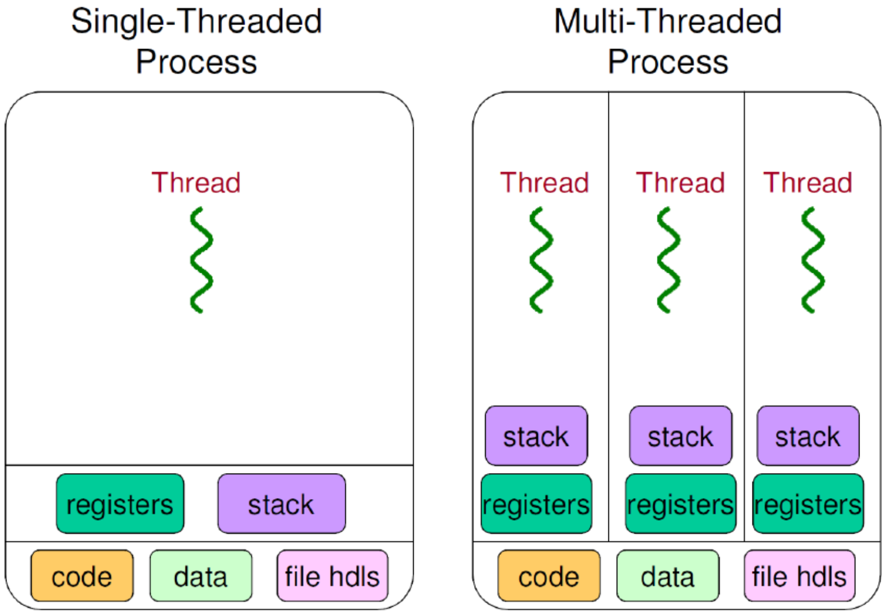
Scheduling

CS 202: Advanced Operating Systems

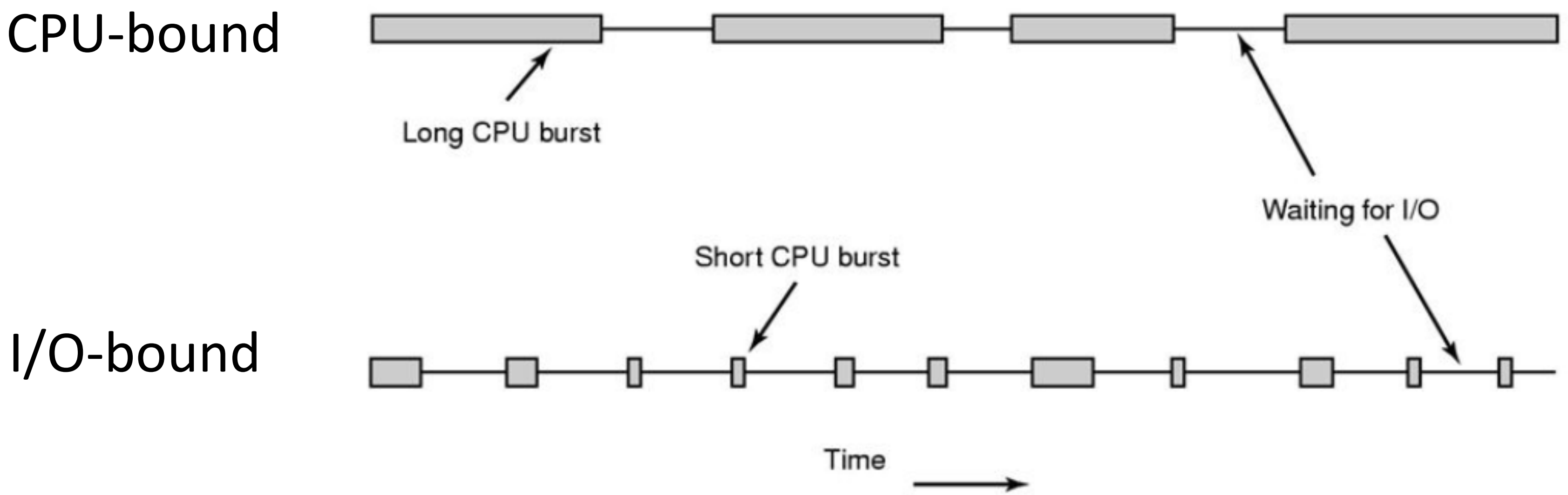
Processes & Threads

- Process is an OS abstraction for execution
- OS with multi-threading support:
  - Thread is the unit of scheduling
  - Processes are the containers in which threads execute
- Often referred to as *tasks*



CPU Bound and I/O Bound

- Processes often have an alternating sequence of CPU and I/O bursts
  - Each cycle consist of a CPU burst followed by a (usually longer) I/O burst
  - CPU-bound processes have longer CPU bursts than I/O-bound



CPU Scheduling

- Scheduler runs when we context switch among processes/threads on the ready queue
  - What should it do? Does it matter?

- Making the decision on what thread to run is called scheduling
  - What are the goals of scheduling?
  - What are common scheduling algorithms?

Scheduling

- Right from the start of multiprogramming, scheduling was identified as a big issue
- Scheduling is a form of resource allocation
  - CPU is the resource
  - Resource allocation needed for other resources too; sometimes similar algorithms apply
- Requires ***mechanisms*** and ***policy***
  - Mechanisms: Context switching, timers, process queues, process state information, …
  - Policies: i.e., when to switch and which process/thread to run next

Characterization of Scheduling Policies

- Selection function (which)
  - Determines which process in the ready queue (also called run queue) is selected for execution
- Decision mode (when)
  - Specifies **when** the selection is made
  - Nonpreemptive
    - Once a process is in the running state, it will continue until it terminates or blocks itself for I/O
  - Preemptive
    - Currently running process may be interrupted (time quantum expired) and moved to the ready state by the OS
    - Allows for better service since any one process cannot monopolize the processor for very long

Preemptive vs. Non-preemptive

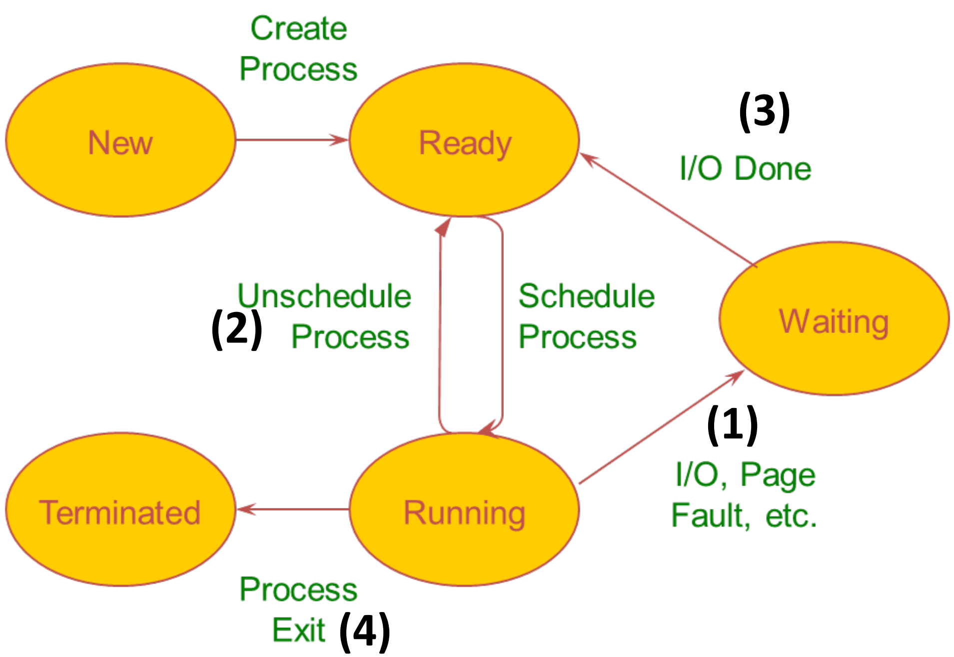
- In *preemptive* systems, we can interrupt a running job (involuntary context switch)

- In *non-preemptive* systems, the scheduler waits for a running job to give up CPU (voluntary context switch)
  - Was it only interesting in the days of batch multiprogramming?
  - Some systems continue to use cooperative scheduling
    - What are such systems?

- Example scheduling algorithms:
  - RR, FCFS, Shortest Job First, Priority Scheduling, ...

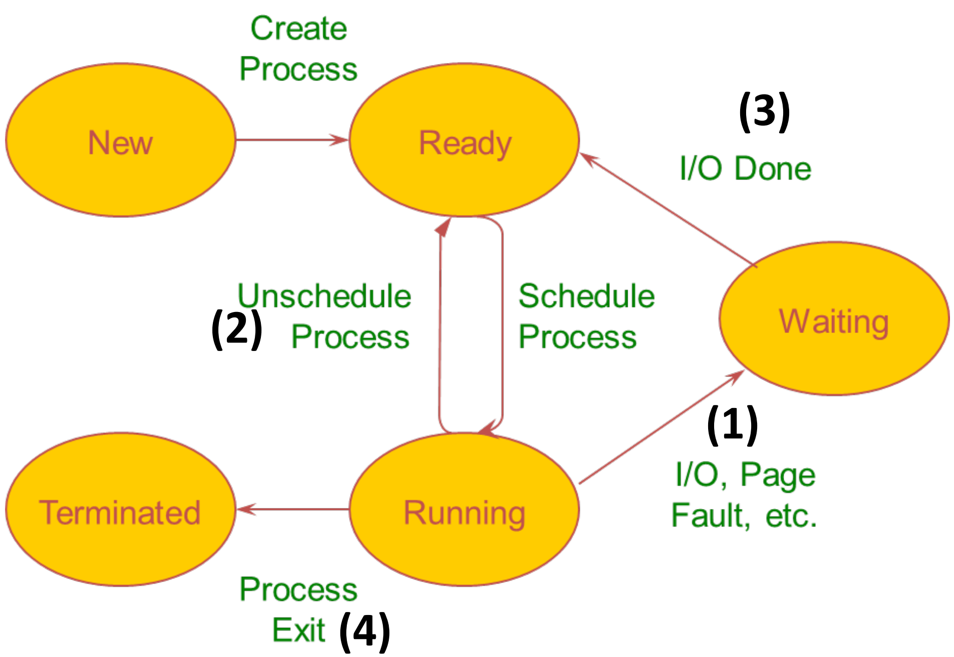
Decision Mode

- CPU scheduling decisions may take place when a process
  - (1) Switches from running to waiting (blocked) state
  - (2) Switches from running to ready state
  - (3) Switches from waiting (blocked) to ready
  - (4) Terminates (exits)



Decision Mode

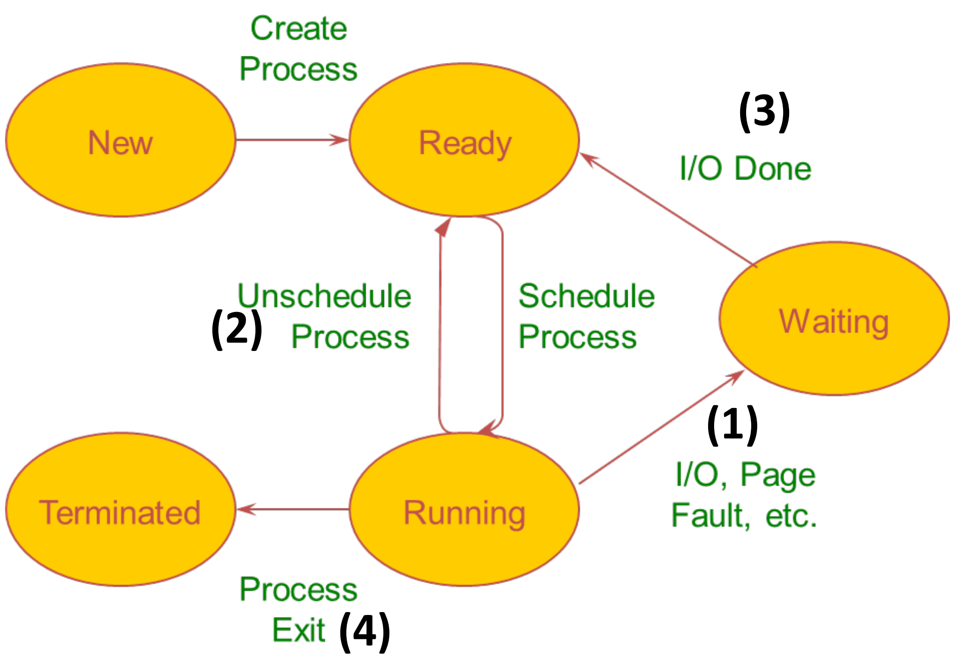
- CPU scheduling decisions may take place when a process
  - (1) Switches from running to waiting (blocked) state
  - (2) Switches from running to ready state
  - (3) Switches from waiting (blocked) to ready
  - (4) Terminates (exits)



- Preemptive:
- Nonpreemptive:

Decision Mode

- CPU scheduling decisions may take place when a process
  - (1) Switches from running to waiting (blocked) state
  - (2) Switches from running to ready state
  - (3) Switches from waiting (blocked) to ready
  - (4) Terminates (exits)



- Preemptive: all cases
- Nonpreemptive:

  (1) & (4)
  (and (3) only when there is no process running)

Example: xv6 scheduling

- Preemptive scheduling
- No explicit run queue
  - xv6 is a simple OS and can create only a fixed number of processes

```
struct proc proc[NPROC];                #define NPROC 64
```

- Scheduler iterates over all processes to choose a ready process

```
enum procstate { UNUSED, USED, SLEEPING, RUNNABLE, RUNNING, ZOMBIE };
void scheduler(void)
    ...
    for(p = proc; p < &proc[NPROC]; p++) {
      acquire(&p->lock);
      if(p->state == RUNNABLE) {
        // Switch to chosen process.
        p->state = RUNNING;
        c->proc = p;
        swtch(&c->context, &p->context);
```

Default scheduling policy: Round Robin

Scheduling Goals

- **What are some reasonable goals for a scheduler?**

Scheduling Goals

- **What are some reasonable goals for a scheduler?**
- Scheduling algorithms can have many different goals:
  - CPU utilization
  - Job throughput (# jobs/unit time)
  - Response time ($Avg(T_{ready})$: avg time spent on ready queue)
  - Fairness (or weighted fairness)
  - …

Scheduling Goals

- **What are some reasonable goals for a scheduler?**
- Scheduling algorithms can have many different goals:
  - CPU utilization
  - Job throughput (# jobs/unit time)
  - Response time ($Avg(T_{ready})$: avg time spent on ready queue)
  - Fairness (or weighted fairness)
  - …
- Non-interactive applications (e.g., supercomputers, MapReduce):
  - Throughput is more important
- Interactive systems (e.g., smartphone):
  - Response time is more important

Scheduling Goals

- **What are some reasonable goals for a scheduler?**
- Scheduling algorithms can have many different goals:
  - CPU utilization
  - Job throughput (# jobs/unit time)
  - Response time (Avg(T_{ready}): avg time spent on ready queue)
  - Fairness (or weighted fairness)
  - …
- Non-interactive applications (e.g., supercomputers, MapReduce):
  - Throughput is more important
- Interactive systems (e.g., smartphone):
  - Response time is more important
- <u>No scheduler can meet all these criteria</u>

Common Scheduling Problems

- Starvation: no progress due to no access to resources
  - E.g., a high priority process always prevents a low priority process from running on the CPU
  - One thread always beats another when acquiring a lock
- Priority inversion
  - A low priority process running before a high priority one
  - Could be a serious problem, especially in real time systems
    - Mars pathfinder “What really happened on Mars?”: https://www.cs.unc.edu/~anderson/teach/comp790/papers/mars_pathfinder_long_version.html
- Other
  - Deadlock, livelock, …

First Come, First Served (FCFS)

- Selection function: FCFS; also called FIFO
- Decision mode: typically, nonpreemptive
  - A process run until it blocks itself

Process	Burst Time	Arrival Time
P1	24	0.000
P2	3	0.001
P3	3	0.002

First Come, First Served (FCFS)

- Selection function: FCFS; also called FIFO
- Decision mode: typically, nonpreemptive
  - A process run until it blocks itself

Process	Burst Time	Arrival Time
P1	24	0.000
P2	3	0.001
P3	3	0.002

- Turnaround time = completion time - arrival time
  - P1 = 24; P2 = 27; P3 = 30
  - Average turnaround time: (24 + 27 + 30) / 3 = 27

The Convoy Effect

- FCFS scheduler, but the arrival order has changed

Process	Burst Time	Arrival Time
P1	24	0.002
P2	3	0.000
P3	3	0.001

The Convoy Effect

- FCFS scheduler, but the arrival order has changed

Process	Burst Time	Arrival Time
P1	24	0.002
P2	3	0.000
P3	3	0.001

- Turnaround time: P1 = 30; P2 =3; P3 = 6
  - Average turnaround time: (3 + 6 + 30) / 3 = 13
  - Much better than the previous arrival order!

The Convoy Effect

- FCFS scheduler, but the arrival order has changed

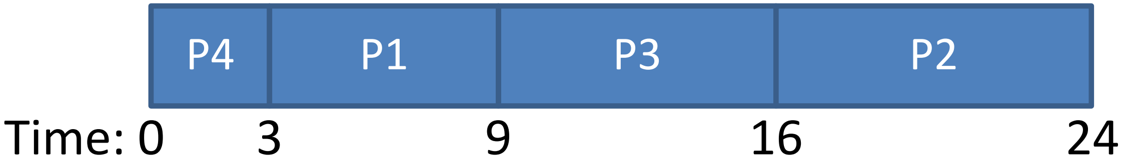
Process	Burst Time	Arrival Time
P1	24	0.002
P2	3	0.000
P3	3	0.001

- Turnaround time: P1 = 30; P2 =3; P3 = 6
  - Average turnaround time: (3 + 6 + 30) / 3 = 13
  - Much better than the previous arrival order!
- Convoy effect (a.k.a. head-of-line blocking)
  - Long process can impede short processes
  - E.g.: CPU bound process followed by I/O bound process

Shortest Job First (SJF)

- Schedule processes based on their next CPU burst length
  - Shortest processes go first

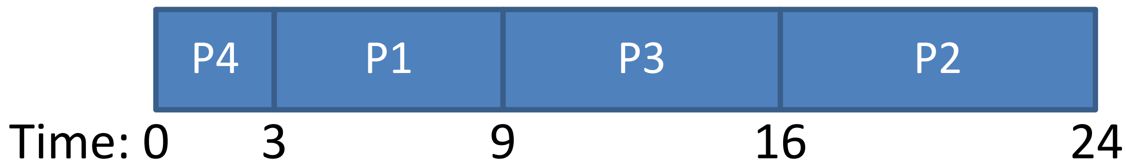
Process	Burst Time	Arrival Time
P1	6	0
P2	8	0
P3	7	0
P4	3	0



Shortest Job First (SJF)

- Schedule processes based on their next CPU burst length
  - Shortest processes go first

Process	Burst Time	Arrival Time
P1	6	0
P2	8	0
P3	7	0
P4	3	0



- Average turnaround time: (3 + 9 + 16 + 24) / 4 = 13
- SJF is optimal: guarantees minimum average wait time

Predicting CPU Burst Length

- Problem: future CPU burst times may be unknown
- Possible solution: estimate the next burst time based on previous burst lengths
  - Assumes process behavior is not highly variable
  - Example: Use exponential averaging
    - t_n – measured length of the n^{th} CPU burst
    - τ_{n+1} – predicted value for $n+1^{th}$ CPU burst
    - α – weight of current and previous measurements ($0 \le \alpha \le 1$)
    - $\tau_{n+1} = \alpha t_n + (1 - \alpha)\, \tau_n$
  - Other approaches can be used too, but is perfect prediction possible?

What About Arrival Time?

- SJF scheduler, CPU burst lengths are known

Process	Burst Time	Arrival Time
P1	24	0
P2	3	2
P3	3	3

What About Arrival Time?

- SJF scheduler, CPU burst lengths are known

Process	Burst Time	Arrival Time
P1	24	0
P2	3	2
P3	3	3

What About Arrival Time?

- SJF scheduler, CPU burst lengths are known

Process	Burst Time	Arrival Time
P1	24	0
P2	3	2
P3	3	3

- Scheduler must choose from *ready* processes at the time of decision making
  - Can lead to head-of-line blocking
  - Average turnaround time: (24 + 25 + 27) / 3 = 25.3

Preemptive SJF (PSJF)

- Also known as "Shortest Time-To-Completion First" (STCF)
- Processes with long bursts can be context switched out in favor or short processes

Process	Burst Time	Arrival Time
P1	24	0
P2	3	2
P3	3	3

Preemptive SJF (PSJF)

- Also known as "Shortest Time-To-Completion First" (STCF)
- Processes with long bursts can be context switched out in favor or short processes

Process	Burst Time	Arrival Time
P1	24	0
P2	3	2
P3	3	3

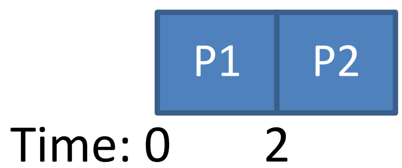
P1

Time: 0

Preemptive SJF (PSJF)

- Also known as "Shortest Time-To-Completion First" (STCF)
- Processes with long bursts can be context switched out in favor or short processes

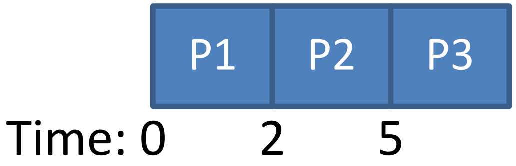
Process	Burst Time	Arrival Time
P1	24	0
P2	3	2
P3	3	3



Preemptive SJF (PSJF)

- Also known as "Shortest Time-To-Completion First" (STCF)
- Processes with long bursts can be context switched out in favor or short processes

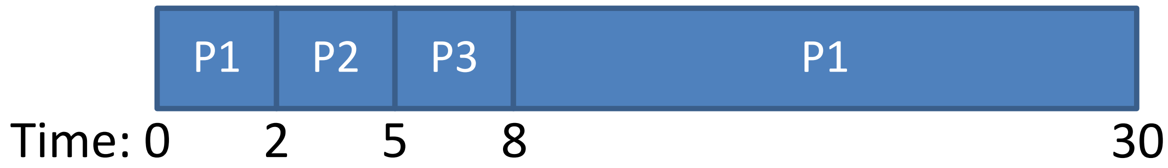
Process	Burst Time	Arrival Time
P1	24	0
P2	3	2
P3	3	3



Preemptive SJF (PSJF)

- Also known as "Shortest Time-To-Completion First" (STCF)
- Processes with long bursts can be context switched out in favor or short processes

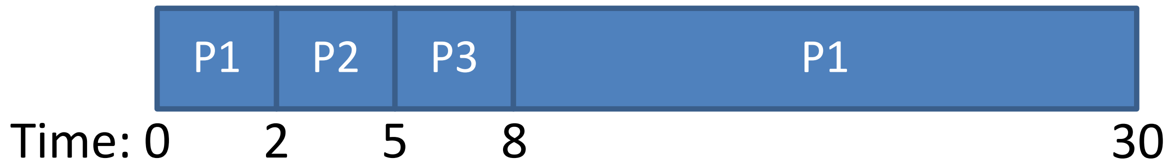
Process	Burst Time	Arrival Time
P1	24	0
P2	3	2
P3	3	3



Preemptive SJF (PSJF)

- Also known as "Shortest Time-To-Completion First" (STCF)
- Processes with long bursts can be context switched out in favor or short processes

Process	Burst Time	Arrival Time
P1	24	0
P2	3	2
P3	3	3



- Turnaround time: P1 = 30; P2 = 3; P3 = 5
  - Average turnaround time: (30 + 3 + 5) / 3 = 12.7
- STCF is also optimal
  - Assuming you know future CPU burst times

Interactive Systems

- Imagine you are typing/clicking in a desktop app
  - You don't care about turnaround time
  - What you care about is responsiveness
    - E.g. if you start typing but the app doesn't show the text for 10 seconds, you'll become frustrated

Interactive Systems

- Imagine you are typing/clicking in a desktop app
  - You don't care about turnaround time
  - What you care about is responsiveness
    - E.g. if you start typing but the app doesn't show the text for 10 seconds, you'll become frustrated

- Response time = first run time – arrival time
  - Note on terminology
    - Some other domains use "latency" to mean this time
    - Example: In real-time embedded systems response time is defined as completion time – arrival time (= turnaround time)
    - Because the final output of execution determines responsiveness to the external events (e.g., control system)

Response vs. Turnaround

- Assume an STCF scheduler

Process	Burst Time	Arrival Time
P1	6	0
P2	8	0
P3	10	0

- Avg. turnaround time: (6 + 14 + 24) / 3 = 14.7
- Avg. response time: (0 + 6 + 14) / 3 = 6.7

Round Robin (RR)

- Round robin (a.k.a time slicing) scheduler is designed to reduce response times
  - RR runs each process for one time slice
    - Also called scheduling quantum
    - Switches to another process in the next time slice
  - The duration of time slice or quantum is determined by the OS considering processor speed (why?)
    - e.g., xv6 time slice length = 1 timer interrupt period (tick)

RR vs. PSJF

(Preemptive SJF)

Process	Burst Time	Arrival Time
P1	6	0
P2	8	0
P3	10	0

PSJF

P1	P2	P3

Time: 0 6 14 24

- Avg. turnaround time: (6 + 14 + 24) / 3 = **14.7**
- Avg. response time: (0 + 6 + 14) / 3 = **6.7**

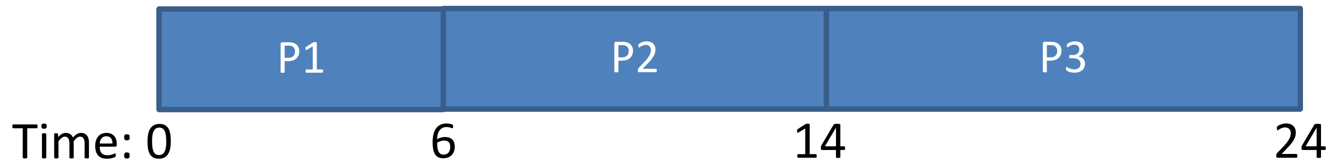
RR

RR vs. PSJF

(Preemptive SJF)

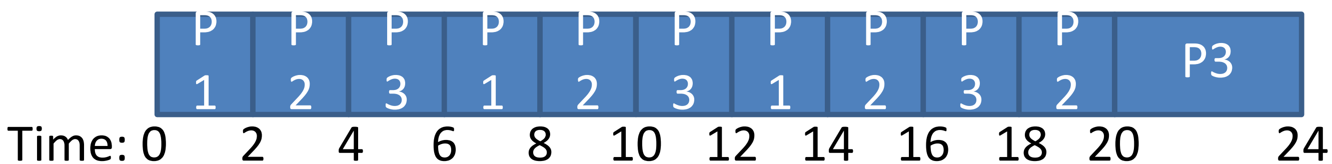
Process	Burst Time	Arrival Time
P1	6	0
P2	8	0
P3	10	0

PSJF



- Avg. turnaround time: (6 + 14 + 24) / 3 = **14.7**
- Avg. response time: (0 + 6 + 14) / 3 = **6.7**

RR



- 2 second time slices
- Avg. turnaround time: (14 + 20 + 24) / 3 = **19.3**
- Avg. response time: (0 + 2 + 4) / 3 = **2**

Round-Robin (RR)

- Achieves fairness
  - Each process receives 1/N CPU time

Round-Robin (RR)

- Achieves fairness
  - Each process receives 1/N CPU time
- Worst possible turnaround times
  - If time quantum is large → FIFO behavior

Round-Robin (RR)

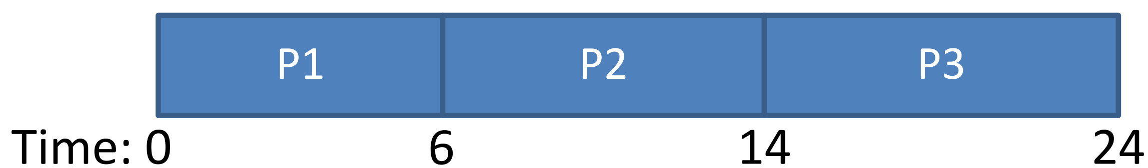
- Achieves fairness
  - Each process receives 1/N CPU time
- Worst possible turnaround times
  - If time quantum is large → FIFO behavior
- How to select the time slice?
  - Smaller time slices = faster response times
  - So why not select a very tiny time slice? (e.g., 1μs)
  - Context switching overhead
    - Each context switch wastes CPU time (~10μs)
    - If time slice is too short, c/s overhead will dominate overall performance
  - Typical time slices are between 1ms and 100ms

Priority Scheduler

- Associate a priority with each process
  - Schedule high priority processes first
  - Can be either preemptive or non-preemptive
  - Priority can be determined statically or dynamically

- Example: Static priority scheduler

Process	Priority	Arrival Time
P1	1	0
P2	2	0
P3	3	0



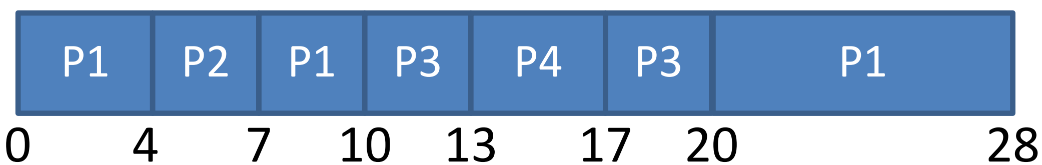
Priority Scheduling

- Problem?
  - Starvation: high priority tasks can dominate the CPU
- Possible solution: dynamically vary priorities
  - Vary based on process behavior
  - Vary based on wait time (i.e. length of time spent in the ready queue)

Earliest Deadline First (EDF)

- Each process has a deadline it must finish by
- Priorities are assigned according to deadlines
  - Tighter deadlines are given higher priority

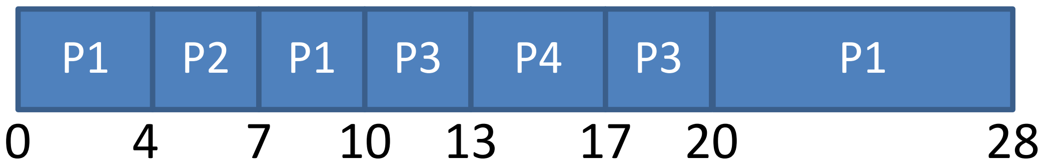
Process	Burst Time	Arrival Time	Deadline
P1	15	0	40
P2	3	4	10
P3	6	10	20
P4	4	13	18



Earliest Deadline First (EDF)

- Each process has a deadline it must finish by
- Priorities are assigned according to deadlines
  - Tighter deadlines are given higher priority

Process	Burst Time	Arrival Time	Deadline
P1	15	0	40
P2	3	4	10
P3	6	10	20
P4	4	13	18



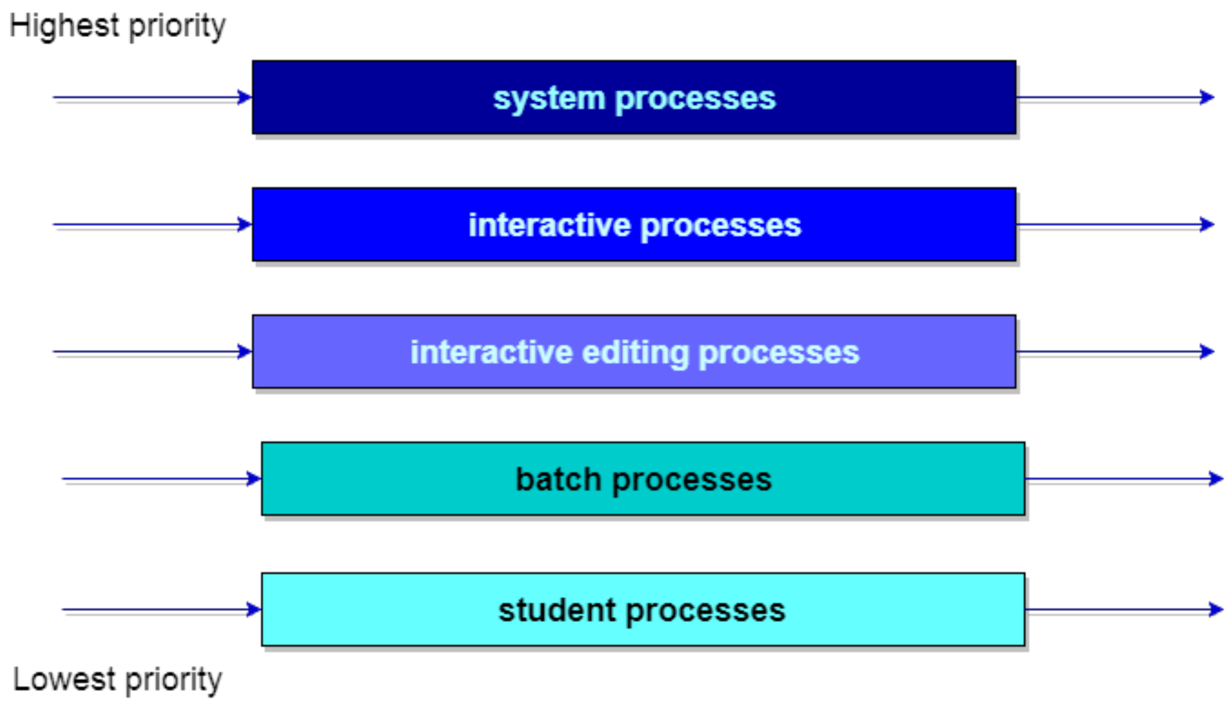
- EDF is optimal (assuming preemption)
- But, it’s only useful if processes have known deadlines
  - Typically used in real-time OSes

Combining Algorithms

- Scheduling algorithms can be combined
  - Minimize response time and turnaround time
  - Dynamically adjust process priorities over time
- How? Multiple-level feedback queues (MLFQ)
  - Goals:
    - Responsiveness, low overhead
    - Starvation freedom
    - Some tasks are high/low priority
    - Fairness (among equal priority tasks)
  - But hard to be perfect at all of them!

Multiple-level feedback queues (MLFQ)

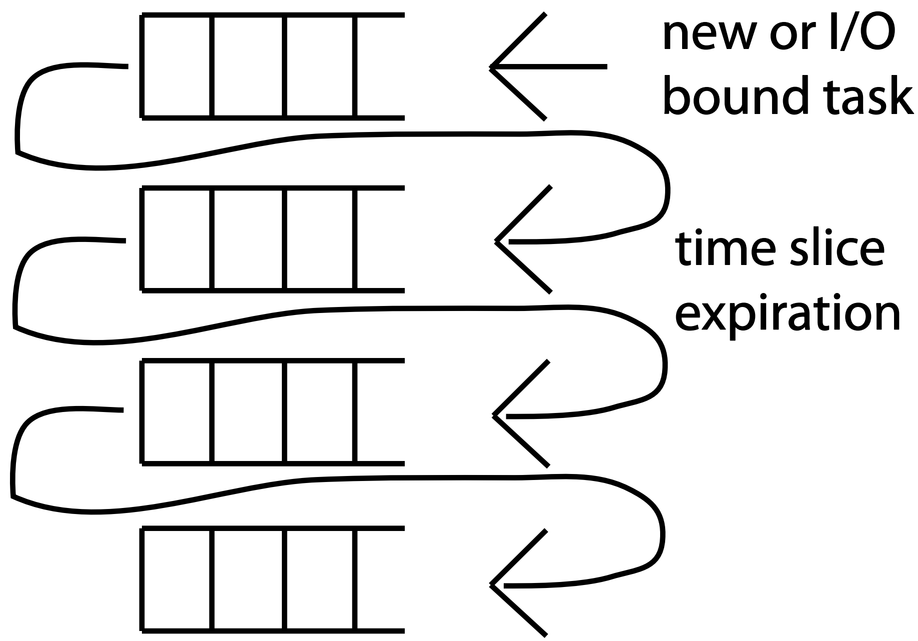
- Multiple queues representing different process types
- Queues have different priorities
- Within each queue, different scheduling policy or time slice can be used
- Processes can be moved among queues based on observed behavior
  - Feedback: Switch from interactive to CPU-bound behavior



MLFQ Example

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

Process Queues



UNIX Scheduler

- Key principles
  - Reward interactive processes over CPU hogs
  - Want to minimize overall response time
    - Time from keystroke (putting process on ready queue) to executing the handler (process running)

UNIX Scheduler

- Key principles
  - Reward interactive processes over CPU hogs
  - Want to minimize overall response time
    - Time from keystroke (putting process on ready queue) to executing the handler (process running)
- Typical implementation
  - Use MLFQ of 3-4 classes spanning many priority levels
  - Priority scheduling across queues, RR within a queue
    - The process with the highest priority always runs
    - Processes with the same priority are scheduled RR
  - Processes dynamically change priority
    - Increases priority if process blocks before end of quantum
    - Decreases priority if process uses entire quantum

Fair Share Schedulers

- Lottery scheduling
- Stride scheduling

Next lecture

Real-Time Scheduling

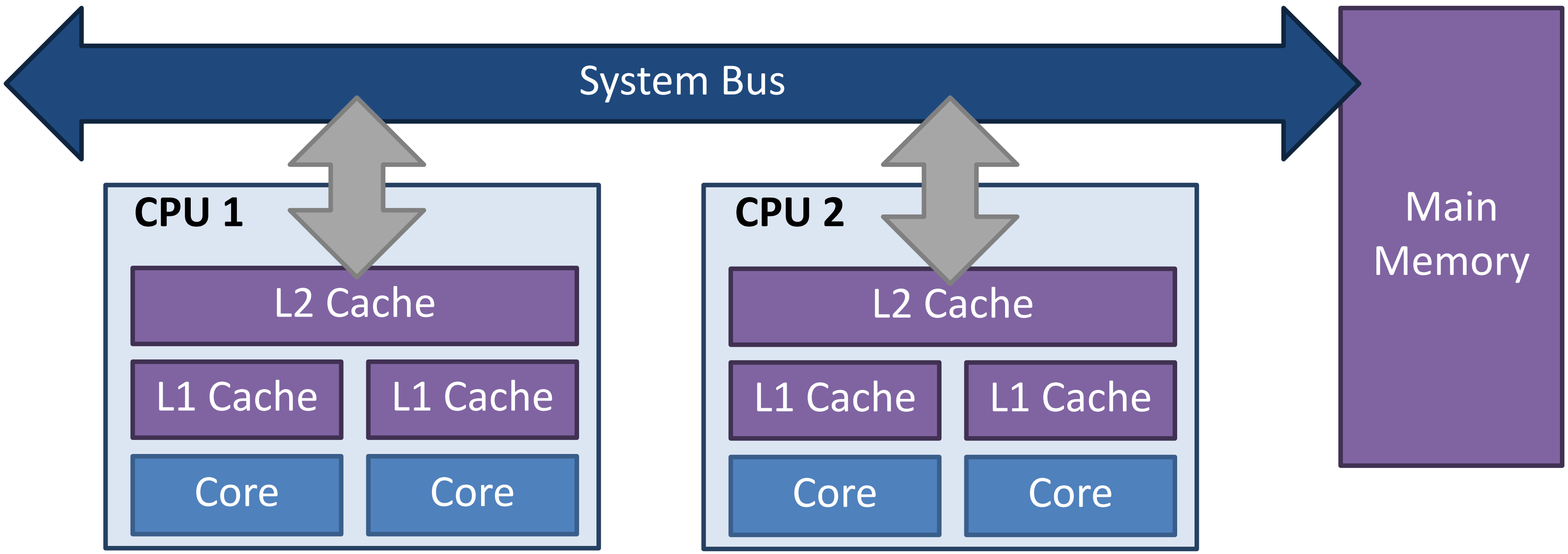
- Hard real-time systems
  - Complete a critical task within a guaranteed amount of time (deadline)
  - Requires worst-case timing analysis
    - Minimize unavoidable and unforeseeable variation in the amount of time to execute a particular process
  - Scheduling algorithms: EDF, RM (Rate Monotonic), ...
- Soft real-time systems
  - Deadlines may be occasionally be missed
    - Deadline miss → Degradation in Quality of Service (QoS) but not catastrophic
  - Question: How to define “occasionally”?
    - Probabilistic requirements
    - Data freshness, usefulness over time
- More in CS 251/EE 255: Real-Time Embedded Systems

Where are we?

- Thus far, all of our schedulers have assumed a single CPU core
- What about systems with multiple CPUs?
  - Things get a lot more complicated when the number of CPUs > 1

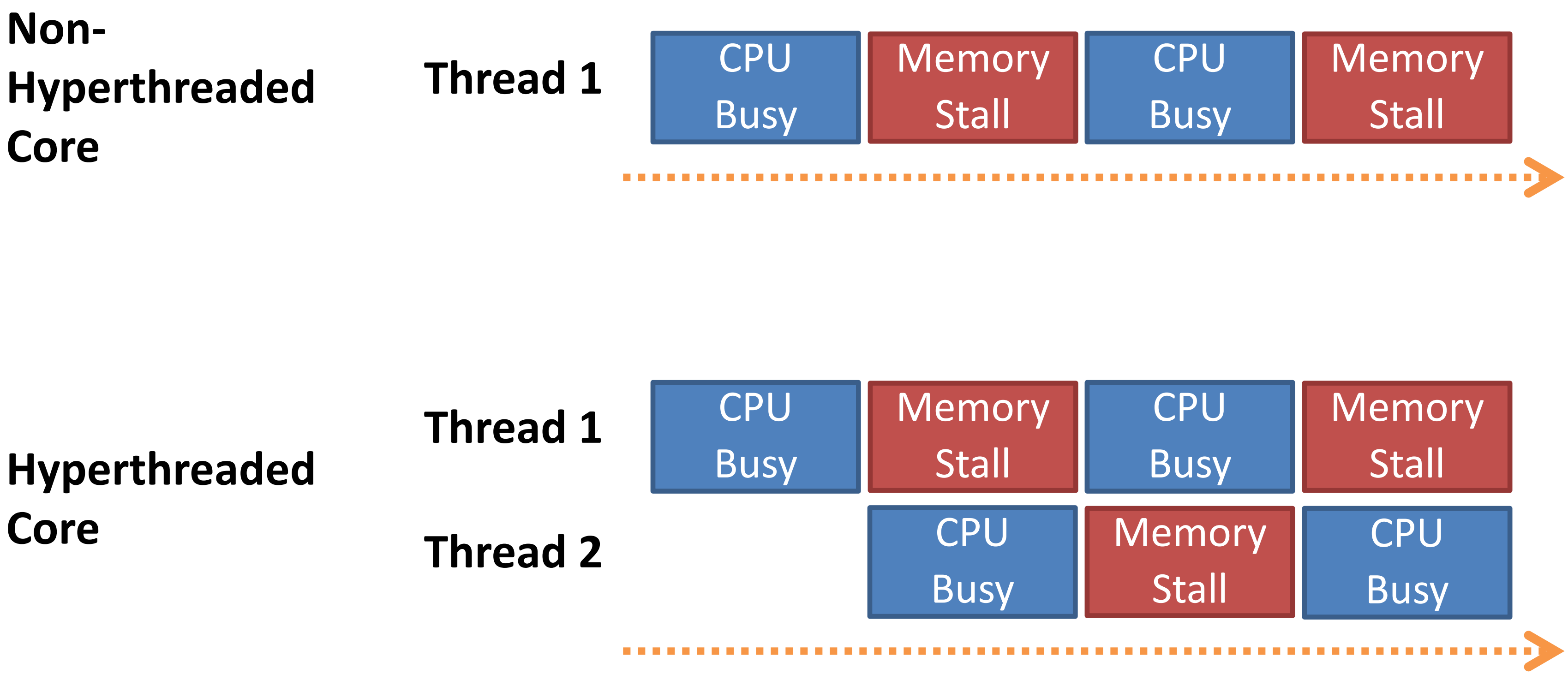
Symmetric Multiprocessing (SMP)

- ≥2 homogeneous processors
  - May be in separate physical packages
- Shared main memory and system bus
- Single OS that treats all processors equally



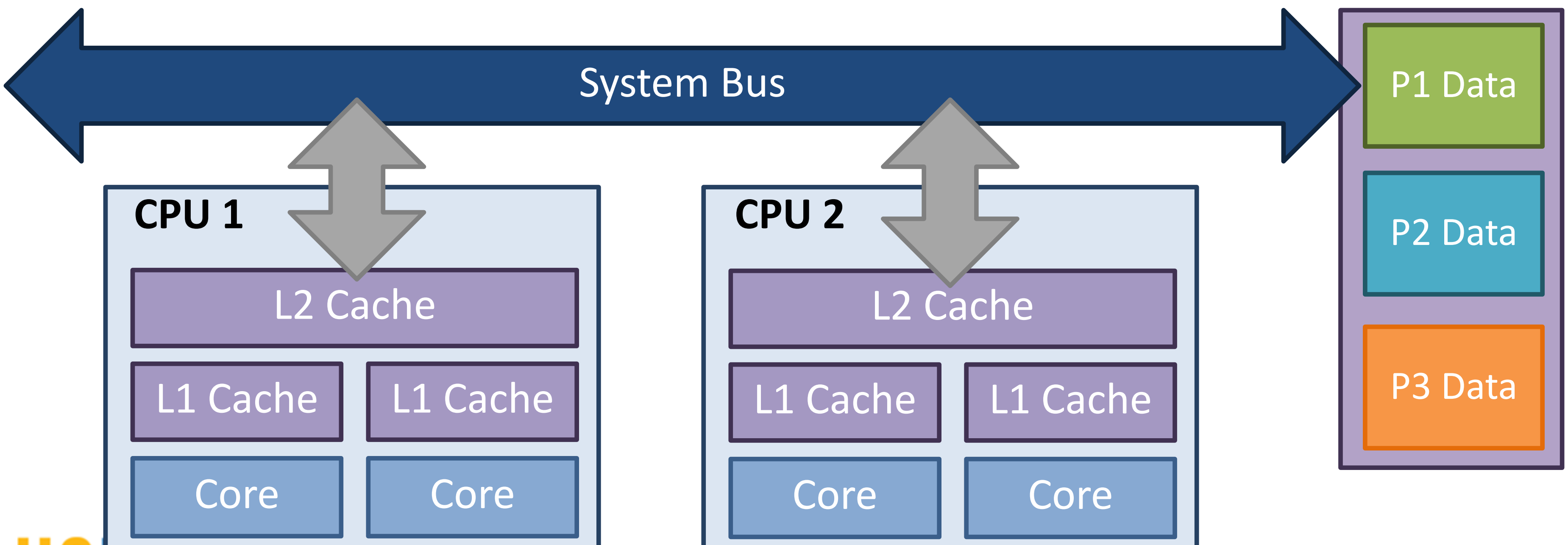
Hyperthreading

- Two hardware threads on a single CPU core
  (logical cores)



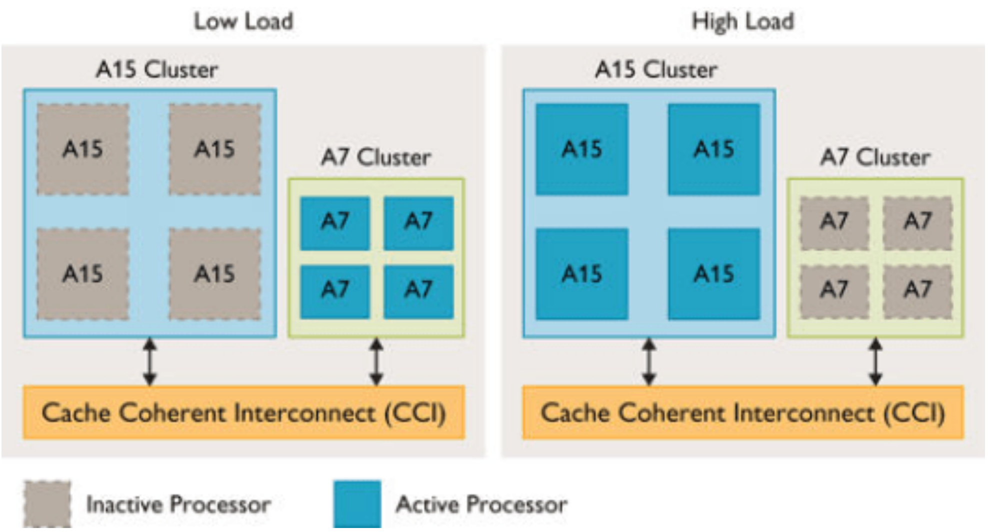
Brief Intro to CPU Caches

- Process performance is linked to locality
  - Ideally, a process should be placed close to its data
- Shared data is problematic due to cache coherency
  - CPU1 writes variable ***x***, new value is cached in CPU1's cache
  - CPU2 reads ***x***, but value in main memory is stale

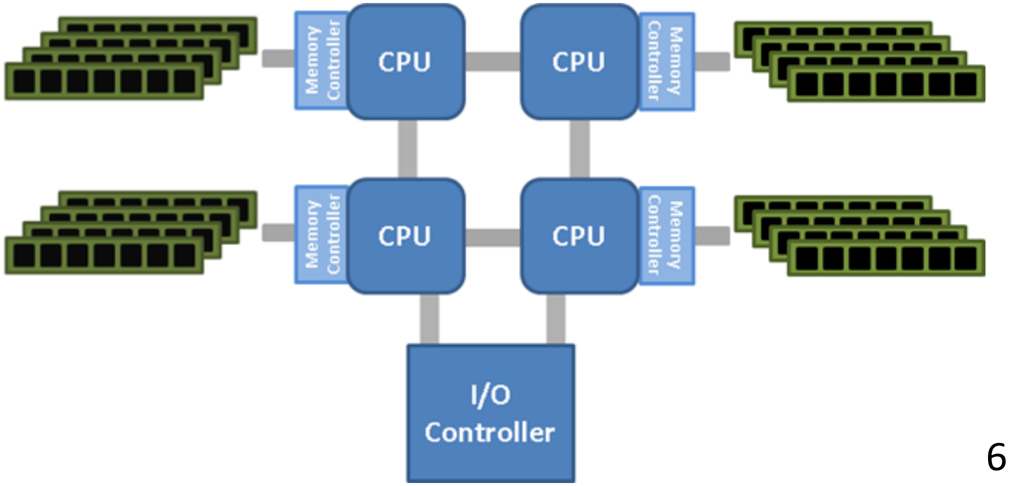


Other Issues

- Processor heterogeneity (e.g, ARM big.LITTLE)
  - How to balance workload?

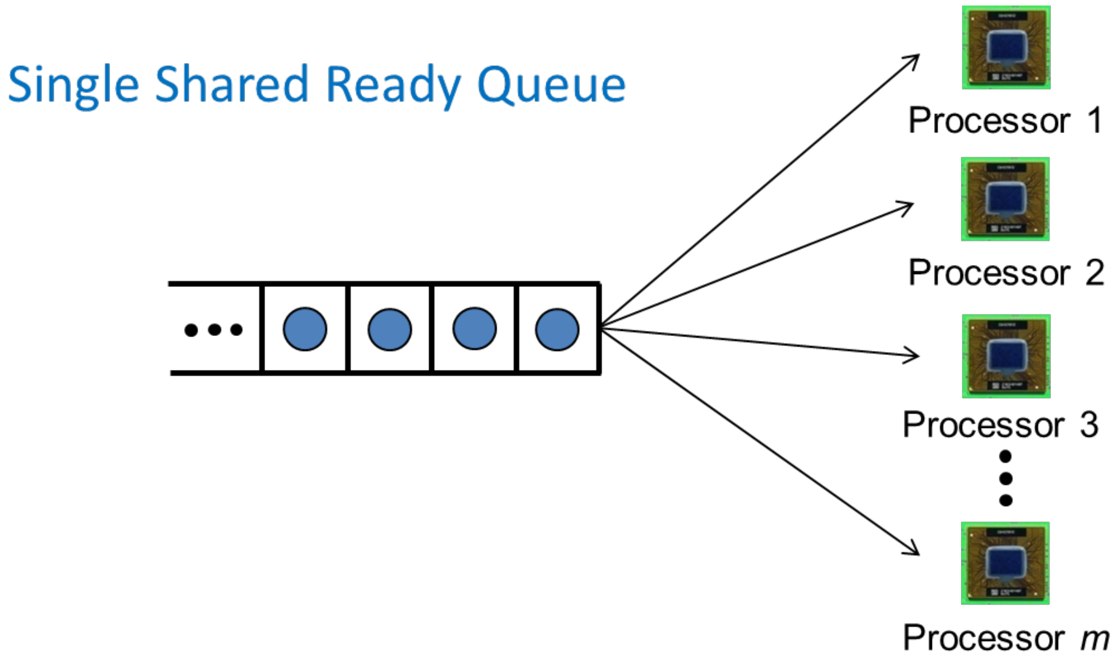


- Non Uniform Memory Access (NUMA)
  - Memory access time depends on the location of the data relative to the requesting CPU core



Multiprocessor Scheduling

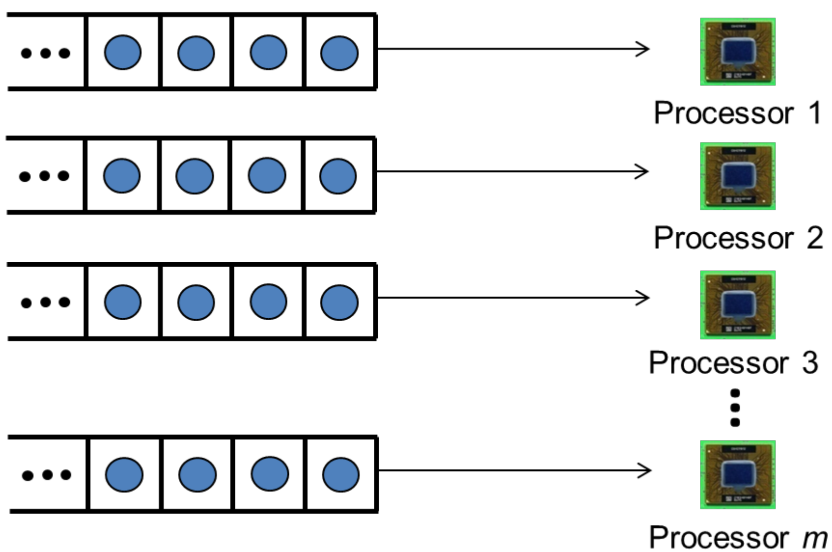
- Global scheduling
  - a.k.a. Single Queue Multiprocessor Scheduling
  - All processes go into a single queue
  - Work conserving: Unused processor time can easily be reclaimed
  - Task migration overhead (ready queue locks; cache reloading)



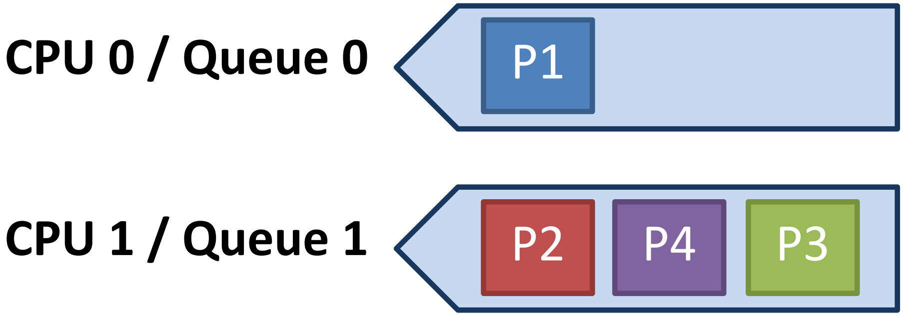
Multiprocessor Scheduling

- Partitioned Scheduling
  - a.k.a. Multiple Queue Multiprocessor Scheduling
  - Each CPU maintains it's own queue of processes
    - Very little shared data; Queues are (mostly) independent
    - Respects cache affinity
  - Load imbalance problem: needs load balancing via task migration

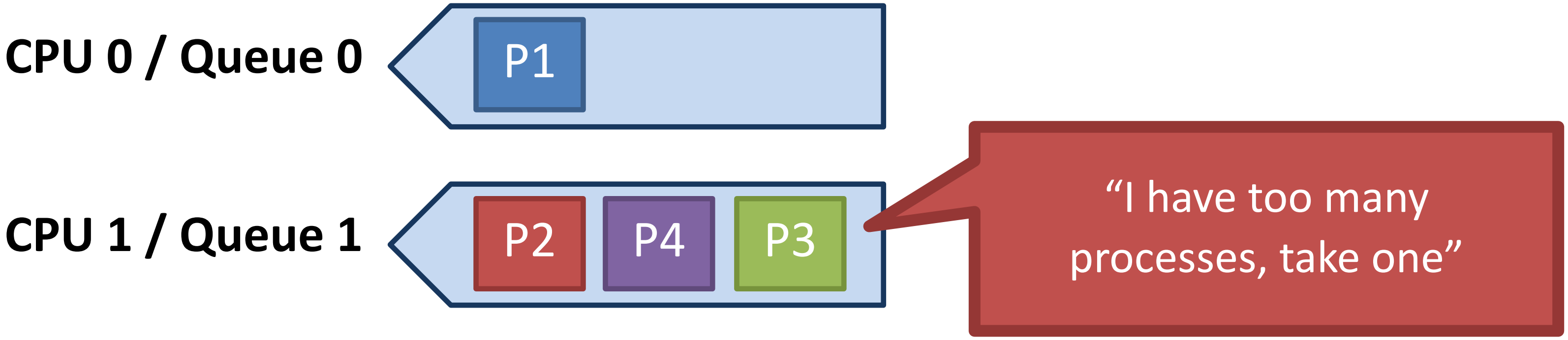
One Ready Queue Per Processor



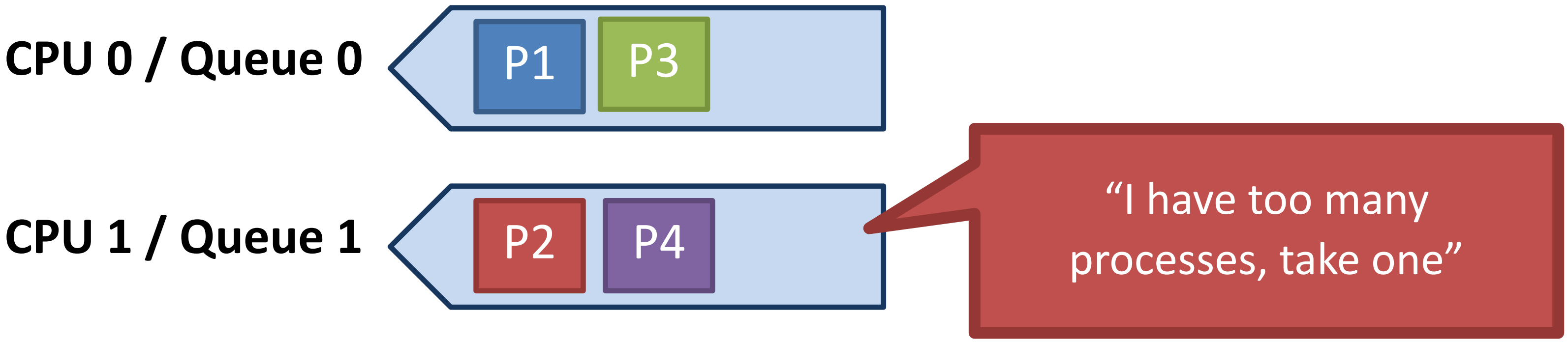
Strategies for Process Migration



Strategies for Process Migration

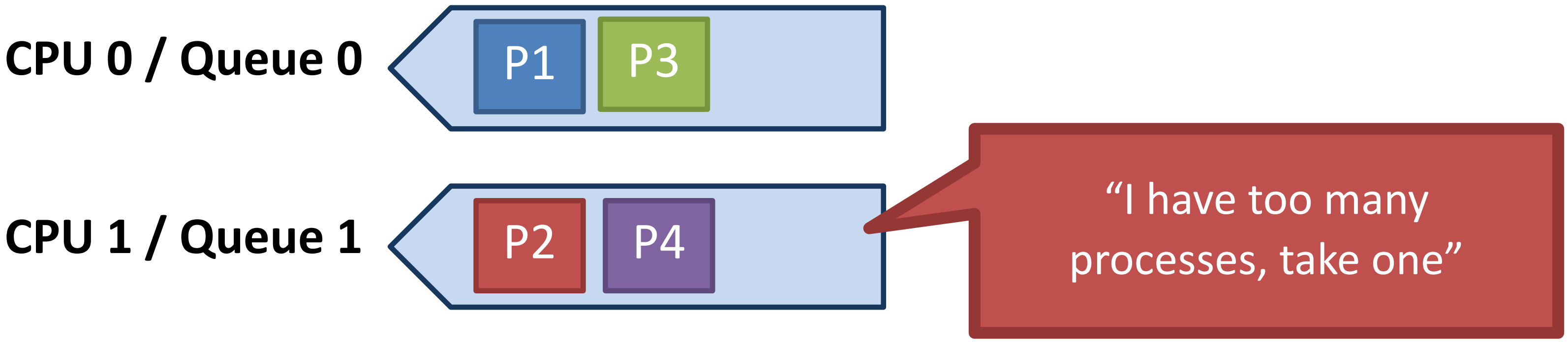


Strategies for Process Migration

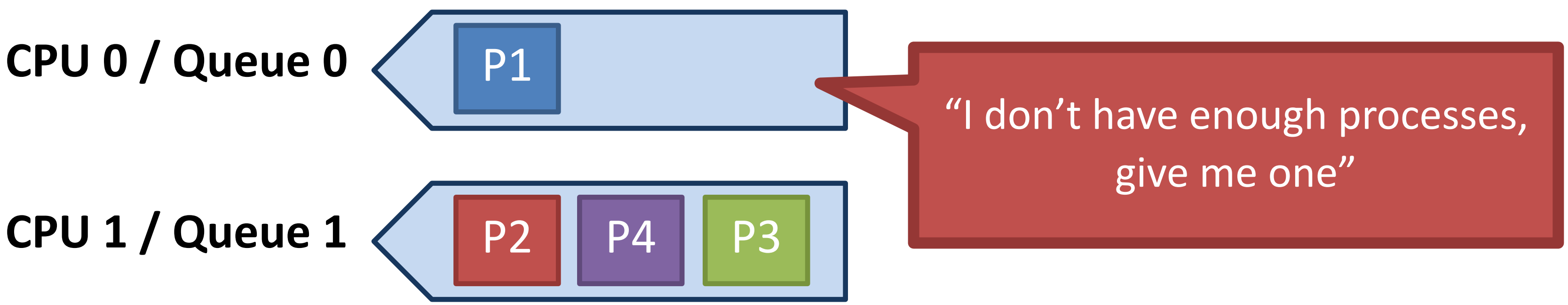


Strategies for Process Migration

- Push migration

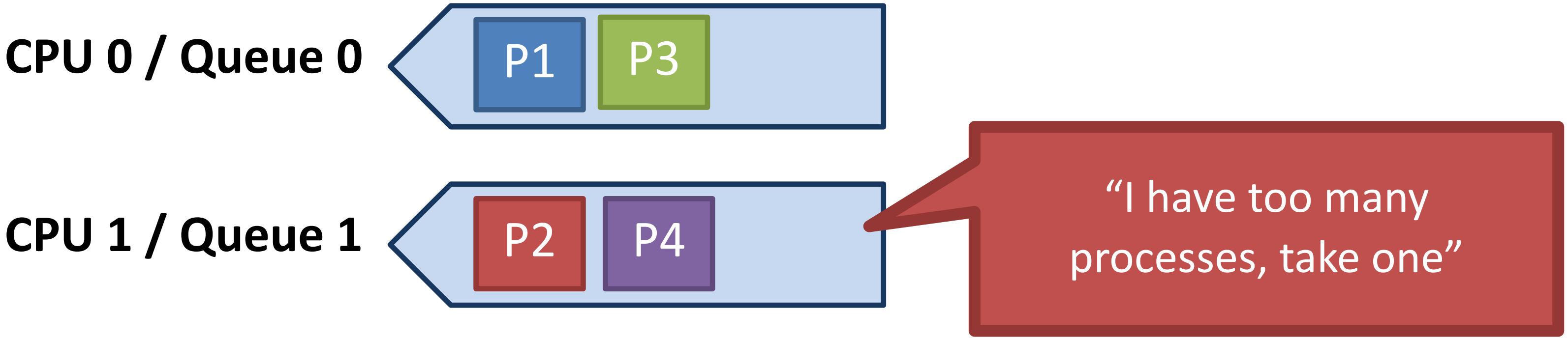


- Pull migration, a.k.a. work stealing

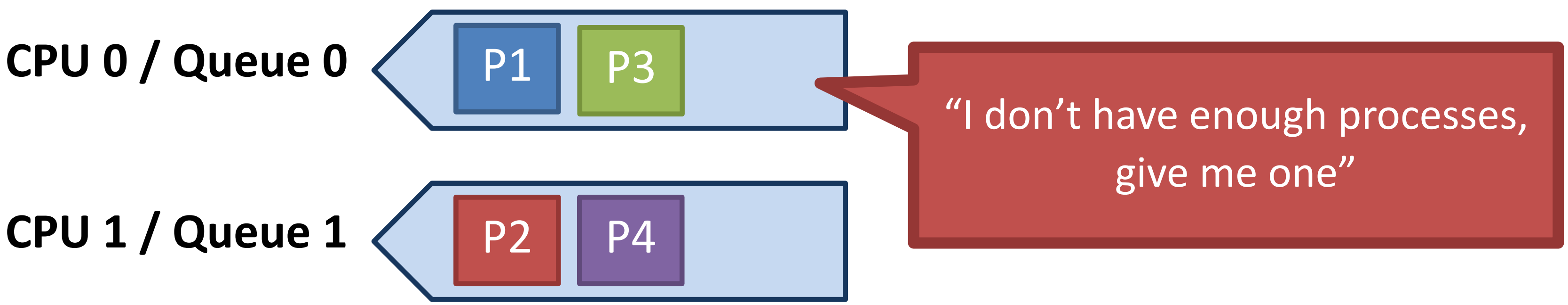


Strategies for Process Migration

- Push migration

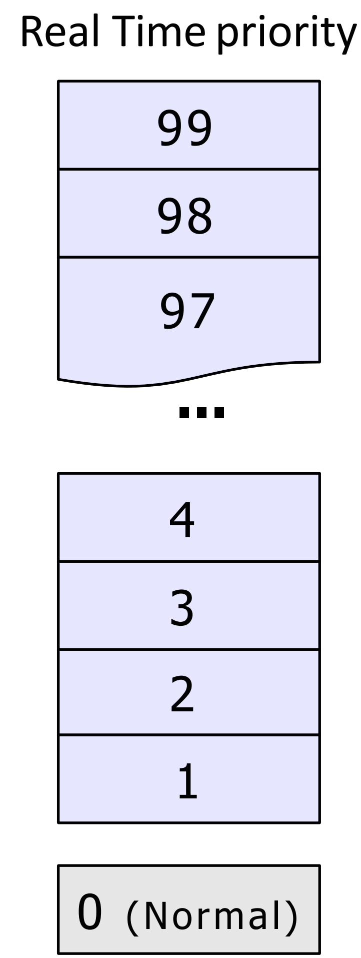


- Pull migration, a.k.a. work stealing



Linux Scheduling

- Linux has a hierarchical scheduling classes
- Time-sharing class
  - SCHED_OTHER: standard time-sharing tasks
  - SCHED_BATCH: batch style tasks
  - SCHED_IDLE: very low-priority background tasks
  - One of 40 priority levels (-20..0..19) – ***nice*** values
- Real-time class
  - 99 priority levels (1 to 99)
  - SCHED_FIFO (FIFO for same-priority tasks)
  - SCHED_RR (round-robin for same-priority tasks)
  - SCHED_DEADLINE: EDF dynamic priority
    - Only in recent Linux kernels. Non-POSIX compliant



Linux Scheduling (2)

- SCHED_OTHER : Default scheduling policy
  - Went through several iterations
- Currently, Completely Fair Scheduler (CFS)
  - Fair scheduler, like stride scheduling
  - Supersedes previous "O(1) scheduler" which emphasized constant time scheduling regardless of # of processes
    - Linux O(1) scheduler was based on MLFQ
  - CFS is O(log(N)) because of red-black tree
  - Is it really fair? No, it's approximate fair scheduling

Linux Multiprocessor Scheduling

- One Ready Queue Per Processor
- Time-sharing (normal) tasks
  - Scheduling happens on a per-processor basis
    - To minimize migration overhead
  - Work stealing used for load balancing
    - Migration occasionally occur
- Real-time tasks
  - Behaves like global scheduling (using on-demand migrations)
    - A task is scheduled as soon as at least one of the permitted processors is not executing a higher-priority task
  - Partitioned scheduling
    - Can be configured with “CPU affinity mask”
- More in CS 251/EE 255: Real-Time Embedded Systems

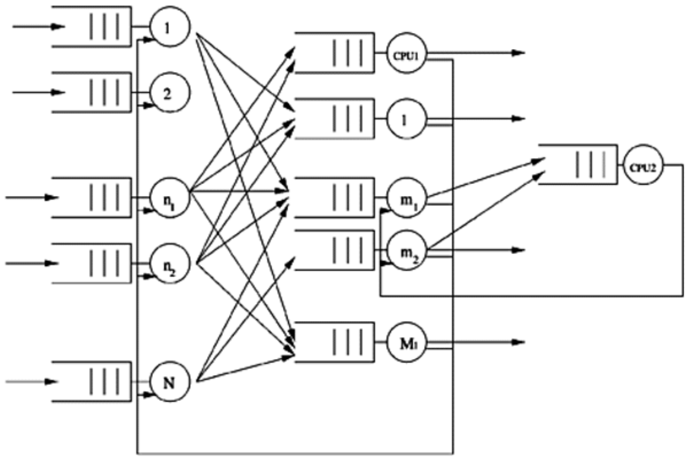
Scheduler Comparison

- Which one is the best?
- Depends on...
  - The system workload (extremely variable)
  - Hardware support for the dispatcher
  - Relative weighting of performance criteria (response time, CPU utilization, throughput...)
  - The evaluation method used (each has its limitations...)

Algorithm Evaluation

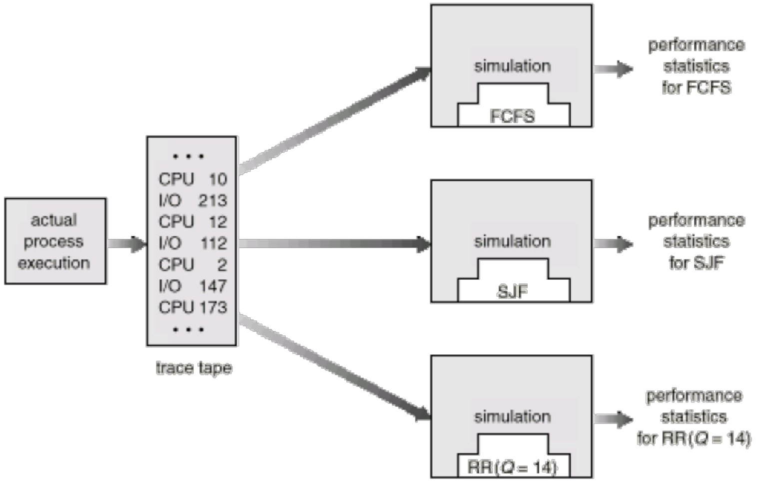
- Deterministic Modeling
  - Analytical evaluation; produces a formula or number for performance evaluation
  - Simple and fast
  - Too specific and require too much exact knowledge
- Queuing Models
  - Based on queuing theory

M/M/1 queue



Algorithm Evaluation (2)

- Simulation Analysis
  - Trace-Driven Simulation
    - Trace: a time-ordered record of events on a real system
    - Generally used in analyzing or tuning resource management systems
  - Example
    - Paging algorithms, Cache analysis, CPU scheduling algorithms, Deadlock prevention algorithms, Storage allocation algorithms, etc



- Full system evaluation