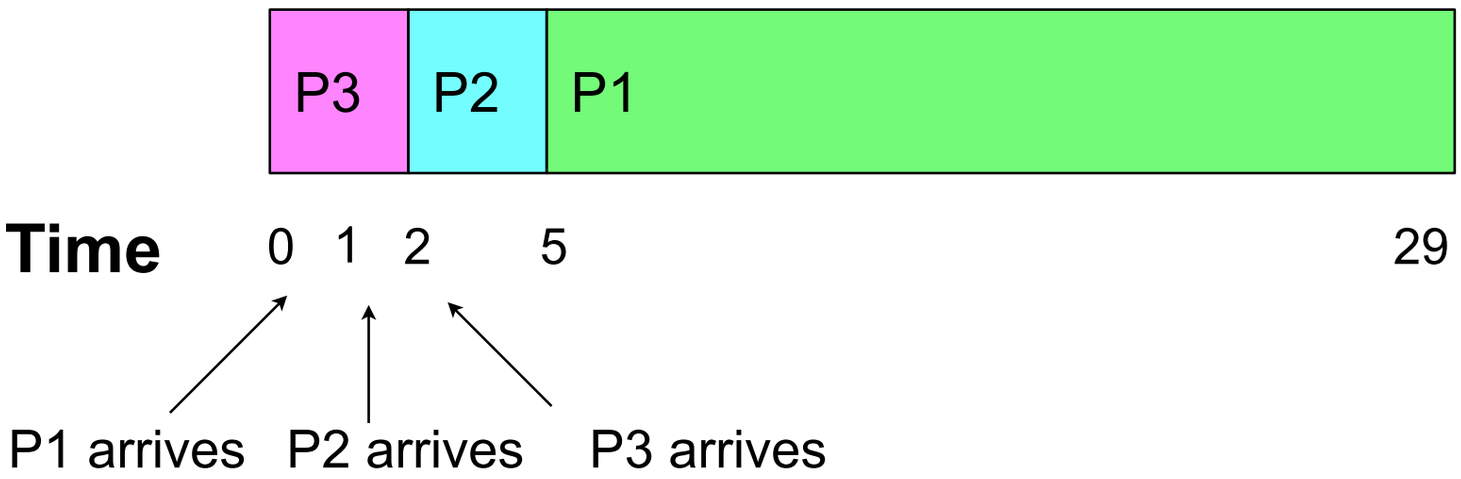
FSCS Lesson-Learned!

- **Average wait time** matters! The goal should be to keep users happy



Turnaround Times

P1: 24 sec
P2: 26 sec
P3: 27 sec



Long CPU bound jobs have a lot less to lose!

P1: 29 sec
P2: 5 sec
P3: 2 sec

Evaluation of FCFS

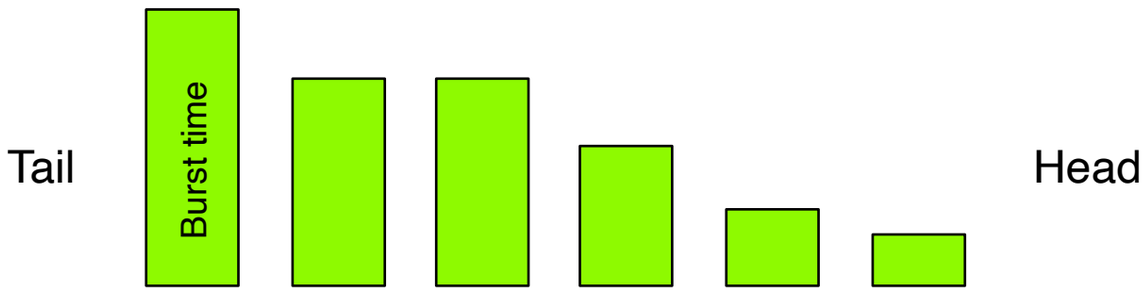
- Category: Non-preemptive (Cooperative)
- Metrics:
  - Wait Time: Depends on arrival order
  - Fairness: Well, FCFS *seems* fair... but is it?
    - Long bursts vs. short bursts?
  - Starvation: Can a job wait indefinitely?
    - Optimistically, if everyone writes good code, no.
  - Scheduling overhead: Minimal
    - Assumes ready queue is a linked list implementation
    - Enqueue/dequeue: O(1)

FCFS Scheduling Summary

- Pros
  - Simple to implement (FIFO Queue)
  - Fair in the sense that every process is treated equally.
- Cons
  - No control over arrival order
  - Arrival order heavily affects wait time and completion time
  - Fairness? Long CPU-bound jobs can monopolize the CPU
    - FCFS punishes **short** and **I/O bound jobs**
    - Imagine **dsh** shell:
      - After every keystroke, you go to the back of the line because it's an I/O!

Shortest Job First (SJF) Policy

- Keep people happy. Minimize wait (response) times.
  - *Prioritize* short and I/O-intensive jobs



Queue (Requires Crystal Ball because we don't actually know burst times)

Shortest Job First Policy

1. Run the thread that has the shortest expected CPU burst.
2. Run job non-preemptively, i.e.,

   Until termination, self-yield, or I/O
3. What if two jobs have the same expected burst duration?

   Use FCFS to break tie

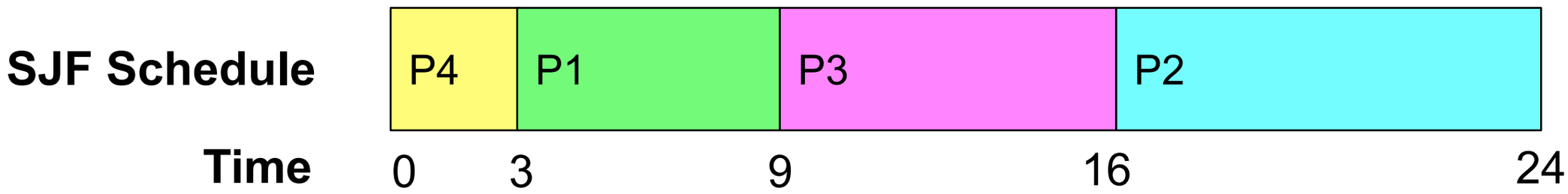
SJF Example

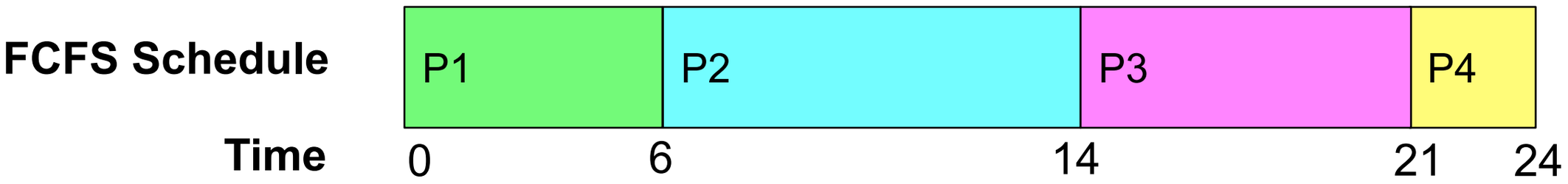
Job (Arrival Time)	P1 (t=0)	P2 (t=0)	P3 (t=0)	P4 (t=0)
Burst Length	6	8	7	3

- Suppose four threads arrive at approximately time t = 0.
- Produce the schedule for SJF and FCFS
  - What is the average wait time of each policy?
  - What is the average turnaround time?

SJF Example (2)

Job (Arrival Time)	P1 (t=0)	P2 (t=0)	P3 (t=0)	P4 (t=0)
Burst Length	6	8	7	3





- SJF
  - Average Wait Time = 7
  - Average Turnaround Time = 13
- FCFS

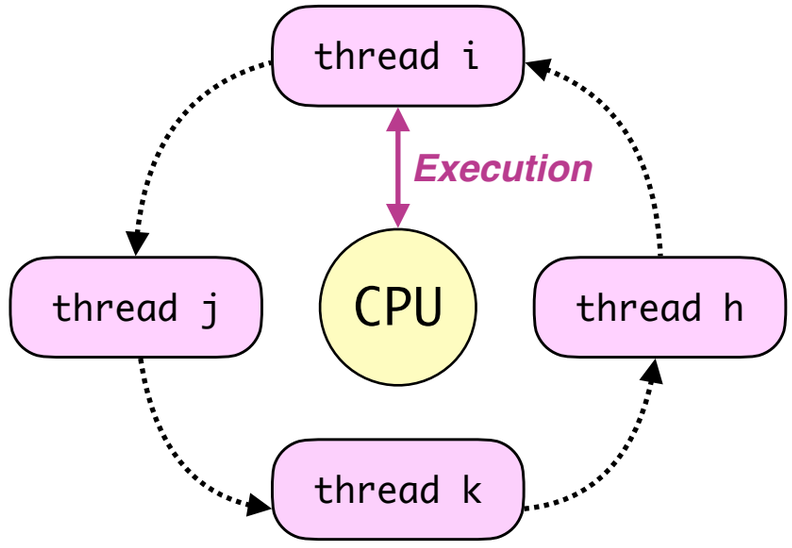
  Arrival order: P1, P2, P3, P4
  - Average Wait Time = 10.25
  - Average Turnaround Time = 16.25

Evaluation of SJF (Non-Preemptive)

- Wait Time: SJF is a *provably optimal* non-preemptive policy for minimizing wait time
- Fairness: Winners and losers?
  - Rewards IO-intensive and short jobs
  - Penalizes CPU-intensive jobs
- Starvation:
  - Possible for long bursts, if short bursts keep arriving in the meantime
- Scheduling overhead:
  - High! Predicting burst times; re-organizing the queue
- **Exists only in theory:**
  - Need a crystal ball to predict burst times

Round Robin Scheduling (RR)

- We don't have a crystal ball... another way to combat a bad arrival order?



Round Robin (RR) Policy

1. Dequeue a job from head of ready queue
2. Run that job for *at most* one time slice *T*

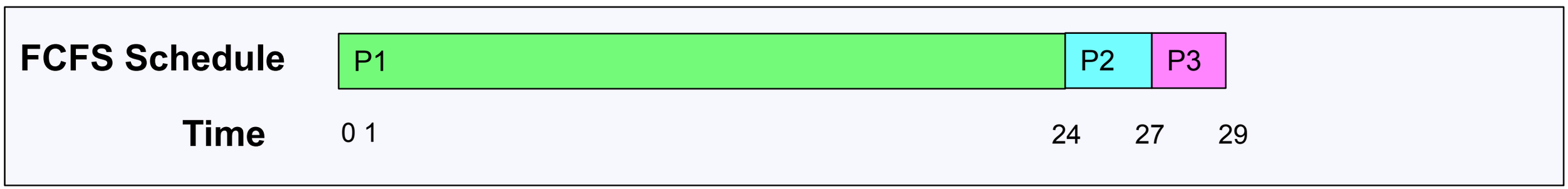
   If job hasn't completed: preempt it and add it to the tail of ready queue
3. If job terminates, self-yields, or blocks before *T* is up,

   Go back to step 1.

Remember When This Blew up FCFS?

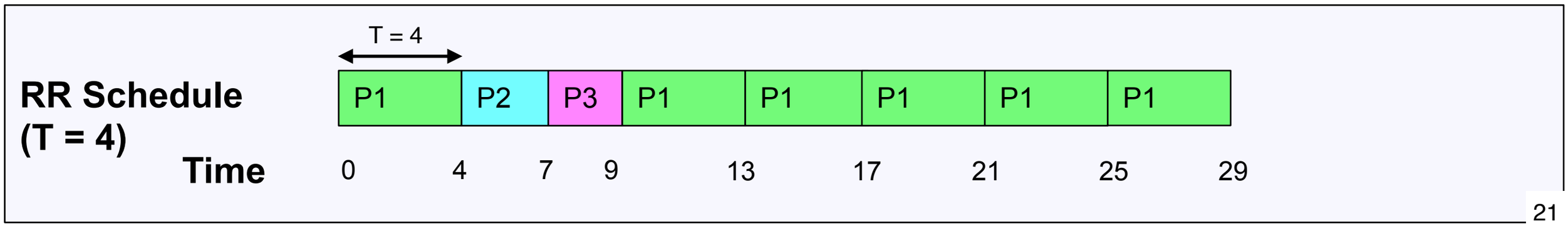
Job (Arrival Time)	P1 (t=0)	P2 (t=1)	P3 (t=2)
Burst Length	24	3	2

Wait Time
P1: 0
P2: 24 - 1 = 23
P3: 27 - 2 = 25
Avg = 16



Arrival order does not significantly impact RR!

Wait Time
P1: 0 + (9 - 4) = 5
P2: 4 - 1 = 3
P3: 7 - 2 = 5
Avg = 4.333



Round Robin (RR) Implementation

- Mechanism for enabling RR policy?
  - A simple FIFO ready queue just like FCFS
    - Here's a great example of why we separate policy from mechanism.
    - The old FCFS mechanism can be used to support a different policy.
  - Plus, a hardware timer that interrupts CPU at a periodic interval
    - On timer interrupt, the interrupt handler routine simply:
      - Dequeues next thread from ready queue
      - Calls contextSwitch()

RR Is Not *Purely* Preemptive by Definition

Run RR scheduler when....

(1) a thread blocks on I/O

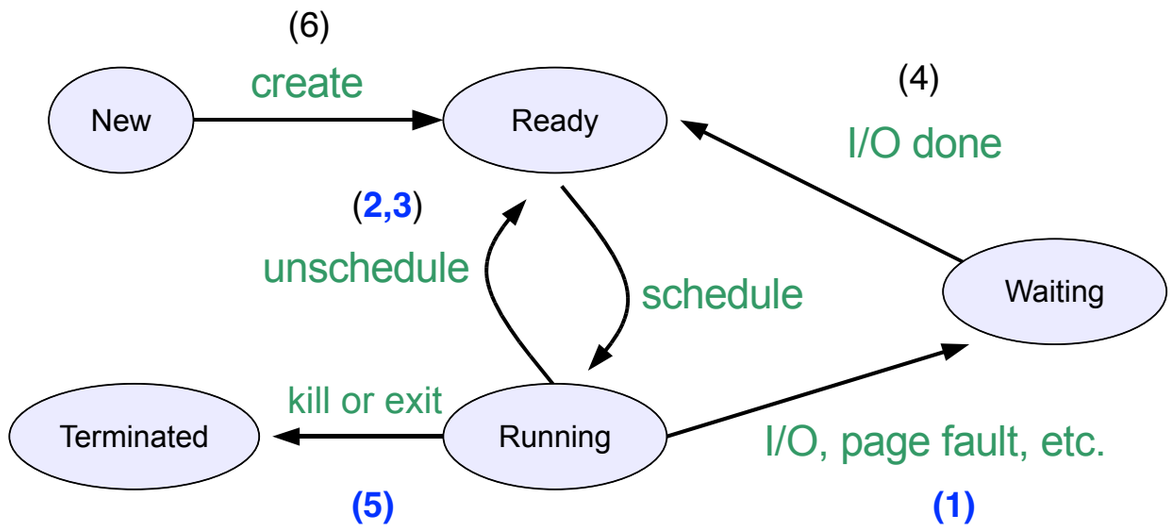
(2) **a thread is interrupted**

(3) the running thread *yields the CPU*

(4) ~~I/O completes for a thread~~

(5) a thread terminates

(6) ~~a thread is created~~



RR: Thoughts on Quantum Size?

- Important parameter:
  - How do you choose quantum size *T*?

- What if *T* is **too large**?
  - Response time suffers
  - Bad for I/O-intensive jobs; great for CPU-bound jobs
  - What if we assign an infinite time slice ($T = \infty$)?

- What if *T* is **too small**?
  - Great for response (wait) time, but really bad for CPU-intensive jobs
    - Context switching takes time, and doesn't do any "real work!"

Actual Choices of Quantum Size T

- In practice, we need to balance short-job performance and long-job throughput:
  - Typical quantum size in modern OS is between 10ms – 100ms
  - Typical context-switching overhead is 0.1ms – 1ms
    - (Roughly 1% overhead due to context-switching)

- Initially, UNIX quantum size was 1 second
  - Worked fine until more than 2-3 people were logged in
  - Imagine: 3 people logged in
    - 3 processes running `gcc` (CPU intensive)
    - It would take 3 seconds to see each keystroke!

Evaluation of RR

- **Class:** Hybrid (preemptive on timer interrupt; otherwise nonpreemptive)
- **Wait Time (Response Time):**
  - No job waits longer than nT time units to get scheduled
  - So, it depends on quantum size T
- **Fairness:**
  - Fairer than FCFS to I/O-intensive and short jobs
    - But some I/O-intensive jobs still suffer as they did in FCFS *(why?)*
- **Starvation:** Not possible
- **Scheduling overhead:**
  - Can be high depending on choice of T.

Shortest Remaining Time First (SRTF)

- SRTF = SJF + preemption

Shortest Remaining Time Policy (Preemptive)

1. Choose job that has the shortest CPU Burst
2. Run job preemptively, i.e.,
   - Until termination, self yield, or blocks on I/O, or
   - Until a job (re-)enters the ready queue
3. Choose to run another job if it has a smaller burst length than what is remaining on current job
4. Tie breaker: Use FCFS

SRTF Example

Either:
(1) Arriving from creation or
(2) From finishing previous I/O routine

Job (Arrival Time)	P1 (t=11)	P2 (t=0)	P3 (t=4)	P4 (t=2)
Burst Length	6	7	8	4

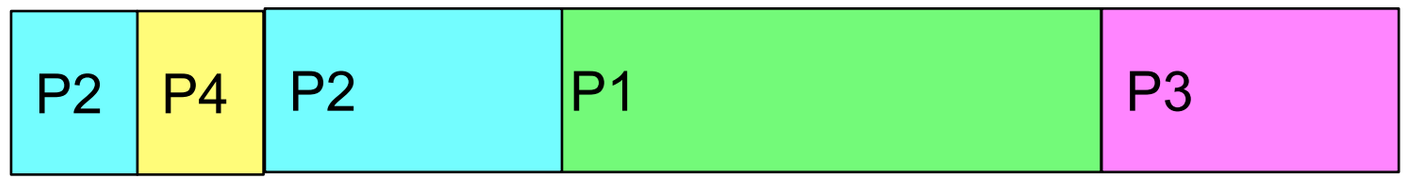
- Produce the Gantt Chart if SRTF policy is in place
  - What is the average wait time?
  - What is the average turnaround time?

SRTF Example

Either:
(1) Arriving from creation or
(2) From finishing previous I/O routine

Job (Arrival Time)	P1 (t=11)	P2 (t=0)	P3 (t=4)	P4 (t=2)
Burst Length	6	7	8	4

SRTF Schedule



Time 0 2 6 11 17 25

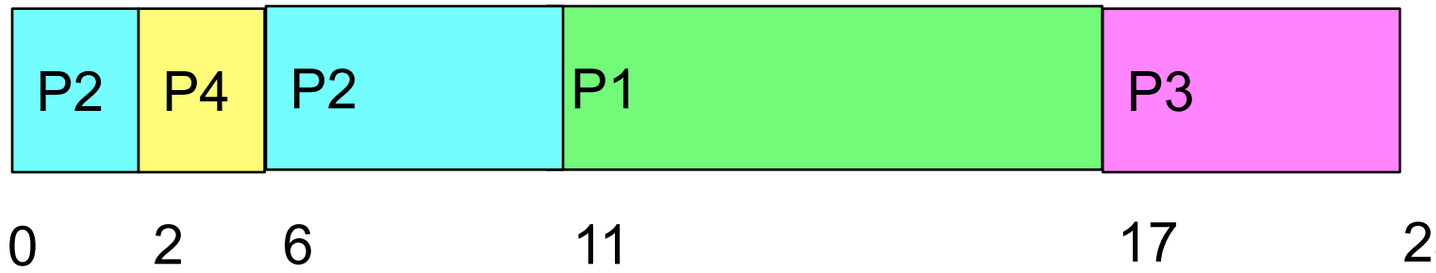
- Average wait time (AWT) = 4.25
- Average turnaround time (ATT) = 10.5

SRTF vs. SJF (Preemption vs. Non)

Either:
(1) Arriving from creation or
(2) From finishing previous I/O routine

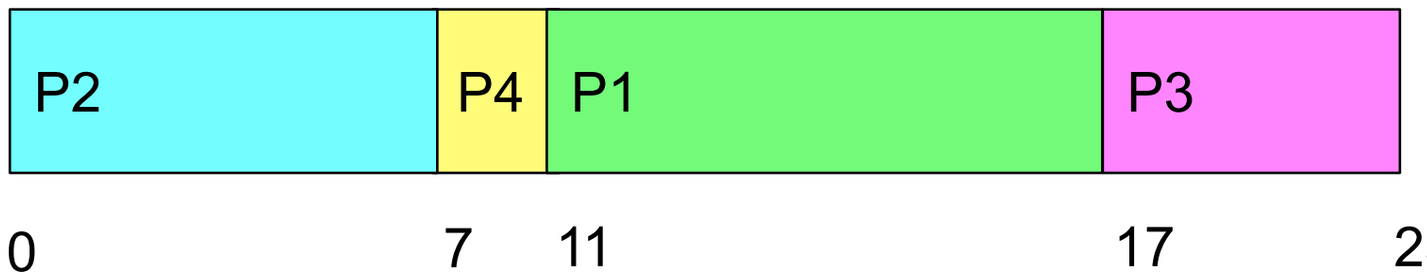
Job (Arrival Time)	P1 (t=11)	P2 (t=0)	P3 (t=4)	P4 (t=2)
Burst Length	6	7	8	4

SRTF Schedule



- AWT = 4.25
- ATT = 10.5

SJF Schedule



- AWT = 4.5
- ATT = 10.75

SRTF Example

- Same example from earlier

- What is the Gantt Chart and average wait time using SRTF?

Job #	Arrival Time	CPU Burst Time
j1	0 ms	25 ms
j2	1 ms	15 ms
j3	20 ms	5 ms
j4	25 ms	3 ms

Solution

- For each policy, what is the **average wait time**?

- You should get:
  - FCFS: 16
  - SJF: 10
  - RR: 18
  - **SRTF: 5.75**

Job #	Arrival Time	CPU Burst Time
j1	0 ms	25 ms
j2	1 ms	15 ms
j3	20 ms	5 ms
j4	25 ms	3 ms

Evaluation of SRTF

- **Class:** Preemptive
- **Response Time:** SRTF is a *provably optimal* preemptive policy
- **Fairness:** Like SJF, more fair to I/O intensive jobs
- **Starvation:** Like SJF, possible if I/O intensive jobs keep arriving
- **Scheduling overhead:**
  - High; prediction is hard.
  - Need to keep track of remaining times of each thread.
- Again, this policy is theoretical.

Goals for This Lecture

- Objectives
  - Characterizing Thread/Process Execution
  - Scheduling Metrics and Goals
- Scheduling Policies
  - Multi-level Queues
- Multiprocessing Considerations
  - Hyperthreading
  - Scheduling on Multicore CPUs
- Conclusion

Priority Scheduling and Multi-Level Queues

- Associate a priority number with each thread.
  - Lower number $\Longrightarrow$ higher priority

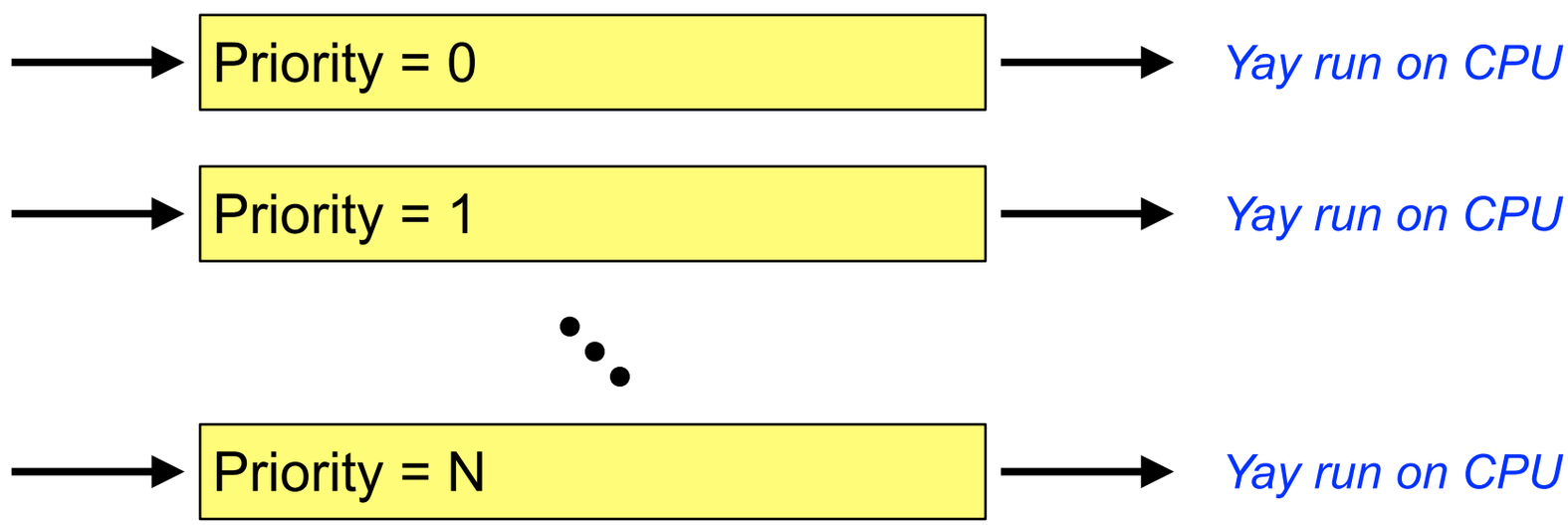
Priority Scheduling Policy

1. Assign each job a priority number
2. Choose job that has the highest priority
3. Run job non-preemptively or preemptively
4. Use FCFS to break ties

- Implement using a Priority Queue (min-heap)
  - Insertion / deletion: $O(\log n)$

Multi-Level Queues

- **Policy:** We want priority scheduling
- **Mechanism:** Multiple Queues
  - Add to each thread control block a priority value
  - On (re)entry to ready queue, put thread in the queue corresponding to its priority
  - Scheduler picks the next thread the from the topmost non-empty queue
    - RR policy is often used as an "overlay" policy



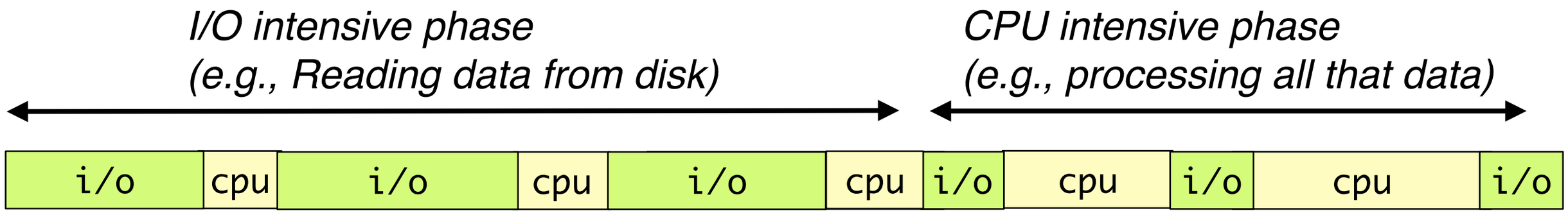
Problem #1: How to Approximate SJF/SRTF?

- How do we approximate SJF/SRTF policies using multilevel queue scheduling?

- Easy, right?
  - Give interactive, I/O-intensive jobs the highest priority
  - Give CPU-intensive jobs the lowest priority
  - *But... we don't have a crystal ball!*

Problem #2: Jobs Can Change Run Profiles

- Jobs may not be *exclusively* CPU- or I/O-intensive.
  - They could oscillate between the two phases.
  - So it's hard to know what priority to give thread from the start!



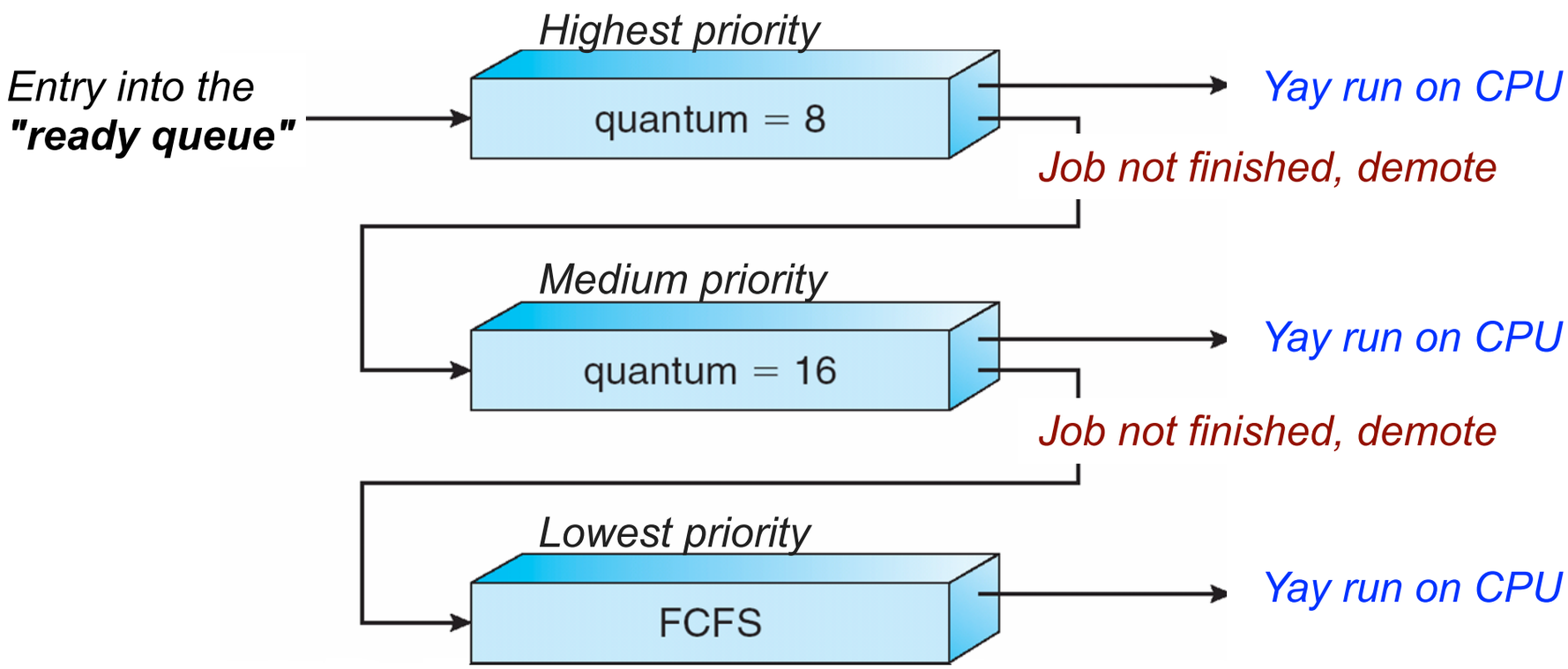
*OS: Want to give you **high** priority here!!*

*OS: Give you **lower** priority here!!*

- If OS gave the above thread **priority = 0**, it will be allowed to hog the CPU when it enters the CPU intensive phase!

Multi-Level Feedback Queues

- Dynamically adjust priority of a job based on its CPU Usage
  - Each queue gets its own scheduling policy.
  - Job always starts in highest priority queue
    - If its CPU burst finishes, it'll be placed in high queue again!
  - Demote job down to next queue if quantum expires
  - Scheduler chooses the job from the top most non-empty queue



Evaluation of MLFQ

- Either
  - Non-preemptive (resulting schedule approximates SJF)
  - Preemptive (resulting schedule approximates STRF)
- Response Time:
  - Approximates SRTF!
- Fairness:
  - Prioritizes I/O intensive jobs!
- Starvation:
  - CPU hogs can starve; Use an aging policy
- Scheduling overhead:
  - $O(1)$ time!

Goals for This Lecture

- Objectives
  - Characterizing Thread/Process Execution
  - Scheduling Metrics and Goals
- Scheduling Policies
  - Real Examples
- Multiprocessing Considerations
  - Hyperthreading
  - Scheduling on Multicore CPUs
- Conclusion

Example: OSX

- Mac OSX uses the CMU Mach 3 Scheduler
  - 4 priority "bands"
  - Allows users to specify getting a certain
    % time slice within a certain period of time
  - For example, *I need 40,000 of the next 1,000,000 cycles*

Example: Mac OS X

- Multi-Level Queue approach:
  - There are priorities within each "band," also,
  - Threads can change priorities, but only within bands

Priority Band	
Kernel Mode Only	Threads created inside the kernel that need to run at a higher priority than users' threads
System High Priority	Users' threads whose priority have been raised above *normal* threads
Real-Time Threads	Threads whose priority is based on getting a fraction of total clock cycles
Normal	User application priority (lowest priority)

Example to Set a Thread to be "Real-Time"

- This only works on a Mac

```
#include <mach/mach_init.h>
#include <mach/thread_policy.h>
#include <mach/sched.h>

int set_realtime(int period, int computation, int constraint) {

    struct thread_time_constraint_policy ttcpolicy;
    int ret;

    ttcpolicy.period = period; // HZ/160
    ttcpolicy.computation = computation; // HZ/3300
    ttcpolicy.constraint = constraint; // HZ/2200
    ttcpolicy.preemptible = 1;

    if ((ret=thread_policy_set(mach_thread_self(),
          THREAD_TIME_CONSTRAINT_POLICY,
          (thread_policy_t) &ttcpolicy,
          THREAD_TIME_CONSTRAINT_POLICY_COUNT)) != KERN_SUCCESS) {
            fprintf(stderr, "set_realtime() failed.\n");
            return 0;
    }
    return 1;
}
```

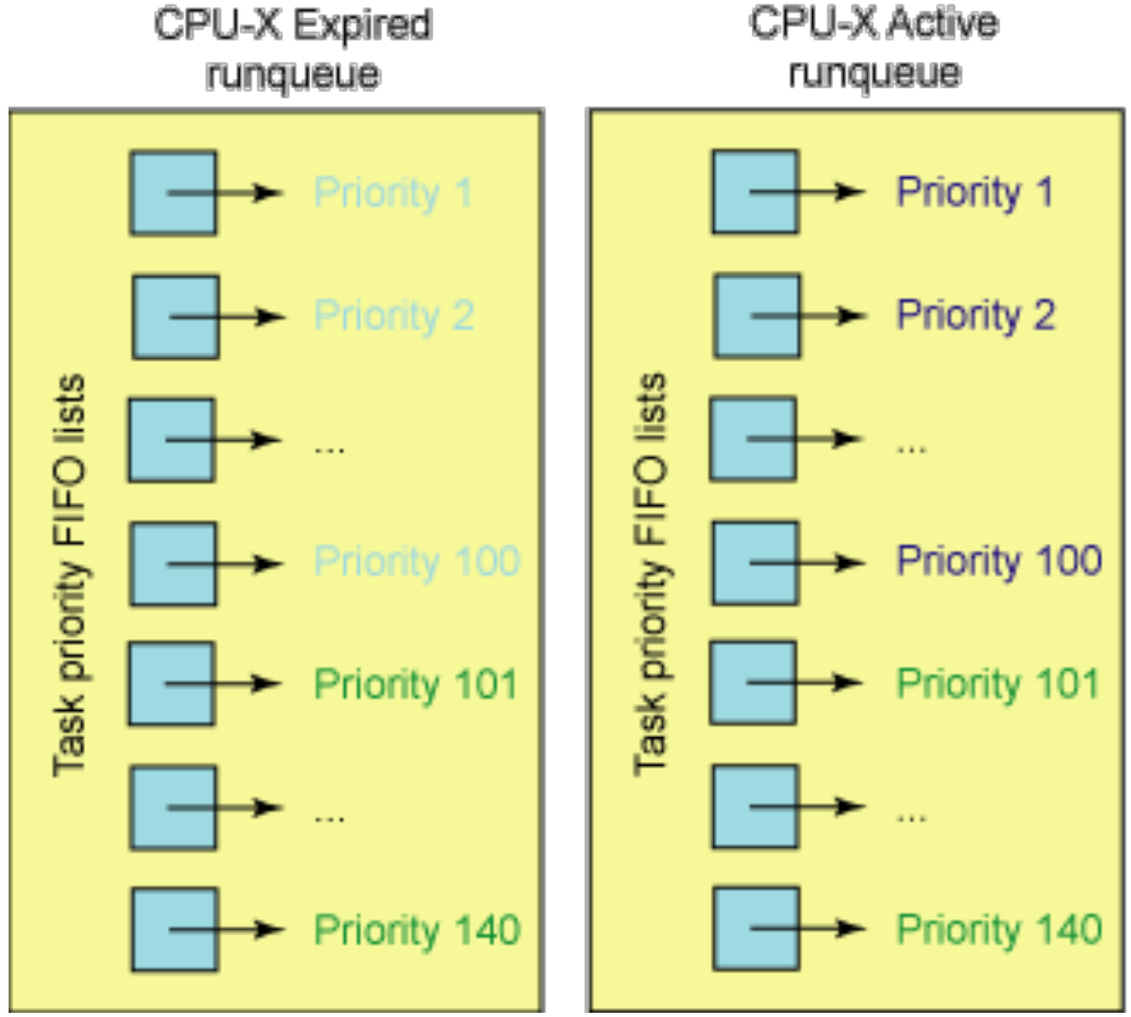
Example: Mac OS X (Cont.)

- Real-Time Threads are penalized if it repeatedly does not block on I/O under the fixed time-slice
  - Threads scheduled under the *Real-Time* band, but turns out to be CPU-bound are demoted to *Normal*

- Threads that are CPU-bound receive lowest priority

Example: Linux O(1) Scheduler (2.5 - 2.6.23)

- Two separate run-queues: *Active* and *Expired*
- Each "run-queue" has 140 separate FCFS queues, each with a priority value (Low value = higher priority)



Linux 0(1) Scheduler (No Longer in Use)

Linux O(1) Scheduler (priority-based variant of RR)

1. Find highest priority *active run-queue* with a process in its list
2. Dequeue a job from that list
3. Run the job preemptively
4. When quantum expires, and job has not finished, place job on same level in the *expired run-queue*

Goals for This Lecture

- Objectives
  - Characterizing Thread/Process Execution
  - Scheduling Metrics and Goals
- Scheduling Policies
  - Preemptive
  - Non-preemptive
  - Evaluation
- Multiprocessing Considerations
  - Hyperthreading
  - Scheduling on Multicore CPUs
- Conclusion

"Hardware" Threads

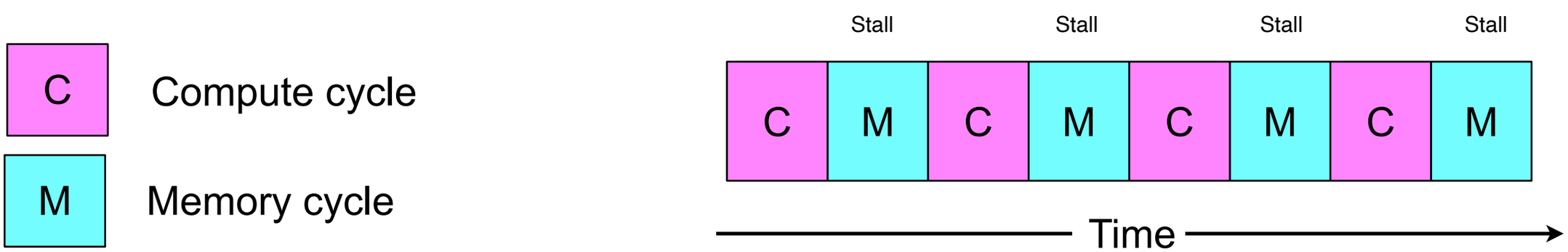
- Software thread vs hardware thread

- We've assumed that a CPU Core is only capable of running one software thread at a time:
  - Each CPU Core has:
    - Special registers: 1 PC register + 1 SP register + 1 BP register + ...
    - 1 set of general-purpose registers
  - We say that such a CPU core has only 1 "hardware thread"

CPU-Memory Stalls

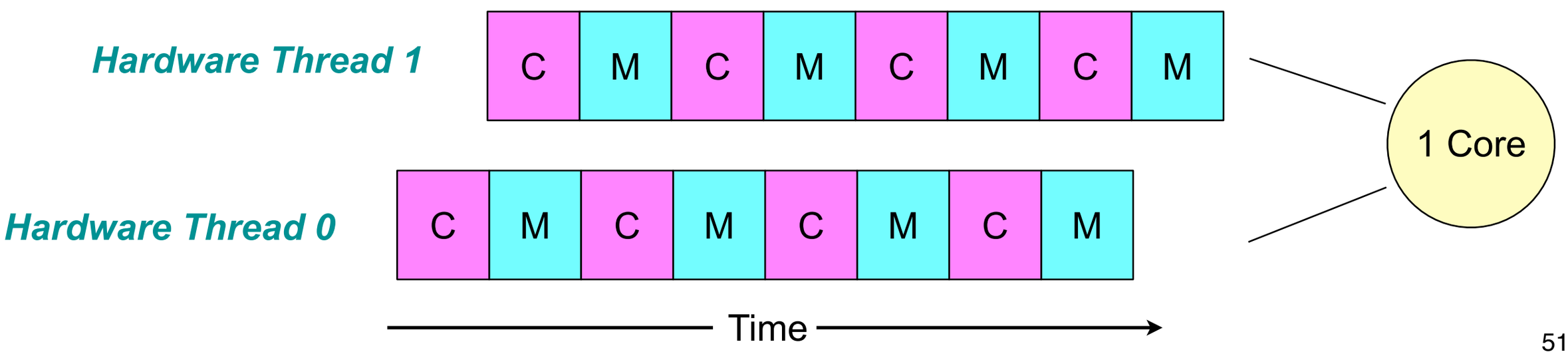
- At the CPU level, thread execution alternates between computation and memory.

- **Problem:** Memory cycles are slow and can stall execution
  - If a thread is waiting on a memory cycle, no work can proceed until data is written back into register (a memory cycle costs ~200 CPU cycles)
  - Not slow enough to warrant a context switch (which costs 10,000s of cycles), but these *memory stalls* add up!



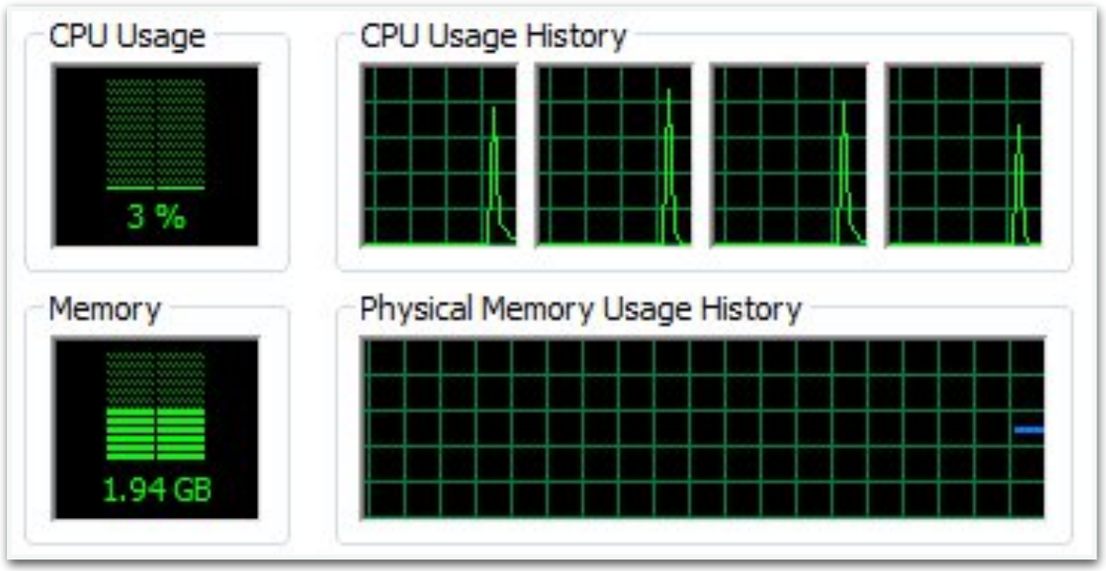
Hyperthreading (Cont.)

- *Hyperthreading (logical processors)*
  - Pre-dates multicore CPUs
  - **Duplicate** some CPU hardware
    - Special registers and general purpose ones too
  - Allows two "hardware threads" to run simultaneously on one core
  - To the OS (and users) each core appears as two CPUs!



What Does the OS See?

- Hyperthreading exposes two hardware threads per core to the OS
  - Each hardware thread is also called "Logical CPUs"
  - Lower left figure: A dual-core CPU with hyperthreading enabled shows as 4 cores to the OS



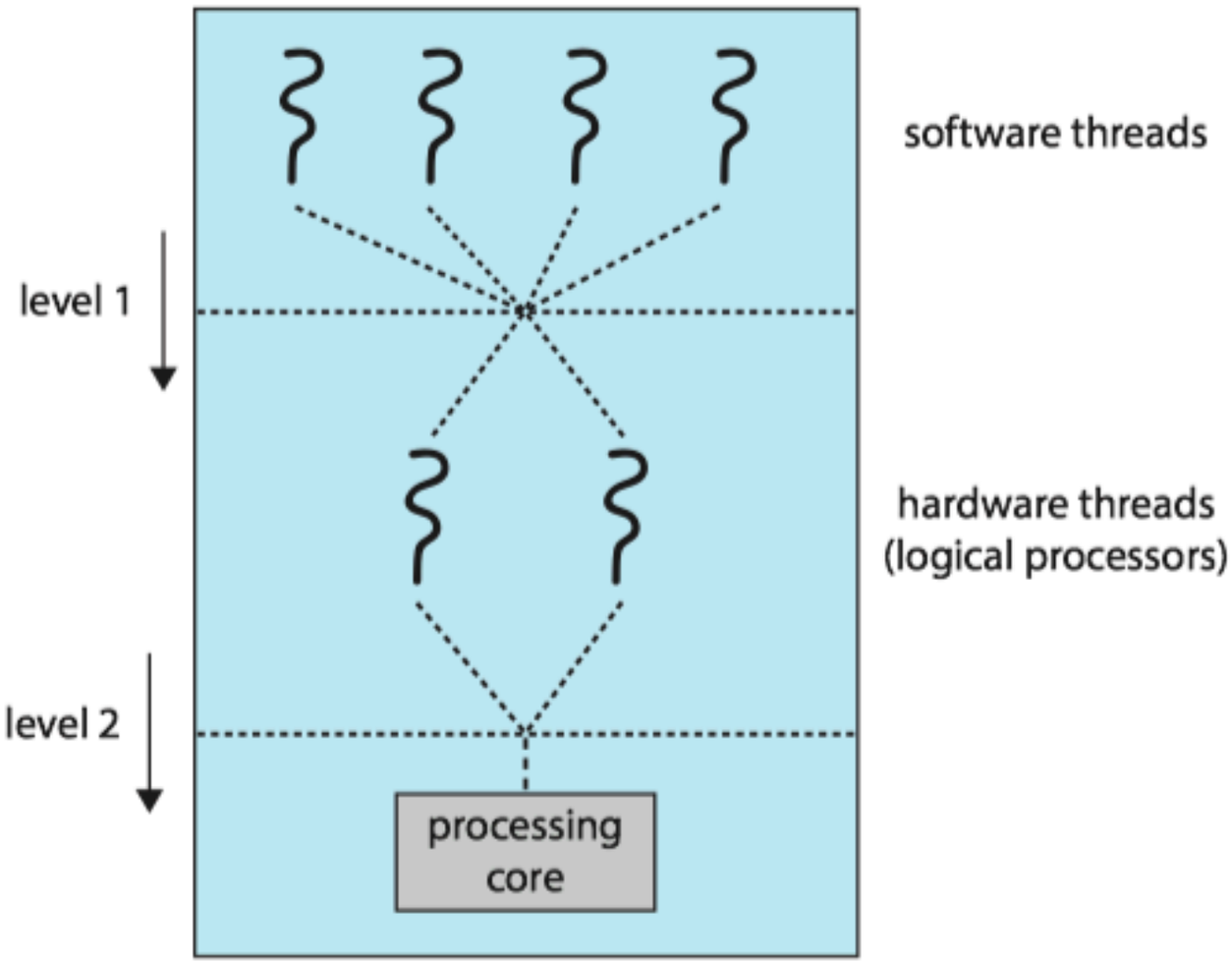
From my MacOS command line:

```
dsh> sysctl hw.physicalcpu hw.logicalcpu
hw.physicalcpu: 8
hw.logicalcpu: 16
dsh>
```

(I run an Intel processor with 8 cores)

2-level Scheduling on Hyperthreaded CPUs

- There are now two levels of scheduling:
  - **OS level:** Which software thread to run now?
  - **CPU level:** Which hardware thread to run on?
    - RR suffices here.

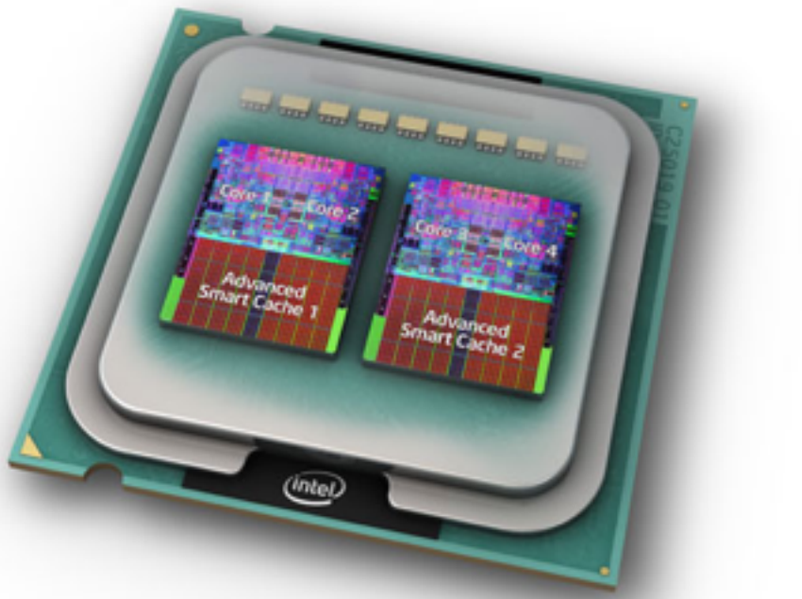


Goals for This Lecture

- Objectives
  - Characterizing Thread/Process Execution
  - Scheduling Metrics and Goals
- Scheduling Policies
  - Preemptive
  - Non-preemptive
  - Evaluation
- Multiprocessing Considerations
  - Hyperthreading
  - Scheduling on Multicore CPUs
- Conclusion

Multiprocessors vs. Multicore CPUs

- Multiprocessors (2+ standalone CPUs) vs. Multicore CPUs:
  - Major architectural differences
    - 1 multicore CPU uses less power than multiple standalone CPU
    - Each "core" on a multicore chip is weaker than a standalone CPU
    - On-chip communication is faster than inter-chip communication
    - Multicore CPU have *shared* caches; Multiprocessors have independent caches

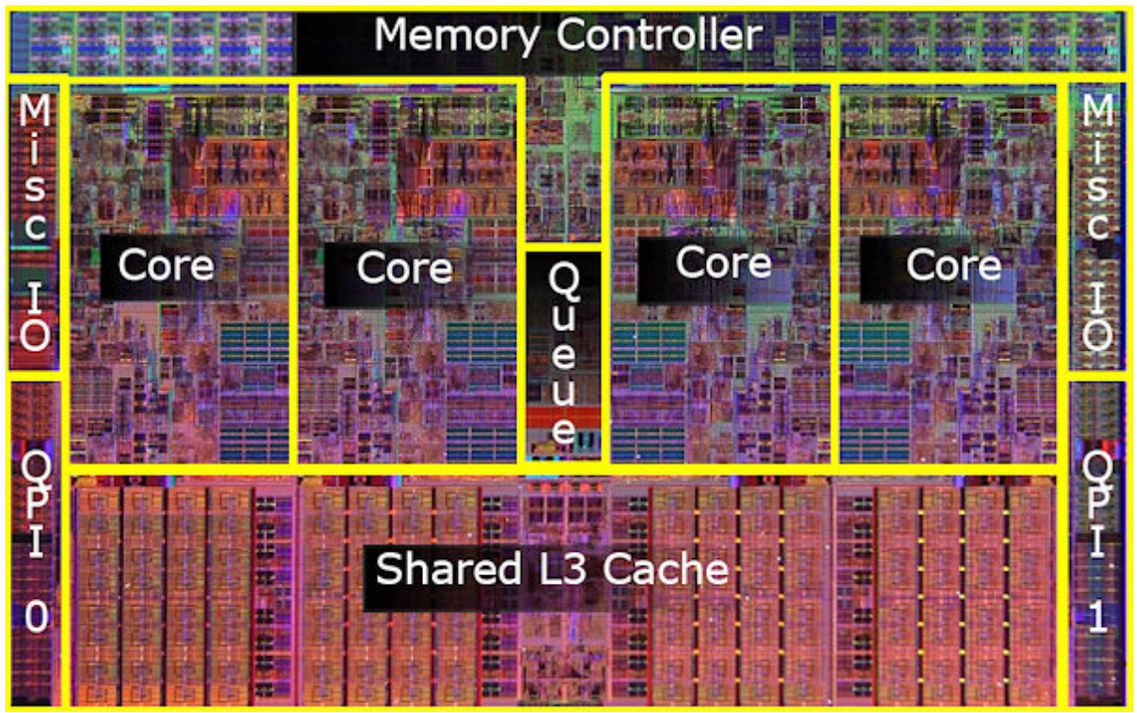


Architecture #1: SMP

- *Symmetric Multiprocessing (or SMP)*
  - Cores are homogeneous. Each core is self-scheduling.
  - All jobs may be in a common ready queue
    - Or we could have one ready queue per core
  - Most OS's today support SMP (e.g., pthreads)

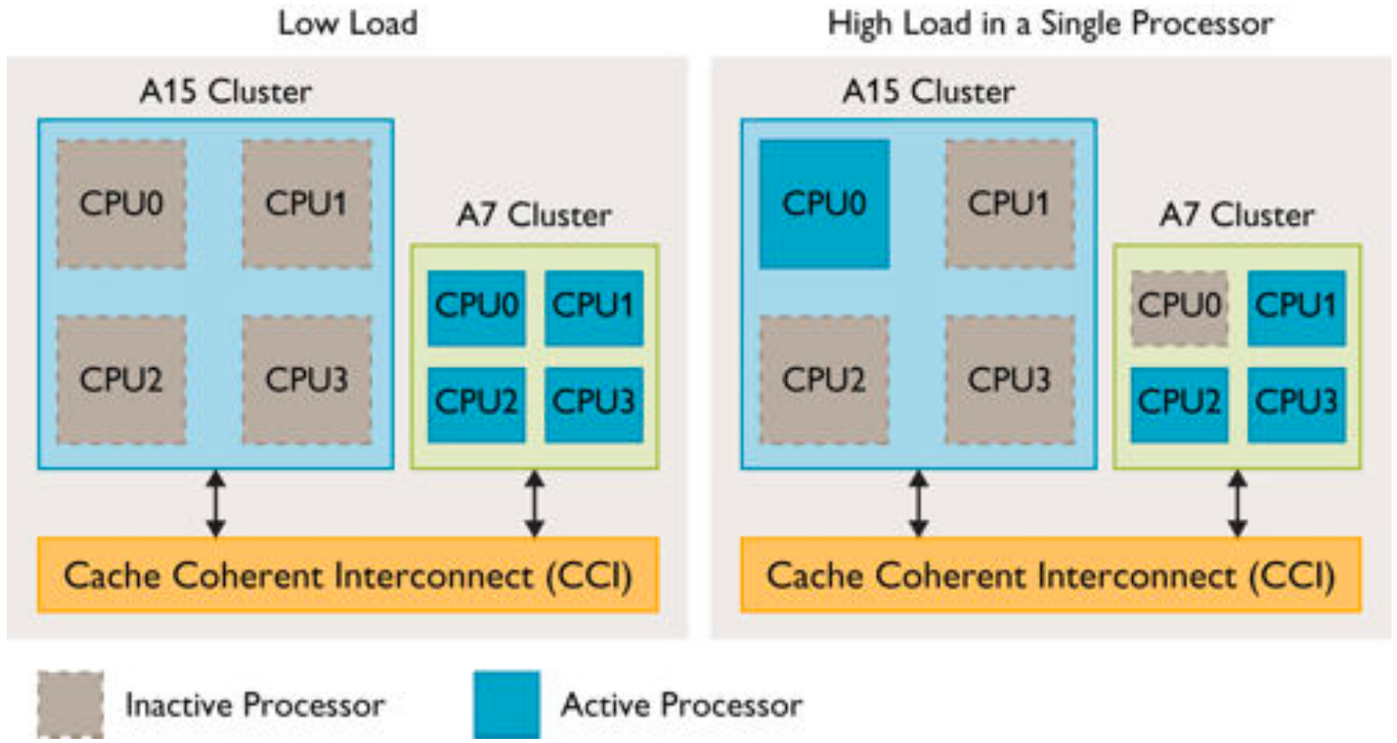
- **What's the tradeoff between:**
  One common ready queue vs.
  one queue per core?

Intel Core i7



Architecture #2: AMP (or HMP)

- *Asymmetric Multiprocessing (or Heterogeneous Multiprocessing)*
  - Designate a head (**brawny**) core to run the OS
    - Unlike SMP, the brawny core dispatches jobs to puny cores
  - Worker (**puny**) cores' only task is to execute user jobs
  - Found in niche & mobile devices, due to energy savings



ARM big.LITTLE

How to Assign Jobs to CPU Cores?

- *Static Job Assignment*
  - Once a thread is assigned to a core, it stays there until it terminates
    - Implementation: Simple, just have one ready queue for each CPU core
  - Pros:
    - Less overhead: no need to determine which core to run a thread once the initial decision is made.
    - Exploits ***CPU Affinity*** *(???)*
  - Cons:
    - ***Load balancing*** is a new problem
    - 1 core may be idle when another has a long ready queue

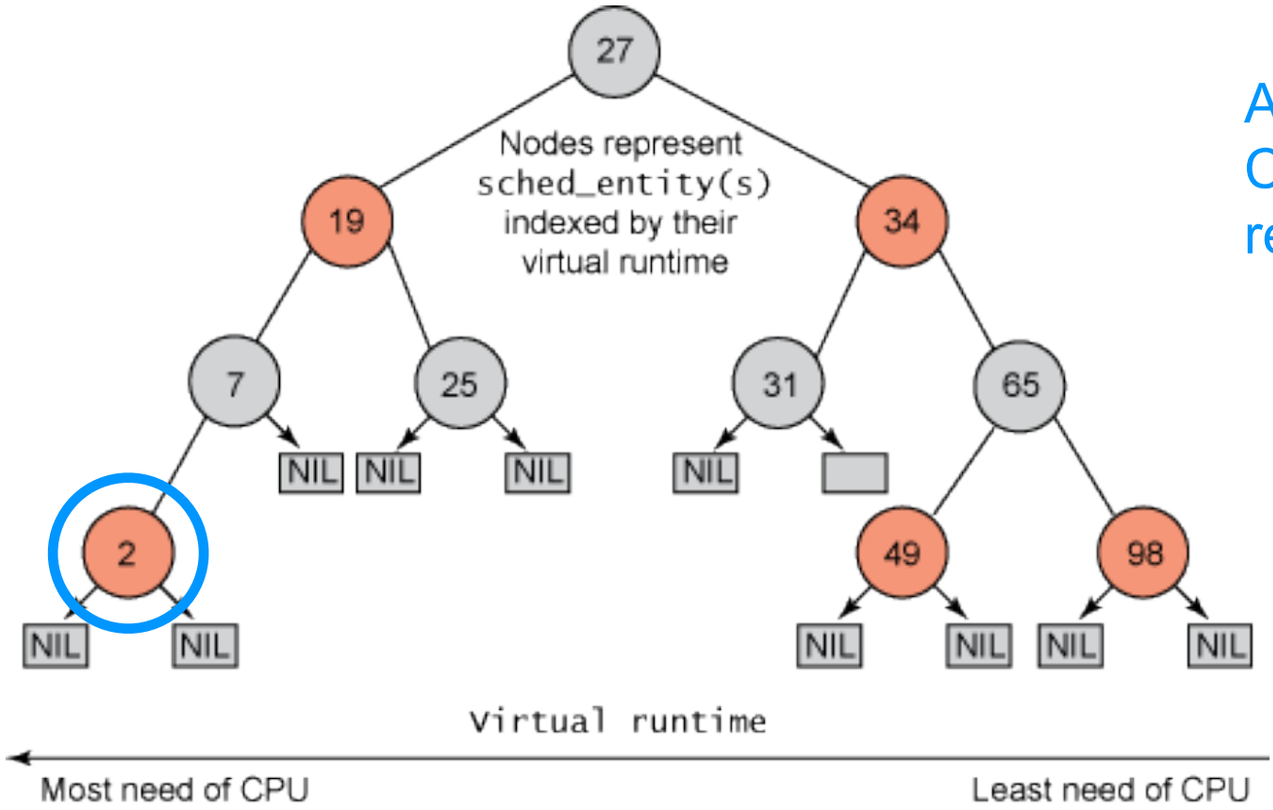
How to Assign Jobs to CPU Cores?

- *Dynamic Job Assignment (More common)*
  - Always schedule next thread on a free core
    - Can either use 1 shared ready queue, or multiple ready queues
  - Pros:
    - Can support *Dynamic Load Balancing*
      - **Policy**: Want all cores to do equal share of work
      - **Mechanism**: work-sharing and work-stealing
        - » Depending on queueing arrangement
        - » Work-sharing used when there's a single common ready queue
        - » Work-stealing used when each core has a ready queue
  - Cons:
    - No guarantees of CPU affinity

Example: Current Linux Kernels Use CFS

- Idea: Give all processes a fair share of the CPU
- *Completely Fair Scheduler (CFS)*
  - Use a Red-Black Tree
    - Each node represents a PCB/TCB
    - Nodes are keyed on **CPU time received**

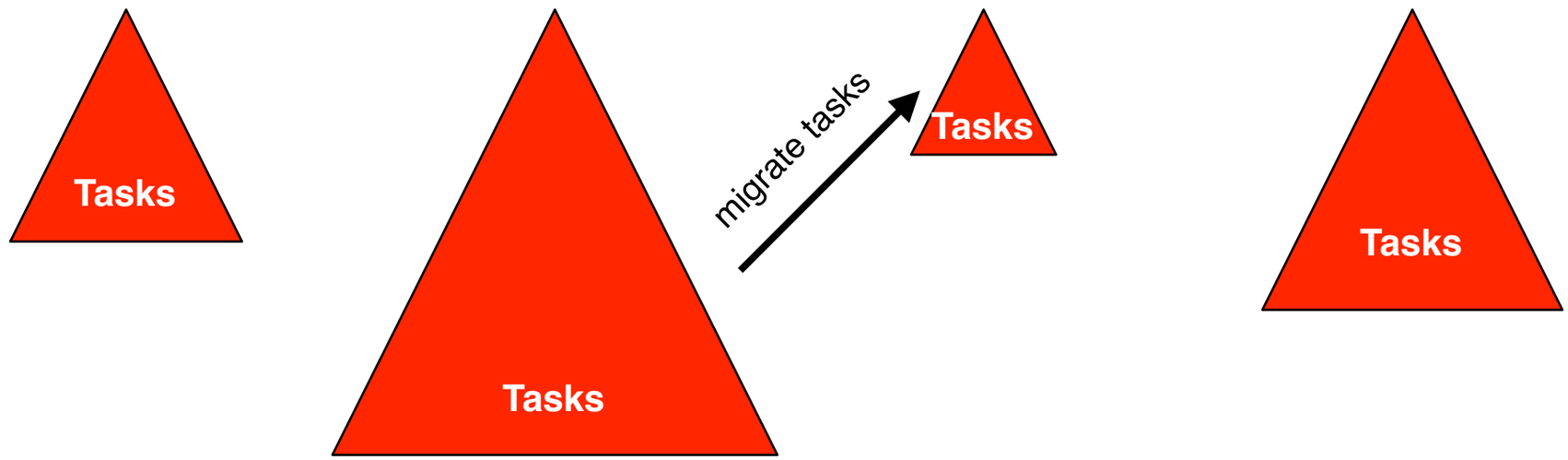
For CPU scheduling (choose left-most node)



After scheduling, add CPU time received, reinsert.

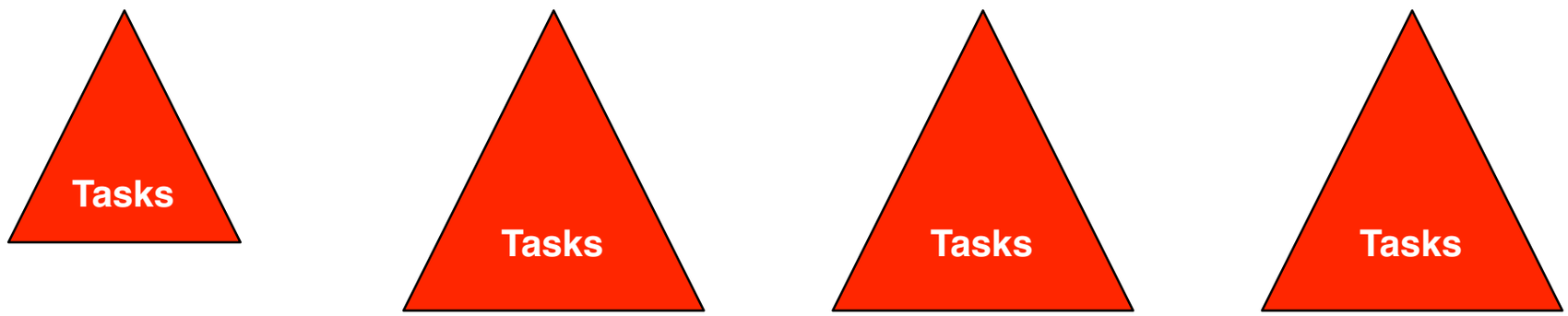
The Linux CFS Uses Work-Stealing

- OS assigns each core its own queue (which is a red-black tree)
  - Periodically check each queue to find the busiest and the lightest

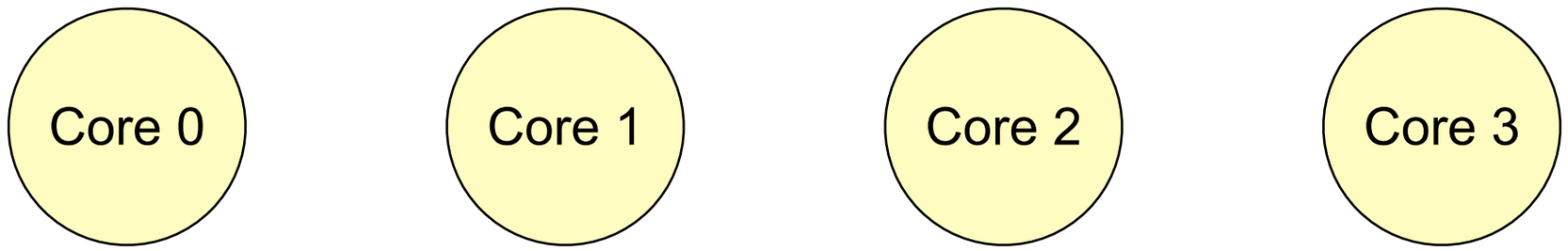


The Linux CFS Uses Work-Stealing (2)

- Kernel assigns each core its own queue (which is a red-black tree)
  - Periodically check each queue to find the busiest and the lightest



What about CPU affinity of threads!?



Pinning: CPU-Affinity Mechanism in Linux

- The Linux TCB has a `cpus_allowed` field, which is a **bit-mask:**
  - Each CPU core denoted by a "bit"
  - A quad-core processor would have a 4-bit sequence `b3 b2 b1 b0`
    - `bi = 0` means task cannot be scheduled on core `i`
    - `bi = 1` means task can be scheduled on core `i`
      - Example: `cpus_allowed = 0xF;  // 0xF (hex) is 1111 (binary)`
        means all four cores are schedule-able for the task

- So, for performance-critical tasks just *pin* them on a specific core, or confine scheduling to an adjacent core (one that shares L2)

Pinning Example: CPU Affinity in Linux (Cont.)

- We can set the CPU affinity of a thread when we create it!

```c
int main(int argc, char *argv[]) {
    cpu_set_t cpus;
    pthread_t thread;
    pthread_attr_t attr;
    pthread_attr_init(&attr);

    //only allow thread to run on 3rd and 4th core
    CPU_ZERO(&cpus);
    CPU_SET(2, &cpus);
    CPU_SET(3, &cpus);
    pthread_attr_setaffinity_np(&attr, sizeof(cpu_set_t), &cpus);

    /* create and dispatch the thread! */
    pthread_create(&thread, &attr, ..., ...);
}
```

Goals for This Lecture

- Objectives
  - Characterizing Thread/Process Execution
  - Scheduling Metrics and Goals
- Scheduling Policies
  - Preemptive
  - Non-preemptive
  - Evaluation
- Multiprocessing Considerations
  - Hyperthreading
  - Scheduling on Multicore CPUs
- Conclusion

In Conclusion...

- CPU scheduling is imperative for multiprogramming and interactive/ real-time systems

- Each scheduling policy has pros and cons
  - There isn't really a perfect scheme
    - *(unless we can read into the future*)
  - But OS can try its best to adapt or approximate the ideal environment

Next Time...

- We've made sure threads/processes can coexist

- They need to communicate to cooperate
  - Read/Write each other's variables
  - How do we make sure our programs will execute **correctly** in such an environment?

- The CPU scheduler makes unpredictable choices that can lead to unexpected execution...
  - *Next Time: Synchronization*

Administrivia 2/28

- Announcements:
  - Hwk 5 due next Friday 3/8
  - Exam 1 progress: 42% done. Maybe done by Friday/Weekend?

- New topic today:
  - Start Chapter 5: CPU Scheduling
  - Characterizing threads as alternating I/O and CPU "bursts"
  - Preemptive vs. non-preemptive scheduling
  - The first scheduling policy: FCFS