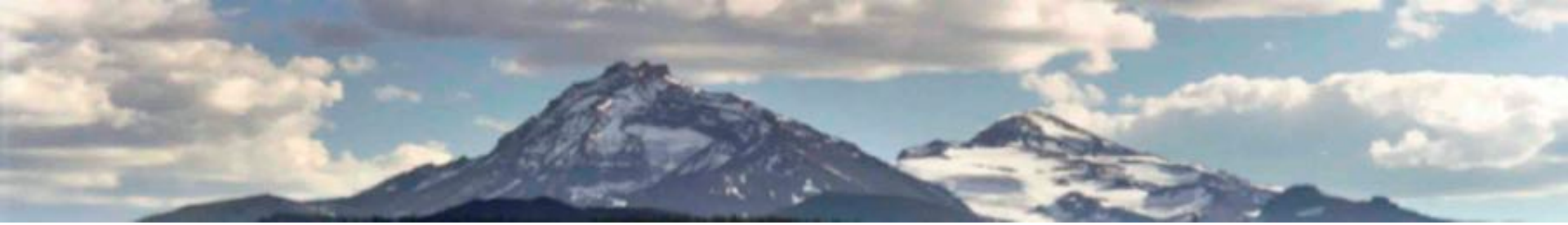


# CIS 415
# Operating Systems
# Scheduling

Prof. Allen D. Malony
Department of Computer and Information Science
Spring 2020

UNIVERSITY OF OREGON

# *Logistics*

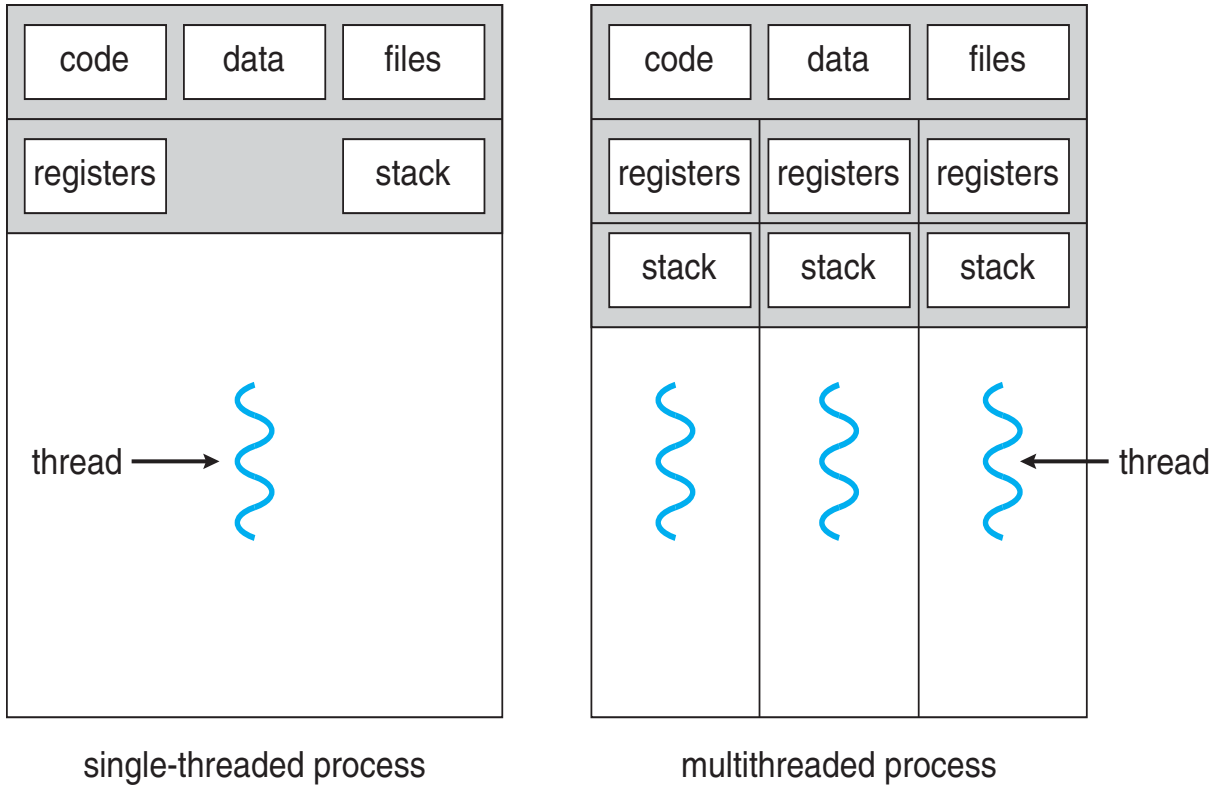
- Project 1 due Sunday, April 19, 11:59pm
- Lab 3 this Friday
  - Clarification on individual work policy
- Read Chapter 5
- Change in grading model
  - Midterm 22%
  - Final 28%
  - Projects 45%
  - Lab 5%

# *Outline*

- Bit more on threads
- Basic scheduling concepts and criteria
- Scheduling algorithms
- Thread scheduling
- Multiple processor scheduling
- Real-Time CPU scheduling
- Algorithm evaluation

# *Single-Threaded vs. Multi-Threaded*

- Regular UNIX process can be thought of as a special case of a multithreaded process
  - A process that contains just one thread



# *Terms You Might Hear*

- *Reentrant code*
  - Code that can be run by multiple threads concurrently
- *Thread-safe libraries*
  - Library code that permits multiple threads to invoke the safe function
  - Mainly concerned with variables that should be private to individual threads
  - Requires moving some global variables to local variables
- Requirements
  - Rely only on input data
    - or some thread-specific data
  - Must be careful about locking (see later)

# *Thread Assignment*

- How many kernel threads should a process have?
  - In an *M:N* model there can be significantly more user-level threads (*M*) than kernel-level threads (*N*)
  - Kernel threads are the "real" threads that can be allocated to a CPU (core) and run
  - Want to keep enough kernel threads to satisfy the desired level of concurrency in a program and activity in the system
- Suppose that all kernel threads except 1 are blocked
  - What happens if the last kernel thread blocks?
  - Does it matter if there are more user threads "ready" to run?
- In multiprocessing, *thread affinity* is the notion of assigning a thread to run on a particular CPU
  - Help to improve the thread's performance by keeping its execution resources (CPU, cache, memory) local to CPU

# *Scheduler Activation*

- It would be nice if the kernel told the application that a kernel thread was blocking
  - *Scheduler activation*
    - at thread block, the kernel tells the application via an *upcall*
    - an *upcall* is a general term for an invocation of application function from the kernel
  - User-level thread scheduler can then get a new user-level thread created
- Way of conveying information between the kernel and the user-level thread scheduler regarding the disposition of:
  - # user-level threads (increase or decrease)
  - User-level thread state
    - running to waiting, waiting to ready

# *Why not threads?*

- Threads can interfere with one another
  - Impact of more threads on caches
  - Impact of more threads on TLB
  - Bug in one thread can lead to problems in others
- Executing multiple threads may slow them down
  - Impact of single thread vs. switching among threads
- Harder to program a multithreaded program
  - Multitasking hides context switching
  - Multithreading introduces concurrency issues

# *Resource Allocation*

- In a multiprogramming / multiprocessing system, OS shares resources among running processes
  - There are different types of OS resources?
- Which process gets access to which resources and why?
  - To maximize performance
  - To increase utilization, throughput, responsiveness, …
  - Enforce priorities

# *Resource Types*

- *Memory*: Allocate portion of finite resource
  - Physical resources are limited
  - Virtual memory tries to make this appear infinite
- *I/O*: Allocate portion of finite resource and time spent with the resource
  - Store information on disk
  - A time slot to store that information
- *CPU*: Allocate time slot with resource
  - A time slot to run instructions
- We will focus on CPU resource allocation for now

# *Types of CPU Scheduling*

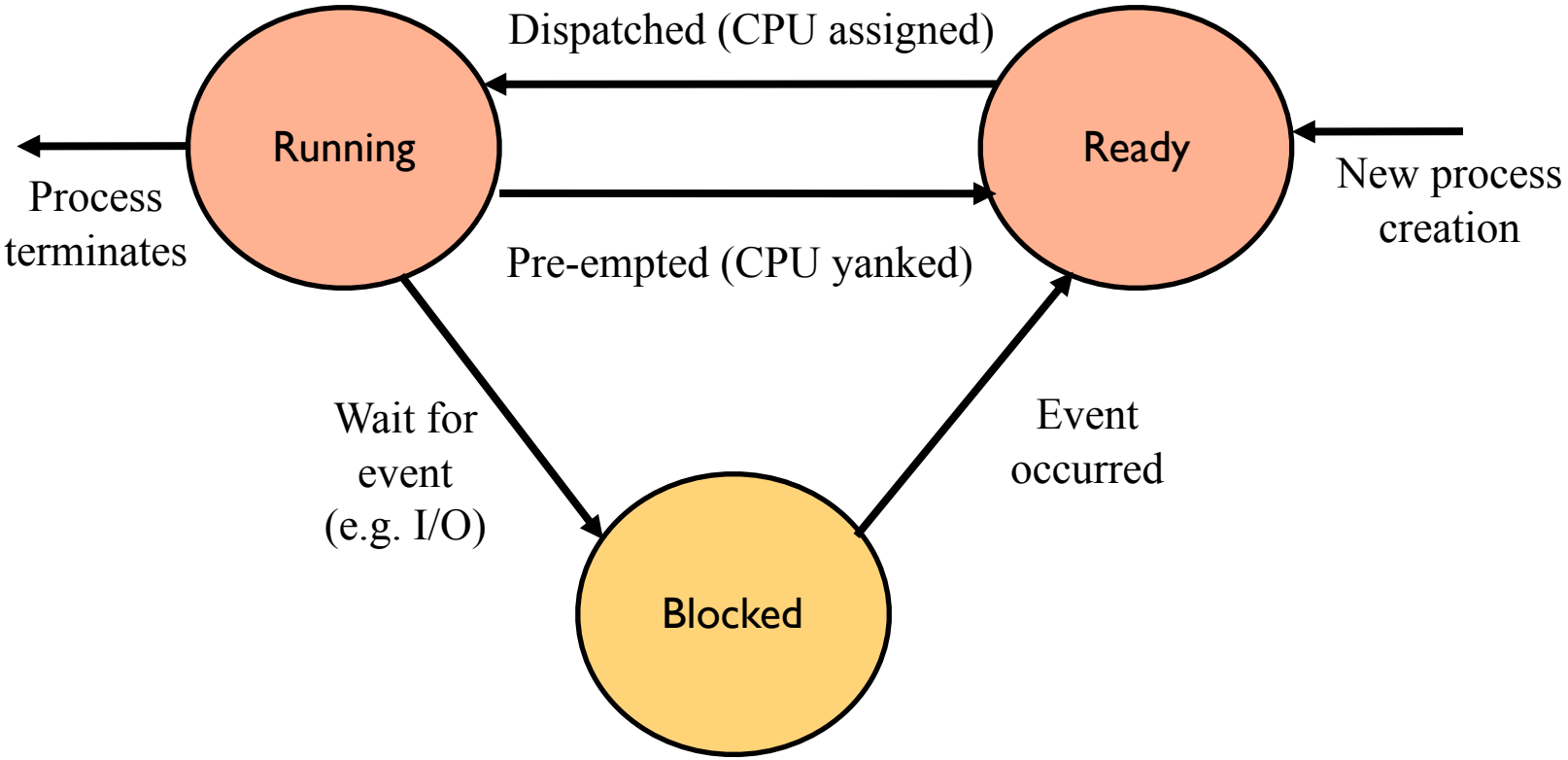
- CPU resource allocation is also known as *scheduling*
- *Long-term scheduling* (admission) : determining whether to add to the set of processes to be executed
- *Medium-term scheduling*: determining whether to add to the number of processes partially or fully in memory (degree of multiprogramming)
- *Short-term scheduling*: determining which process will be executed by the processor
- *I/O scheduling*: determining which process's pending I/O request will be handled by an available I/O device

# *CPU Scheduling Views*

- Single process view
  - GUI (graphical user interface) request
    - click on the mouse (*responsiveness*)
  - Scientific computation
    - long-running, but want to complete ASAP (*time to solution*)
- System view (objectives)
  - Get as many tasks done as quickly as possible
    - *throughput* objective
  - Minimize waiting time for processes
    - *response time* objective
  - Get full utilization from the CPU
    - *utilization* objective

# *Process Scheduling*

- Process transition diagram
- Think about the OS perspective



# *When does scheduling occur?*

- CPU scheduling decisions may take place when:
  1. A process switches from running to waiting state
  2. A process switches from running to ready state
  3. A process switches from waiting to ready
  4. A process terminates
- Process voluntarily gives up (yields) the CPU
  - CPU scheduler kicks in an decides who to go next
  - Process gets put on the ready queue
  - It could get immediately re-scheduled to run

# *Scheduling Problem*

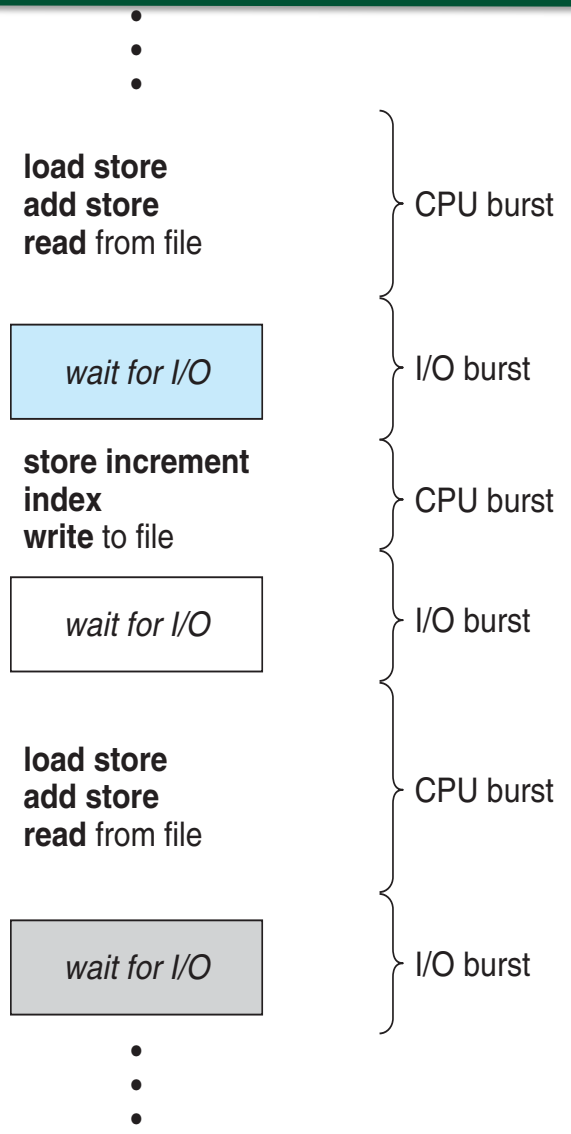
- Choose the ready/running process to run at any time
  - Maximize "performance"
- Model (estimate) "performance" as a function
  - System performance of scheduling each process
    - *f(process) = y*
  - What are some choices for *f(process)*?
- Choose the process with the best *y*
  - Estimating overall performance is intractable
  - Scheduling so all tasks are completed as soon as possible is a NP-complete problem!
  - Adding in pre-emption does not help (as we will see)

# *Preemption*

- Can we reschedule a process that is actively running (i.e., preempt its execution)?
  - If so, we have a *preemptive* scheduler
  - If not, we have a *non-preemptive* scheduler
- Suppose a process becomes ready
  - A new process is created or it is no longer waiting
- There is a currently running process
- However, it may be "better" (whatever this means) to schedule the process just put on the ready queue
  - So, we have to preempt the running process
- In what ways could the new process be better?

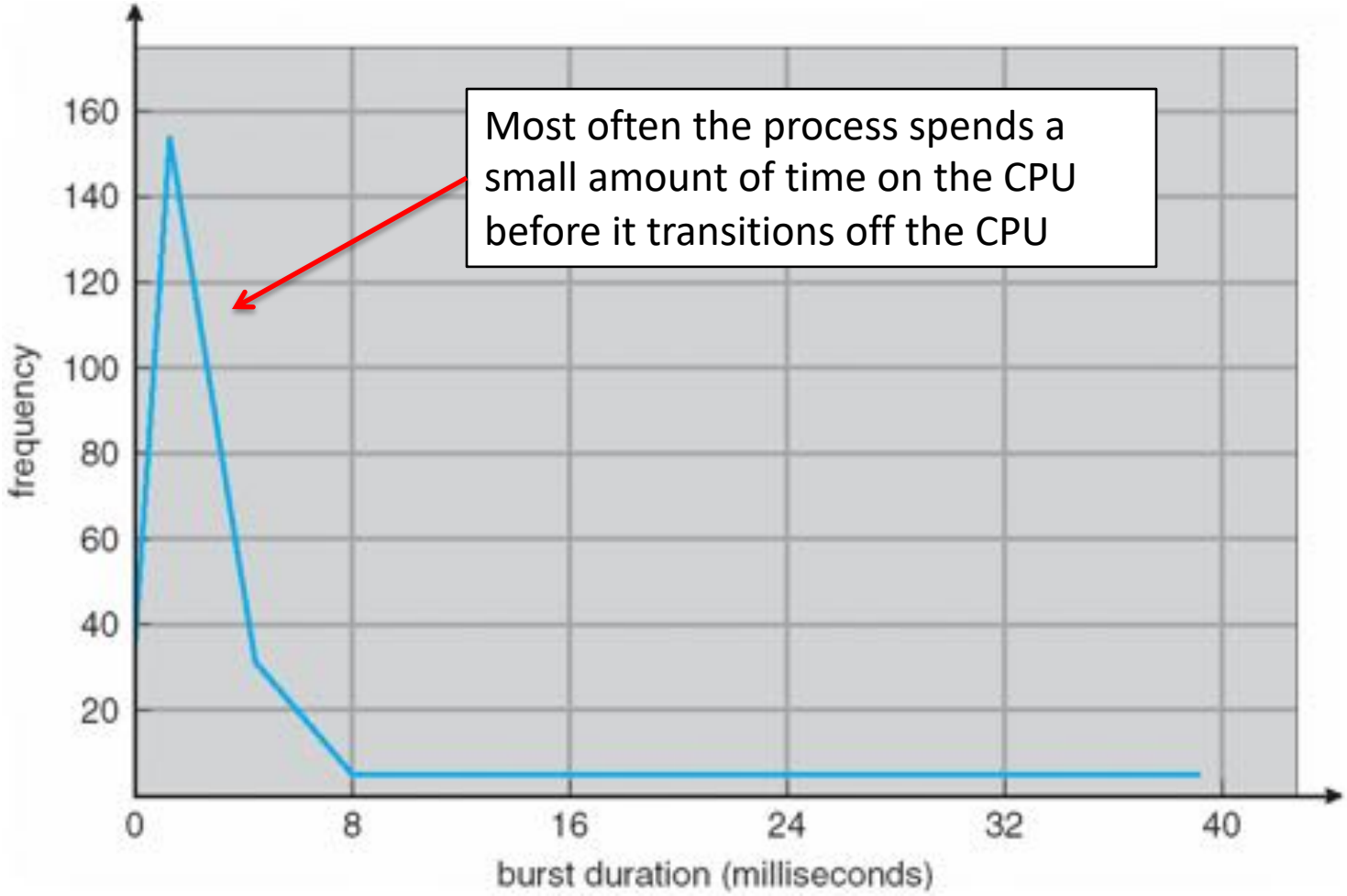
# *Basic Concepts – CPU-I/O Bursts*

- Maximum CPU utilization is obtained with multiprocessing … Why?
- *CPU–I/O burst cycle*
  - Process execution consists of cycles of CPU execution and I/O wait
    - run instructions (CPU burst)
    - wait for I/O (I/O burst)
- CPU burst distributionH
  - How much a CPU is used during a burst?
  - This is a main concern … Why?
- Scheduling is aided by knowing the length of these bursts
  - Hmm, do we know this? … more later



# *Histogram of CPU Burst Times*

- Profile for a particular process



# *Dispatcher*

- Dispatcher module gives control of the CPU to the process selected by the *short-term* scheduler
- This involves:
  - Switching context
    - save context of running process
    - loading process context of selected process to run
  - Switching to user mode
  - Jumping to the proper location in the user program to continue execution of the program
- *Dispatch latency* – time it takes for the dispatcher to stop one process and start another running
  - Context switch time

# *Scheduling Criteria*

- *Utilization/efficiency*
  - Keep the CPU busy 100% of the time with useful work
- *Throughput*
  - Maximize the number of jobs processed per hour.
- *Turnaround time (latency)*
  - From the time of submission to the time of completion.
- *Waiting time*
  - Sum of time spent (in ready queue) waiting to be scheduled on the CPU
- *Response time*
  - Time from submission until the first response is produced (mainly for interactive jobs)
- *Fairness*
  - Make sure each process gets a fair share of the CPU

# *Scheduling Algorithm Optimization Criteria*

- Max CPU utilization
- Max throughput
- Min turnaround time
- Min waiting time
- Min response time

# *Scheduling Algorithms*

- Some may seem intuitively better than others
- But a lot has to do with the type of offered *workload* to the processor
- Best scheduling comes with best context of the tasks to be completed
  - Knowing something about the workload behavior is important
- Distinguish between non-preemptive and preemptive cases
  - This has to do with whether a running process can be stopped during its execution and put back on the ready queue so that another process can acquire the CPU and run

# *First-Come, First-Served (FCFS)*

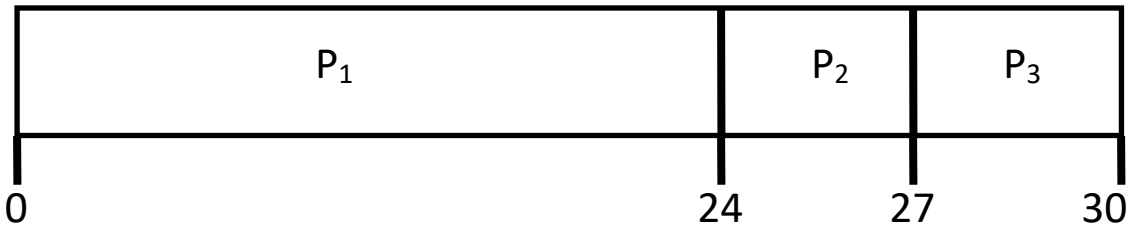
- Serve the jobs in the order they arrive
- Non-preemptive
  - Process is run until it has to wait or terminates
  - OS can not stop the process and put it on ready queue
- Simple and easy to implement
  - When a process is ready, add it to tail of ready queue, and serve the ready queue in FCFS order
- Very fair
  - No process is starved out
  - Service order is immune to job size (does not depend)
  - It depends only on *time of arrival*

# *FCFS*

| Process | Burst Time |
|---|---|
| $P_1$ | 24 |
| $P_2$ | 3 |
| $P_3$ | 3 |

Burst time here represents the job's entire execution time

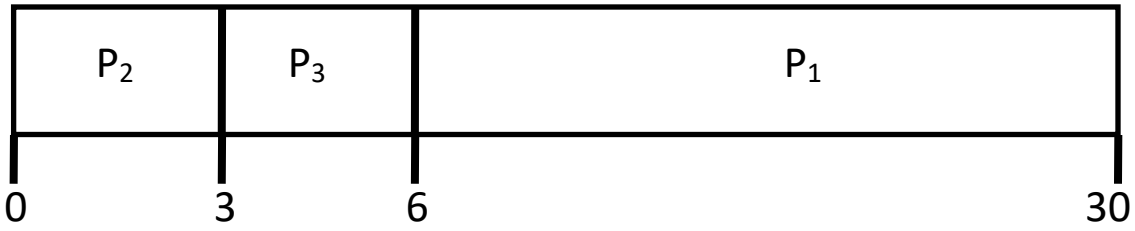
- Suppose that the processes arrive in the order: $P_1$, $P_2$, $P_3$
  - Assume processes arrive at the same time (e.g., time 0)
- The *Gantt chart* for the schedule is:




- Waiting time for $P_1 = 0$; $P_2 = 24$; $P_3 = 27$
- Average waiting time: $(0 + 24 + 27)/3 = 17$

# *Reducing Waiting Time*

- Suppose processes arrive in a different order: $P_2$, $P_3$, $P_1$
  - Again, assume that they arrive at the same time
- The Gantt chart for the schedule is:




- Waiting time for $P_1 = 6$; $P_2 = 0$; $P_3 = 3$
- Average waiting time: $(6 + 0 + 3)/3 = 3$
- Much better than previous case … Why?
- *Convoy effect*: short processes gets placed behind long process in the scheduling order

# *Shortest-Job-First (SJF)*

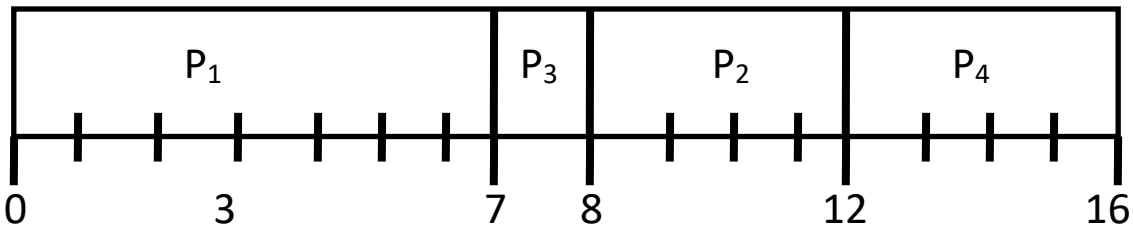
- Suppose we know length of *next* CPU burst
  - Hmm, do we know this for a process? (think about this …)
- Then use these lengths to schedule the process
  - Process with the shortest next CPU burst time goes first
- Two schemes:
  - *Non-preemptive* – once CPU given to the process it cannot be preempted until it completes its CPU burst
  - *Preemptive* – if a new process arrives with CPU burst length less than remaining time of current executing process, then preempt
    - known as the *Shortest-Remaining-Time-First (SRTF)*
- SJF is optimal
  - Gives minimum average waiting time for a set of processes
  - So we should always use it, right?

# *Non-Preemptive SJF*

| Process | Arrival Time | Burst Time |
|---|---|---|
| $P_1$ | 0.0 | 7 |
| $P_2$ | 2.0 | 4 |
| $P_3$ | 4.0 | 1 |
| $P_4$ | 5.0 | 4 |

- SJF (non-preemptive)

Scheduler makes a decision at the time when the next job is to be scheduled



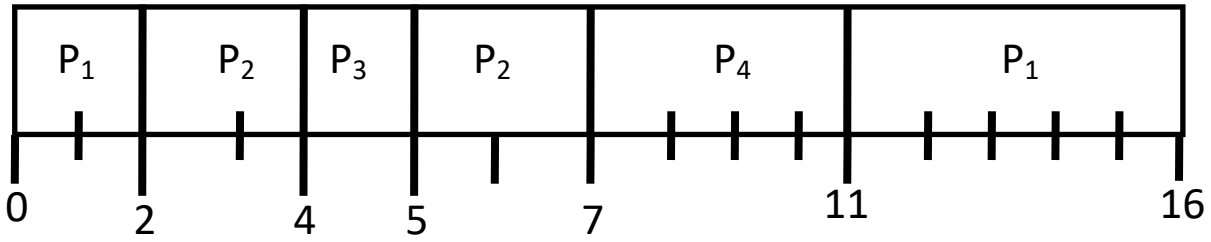

- Average waiting time = (0 + 6 + 3 + 7)/4 = 4

# *Preemptive SJF*

| Process | Arrival Time | Burst Time |
|---|---|---|
| $P_1$ | 0.0 | 7 |
| $P_2$ | 2.0 | 4 |
| $P_3$ | 4.0 | 1 |
| $P_4$ | 5.0 | 4 |

Scheduler makes a decision at any time using preemption to stop the currently running process

- SJF (preemptive)




- Average waiting time = (9 + 1 + 0 +2)/4 = 3

# *Determining Length of Next CPU Burst*

- Can only estimate the length (do not know for sure)
  - Hmm, maybe it is similar to the previous one
  - Thus pick process with shortest "predicted" next CPU burst
- Can be done by using the *length* of previous CPU bursts, for instance, using *exponential averaging*

*Given:* $t_n$ = *actual length of the nth CPU busrt*

$\tau_n$ = *predicted value for the nth CPU burst*

$\tau_n+1$ = *predicted value for the next CPU burst*

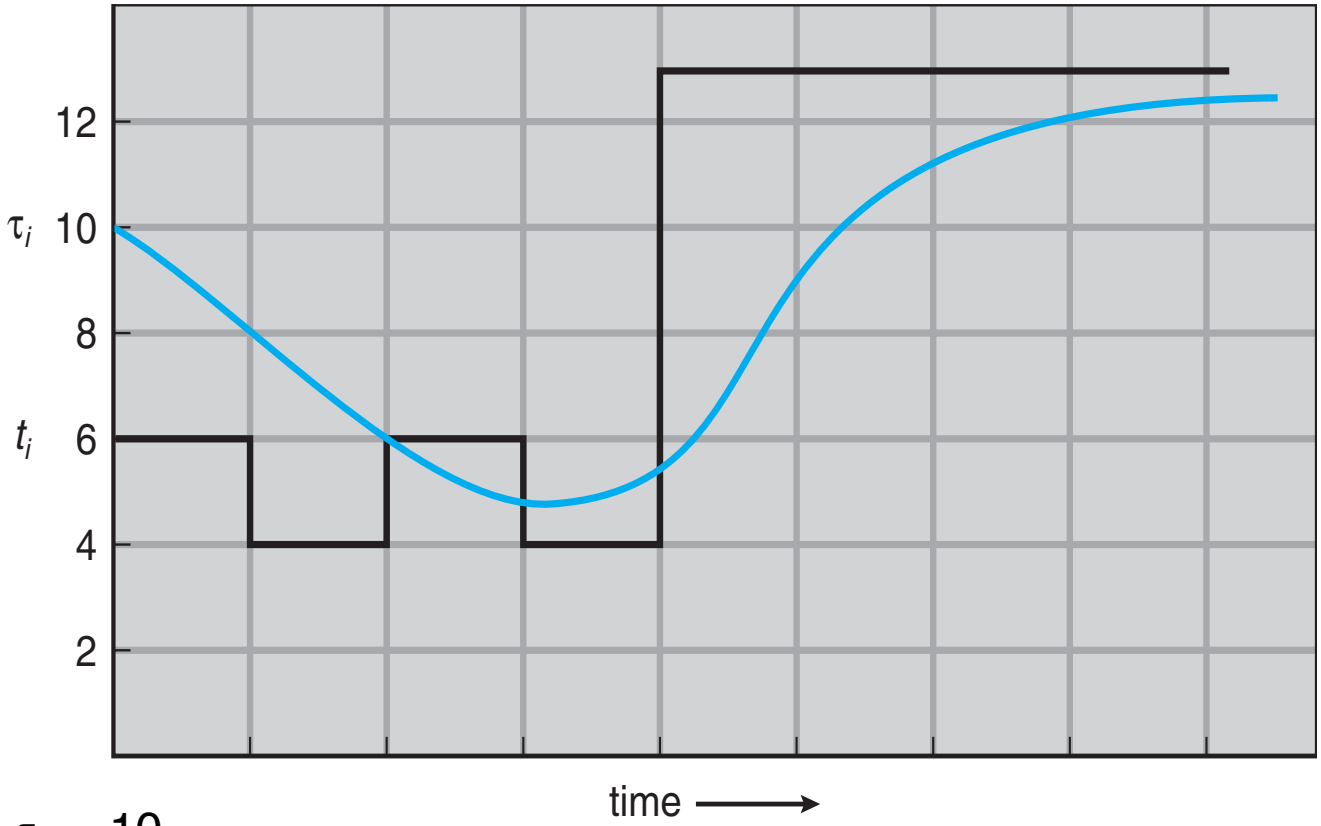
$\alpha$ *such that* $0 < \alpha < 1$

*Define:* $\tau_{n+1} = \alpha * t_n + (1 - \alpha) * \tau_n$

# *Exponential Averaging*

- What to set α to?
  - α = 0? Recent history does not count
  - α = 1? Only the last CPU burst counts
  - Commonly, α set to ½
- Since both α and (1 - α) are less than or equal to 1, each successive term has less weight than its predecessor
- SRTF is the preemptive version

# CPU Burst Prediction




$\alpha = ½$ and $\tau_0 = 10$

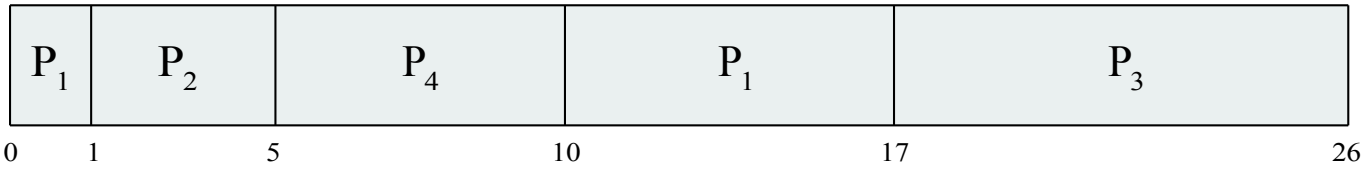
| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| CPU burst ($t_i$) | | 6 | 4 | 6 | 4 | 13 | 13 | 13 | ... |
| "guess" ($\tau_i$) | 10 | 8 | 6 | 6 | 5 | 9 | 11 | 12 | ... |

# *Example of Shortest-remaining-time-first*

- Now we add the concepts of varying arrival times and preemption to the analysis

| Process | Arrival Time | Burst Time |
|---|---|---|
| $P_1$ | 0 | 8 |
| $P_2$ | 1 | 4 |
| $P_3$ | 2 | 9 |
| $P_4$ | 3 | 5 |

- Preemptive SJF Gantt Chart




- Average waiting time = [(10-1)+(1-1)+(17-2)+(5-3)]/4
  = 26/4 = 6.5 msec

# *Scheduling Algorithms*

- First-come, First-serve (FCFS)
  - Non-preemptive
  - Does not account for waiting time (or much else)
    - Convoy problem
- Shortest Job First
  - May be preemptive
  - Optimal for minimizing waiting time (how?)
- Lots more … What do real systems use?

# *Priority Scheduling*

- Each process is given a certain priority "value"
- Always schedule the process with highest priority
  - Preemptive
  - Non-preemptive
- Problems can occur
  - Low priority processes may never execute
  - Process *starvation*
- Use *aging* to address starvation
  - Process increase priority as as time progresses

# *Example of Priority Scheduling*

| Process | Burst Time | Priority |
|---|---|---|
| $P_1$ | 10 | 3 |
| $P_2$ | 1 | 1 |
| $P_3$ | 2 | 4 |
| $P_4$ | 1 | 5 |
| $P_5$ | 5 | 2 |

- Priority scheduling Gantt Chart (non-preemptive)


- Average waiting time = 8.2 msec

# *Priorities*

- Note that FCFS and SJF are specialized versions of *Priority Scheduling*
  - Assigning priorities to the processes in a certain way
- What would the priority function be for FCFS?
- What would the priority function be for SJF?

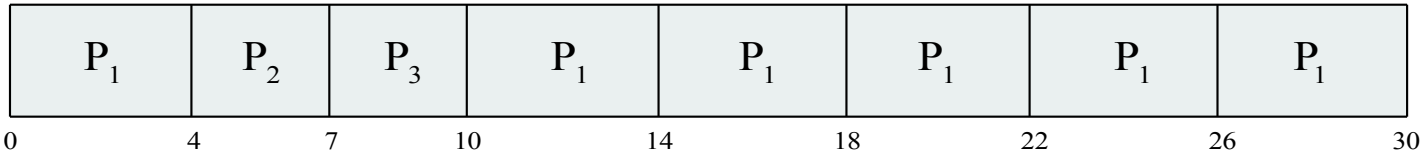
# *Round Robin (RR)*

- Each process gets a small unit of CPU time (time quantum)
  - Usually 10-100 milliseconds
  - After this time has elapsed, the process is *preempted* and added to the end of the ready queue
- Approach
  - Consider *n* processes in the ready queue
  - Consider time quantum is *q*
  - Then each process gets *1/n* of the CPU time
  - In chunks of at most *q* time units at once.
  - No process waits more than *(n-1)q* time units

# *Example of RR with Time Quantum = 4*

| Process | Burst Time |
|---|---|
| $P_1$ | 24 |
| $P_2$ | 3 |
| $P_3$ | 3 |

- The Gantt chart is:

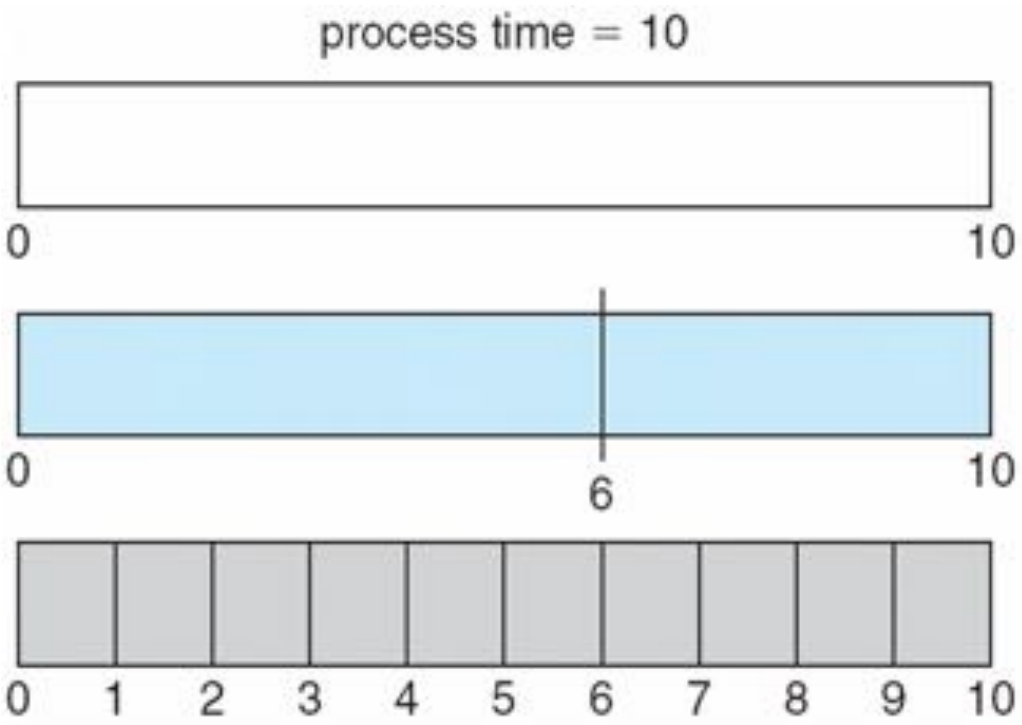



- Typically, higher average turnaround than SJF, but better response times
- *q* should be large compared to context switch time
  - Usually *q* is between 10ms to 100ms
  - Context switch < 10 usec

# *RR Time Quantum*

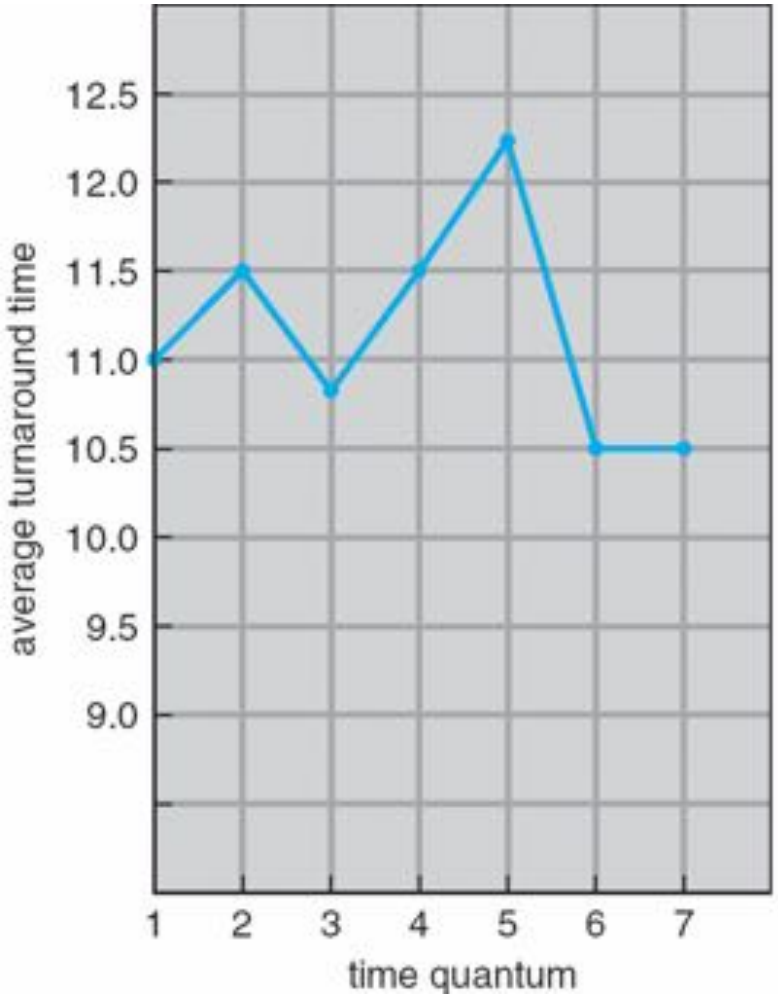
- Round robin allows the CPU to be virtually shared between the processes
  - Each process has the illusion that it is running in isolation (at *1/n*-th the CPU speed)
- Smaller time quantums make this illusion more realistic, but there are problems
  - What is the main problem?
- Larger time quantums will give more preference to processes with larger burst times
  - What scheduling algorithm is approximated when quantums are very large?

# *Time Quantum and Context Switch Time*




| quantum | context switches |
|---|---|
| 12 | 0 |
| 6 | 1 |
| 1 | 9 |

# Turnaround Time Varies With Time Quantum




| process | time |
|---|---|
| $P_1$ | 6 |
| $P_2$ | 3 |
| $P_3$ | 1 |
| $P_4$ | 7 |

80% of CPU bursts should be shorter than q

# *RR Time Quantum*

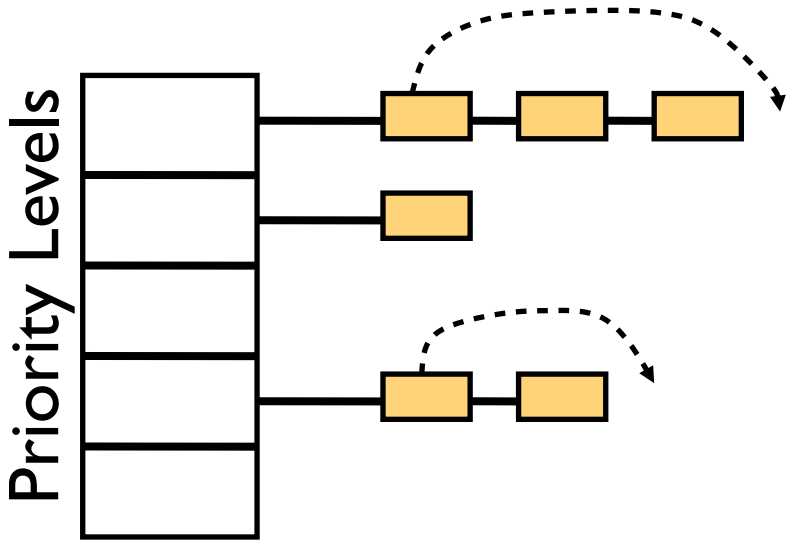
- If time quantum size decreases, what happens to the context switches?
- Context switches are not free!
  - Saving/restoring registers
  - Switching address spaces
  - Indirect costs (cache pollution)

# *Scheduling Desirables*

- SJF
  - Minimize waiting time
    - requires estimate of CPU bursts
- Round robin
  - Share CPU via time quanta
    - if burst turns out to be "too long"
- Priorities
  - Some processes are more important
  - Priorities enable composition of "importance" factors
- No single best approach – now what?

# *Round Robin with Priority*

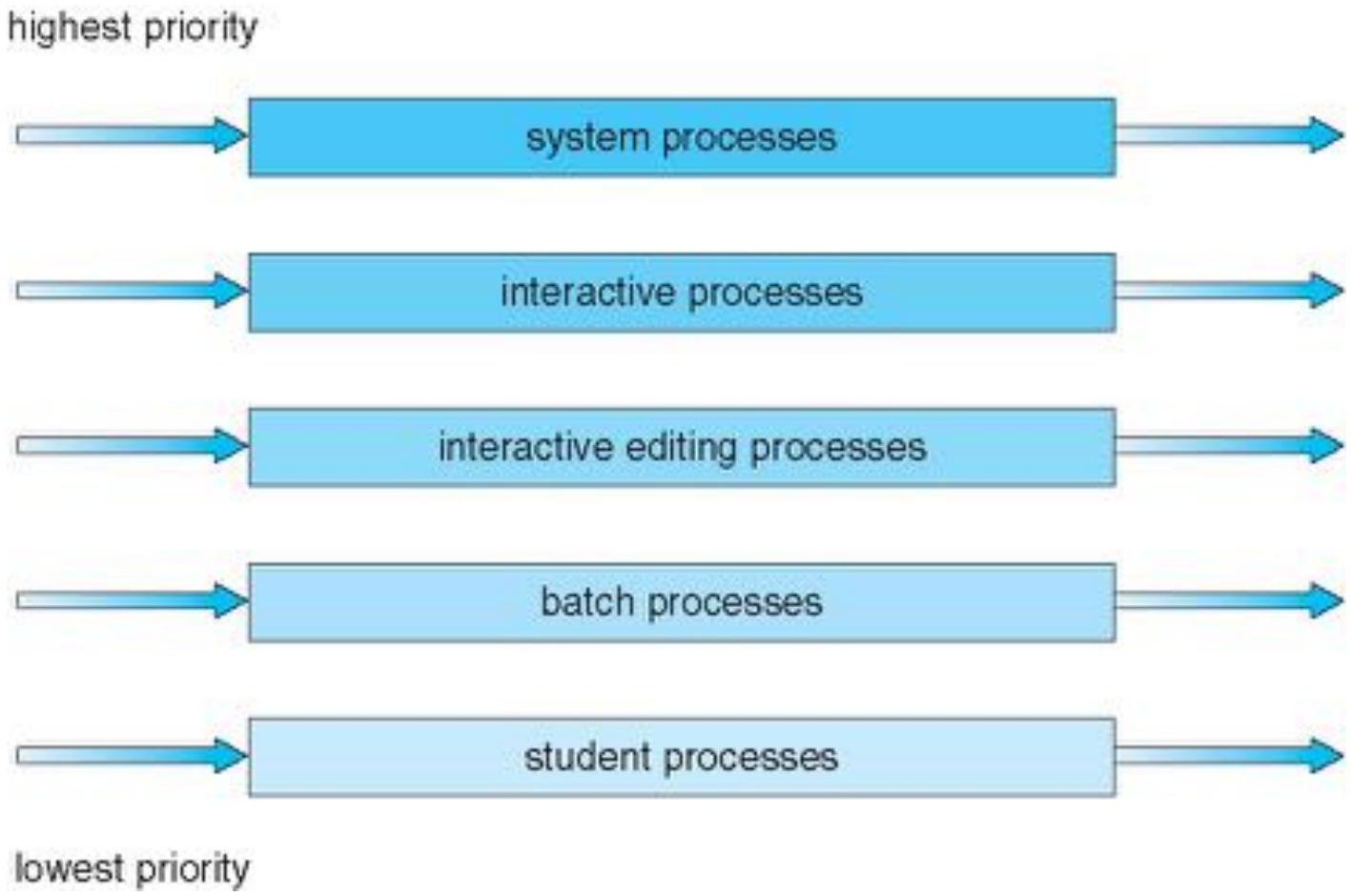
- Have a ready queue for each priority level
- Always service the non-null queue at the highest priority level
- Within each queue, you perform round-robin scheduling between those processes
- Problems?
  - With fixed priorities, processes lower in the priority level can get *starved out!*
  - In general, you employ a mechanism to "age" the priority of processes



# *Multilevel Queue*

- Ready queue is partitioned into separate queues:
  foreground (*interactive*)
  background (*batch*)
- Each queue has its own scheduling algorithm:
  foreground – RR
  background – FCFS
- Scheduling must be done between the queues
  - Fixed priority scheduling
    - possibility of starvation
  - Time slice
    - each queue gets a certain amount of CPU time which it can schedule amongst its processes

# *Multilevel Queue Scheduling*

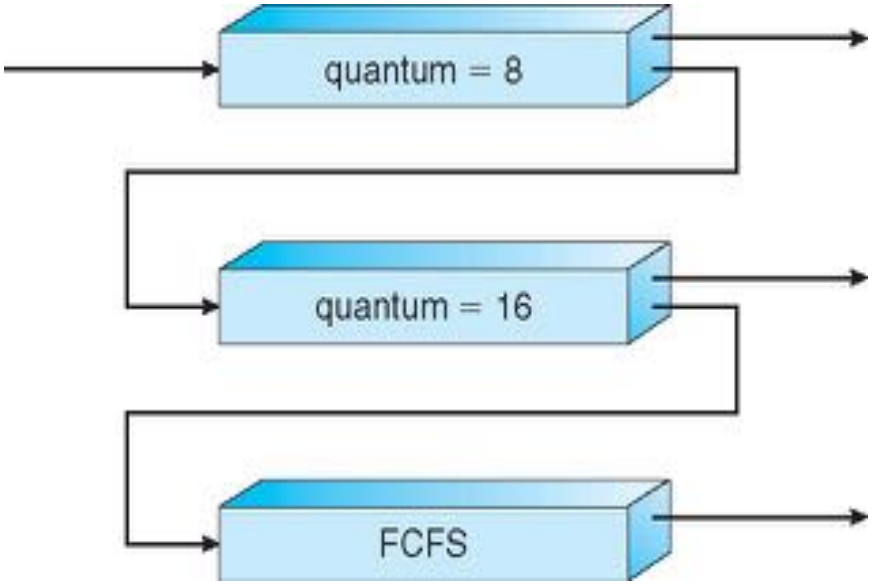


# *Multilevel Feedback Queue*

- A process can move between the various queues; aging can be implemented this way
- Multilevel-feedback-queue scheduler defined by the following parameters:
  - Number of queues
  - Scheduling algorithms for each queue
  - Method used to determine when to upgrade a process
  - Method used to determine when to demote a process
  - Method used to determine which queue a process will enter when that process needs service

# *Example of Multilevel Feedback Queue*

- Three queues:
  - $Q_0$ – RR with time quantum 8 milliseconds
  - $Q_1$ – RR time quantum 16 milliseconds
  - $Q_2$ – FCFS




- Scheduling
  - A new job enters queue $Q_0$ which served FCFS
    - when it gains CPU, job receives 8 milliseconds
    - if it does not finish in 8 milliseconds, job is moved to queue $Q_1$
  - At $Q_1$ job is again served FCFS and receives 16 additional milliseconds
    - if it still does not complete, it is preempted and moved to queue $Q_2$

# *Traditional UNIX Scheduling*

- Multilevel feedback queues
- 128 priorities possible (-64 to +63)
- 1 Round Robin queue per priority
- Every scheduling event the scheduler picks the highest priority (lowest number) non-empty queue and runs jobs in round-robin

# *UNIX Process Scheduling*

- Negative numbers reserved for processes waiting in kernel mode (just woken up by interrupt handlers) (why do they have a higher priority?)
- Time quantum = 1/10 sec (empirically found to be the longest quantum that could be used without loss of the desired response for interactive jobs such as editors)
  - Short time quantum means better interactive response
  - Long time quantum means higher overall system throughput since less context switch overhead and less processor cache flush.
- Priority dynamically adjusted to reflect
  - Resource requirement (e.g., blocked awaiting an event)
  - Resource consumption (e.g., CPU time)

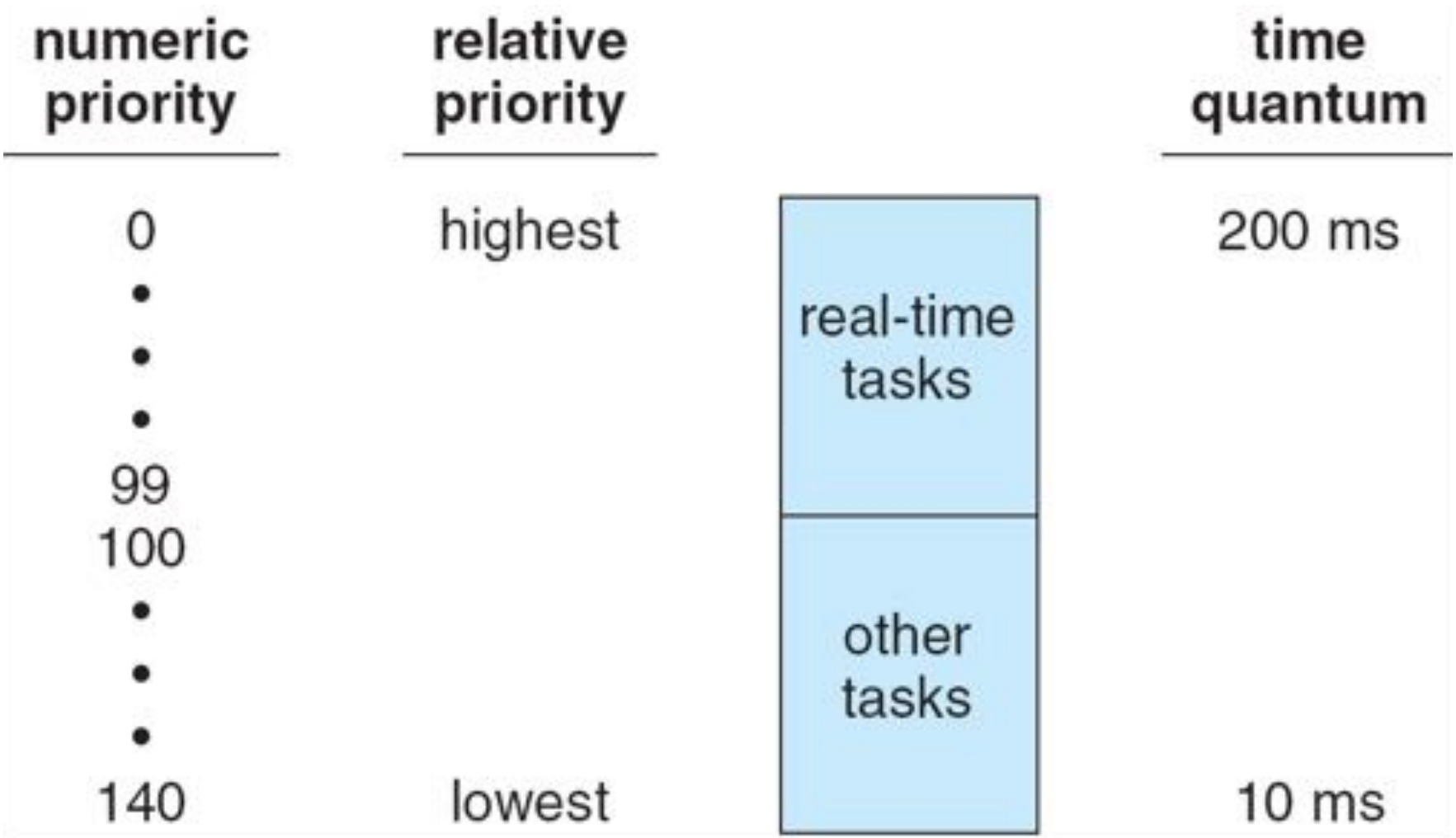
# *Linux Scheduler*

- Kernel 2.4 and earlier: essentially the same as the traditional UNIX scheduler
- Kernel 2.6: O(1) scheduler
  - Time to select process is constant regardless of system load or the number of processors
  - Separate queue for each priority level
  - CPU affinity (keeps processes on same CPU)
- More recently (kernel 2.6.23 and up): CFS
  - Completely Fair Scheduler (runs O(log N))
    - uses red-black trees rather than runqueues

# *Linux Scheduling*

- Two algorithms: time-sharing and real-time
- Time-sharing (still abstracted)
  - Two queues: active and expired
  - In active, until you use your entire time slice (quantum), then expired
    - once in expired, wait for all others to finish (fairness)
  - Priority recalculation -- based on waiting vs. running time
    - from 0-10 milliseconds
    - add waiting time to value, subtract running time
    - adjust the static priority
- Real-time
  - Soft real-time
  - Posix.1b compliant – two classes
    - FCFS and RR (highest priority process always runs first)

# *Priorities and Time-Slice Length*



# *Thread Scheduling*

- When threads are supported, it is threads that are scheduled, not processes
  - Distinction between user-level and kernel-level threads
- Many-to-one and many-to-many models, thread library schedules user-level threads to run on LWP
  - Known as process-contention scope (PCS) since scheduling competition is within the process
  - Typically done via priority set by programmer
- Kernel thread scheduled onto available CPU is system-contention scope (SCS) – competition among all threads in system

# *Multiple-Processor Scheduling*

- CPU scheduling is more complex with multiple CPUs
- Consider *homogeneous* processors within a multiprocessor
- *Asymmetric* multiprocessing
  - Only one processor accesses the system data structures, alleviating the need for data sharing
  - Some parts of the OS only runs on this processor
- *Symmetric* multiprocessing (SMP)
  - Each processor is self-scheduling, all processes in common ready queue, or each has its own private ready queue
  - Currently, most common approach
- Processor affinity
  - Process favors the processor on which it is currently running

# *Multiple-Processor Scheduling*

- Need to keep all CPUs loaded for efficiency
- Load balancing attempts to balance the workload
- Push migration
  - Periodic task checks load on each processor
  - Pushes task from overloaded CPU to other CPUs
- Pull migration
  - Idle processors pulls waiting task from busy processor

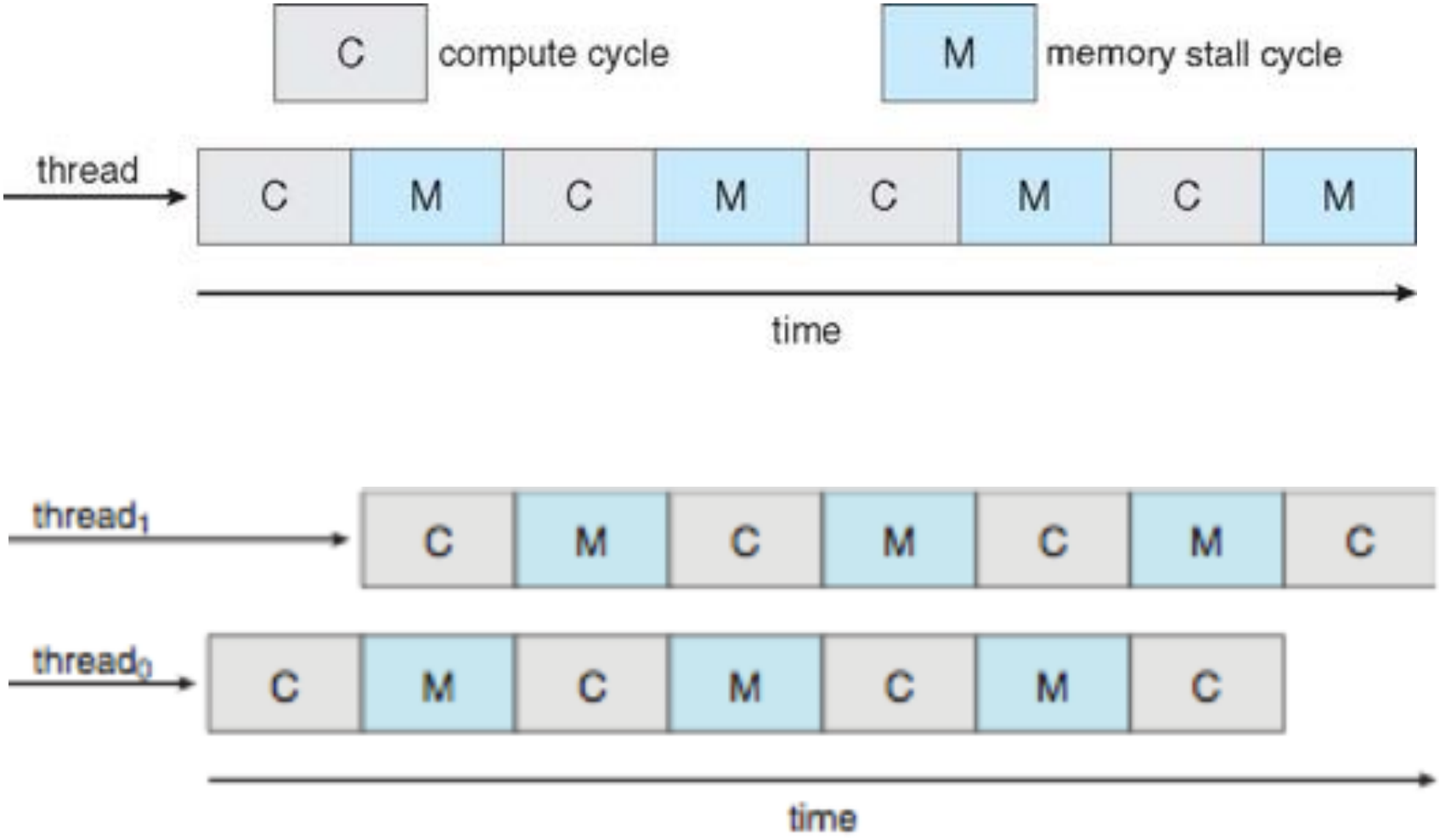
# *Real-time Scheduling*

- *Hard real-time* systems – required to complete a critical task within a guaranteed amount of time
- *Soft real-time* computing – requires that critical threads receive priority over less critical ones

# *Multicore Processors*

- Recent trend to place multiple processor cores on same physical chip
- Faster and consumes less power
- Multiple threads per core also growing
  - Takes advantage of memory stall to make progress on another thread while memory retrieve happens

# *Multithreaded Multicore System*

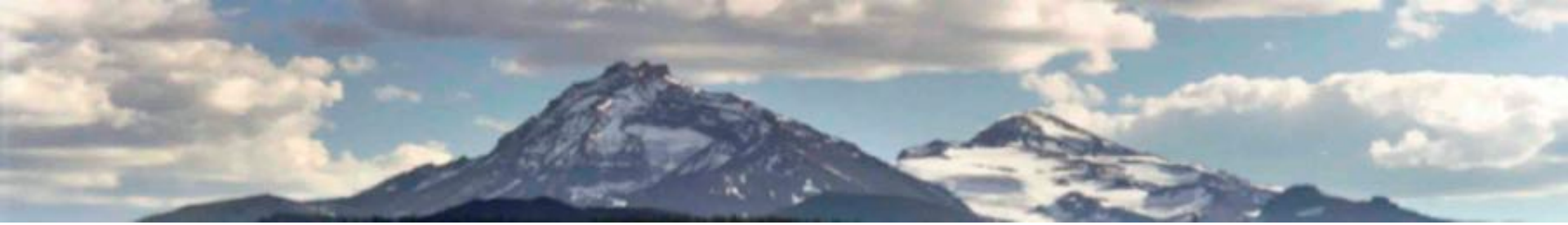


# *Summary*

- CPU Scheduling
  - Algorithms
  - Combination of algorithms
    - Multi-level Feedback Queues
- Scheduling Systems
  - UNIX
  - Linux

# *Next Class*

- Concurrency and synchronization!



# Extra Slides