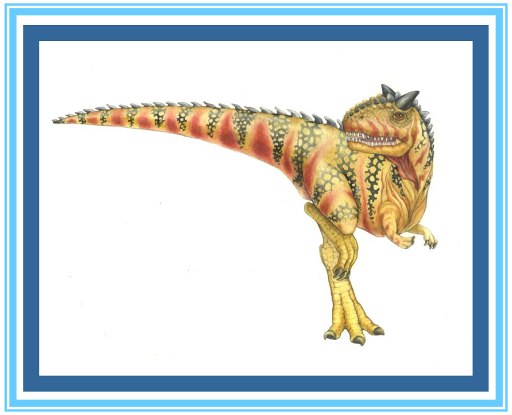
Lecture 5: CPU Scheduling



What Learned So Far?

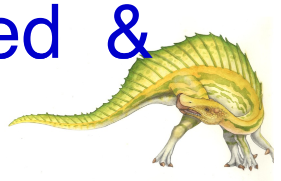
- Lecture 1
  - Role of OS
  - Multi-programming vs. multi-tasking
  - Interrupt driven OS
  - Dual mode in OS
  - Process hogging (infinite loop)

What Learned So Far? (cont.)

- Lecture 2
  - OS services
  - System calls
  - Types of system calls
  - API
  - System programs
  - Various ways to structure OS: simple, layered, microkernel, modular, …
  - Hypervisor and VM

What Learned So Far? (cont.)

- Lecture 3
  - Process concept
  - Process state
  - Process control block (PCB)
  - Process scheduling
  - Short-term scheduler
  - Long-term scheduler
  - Process creation and termination
  - Inter-Process communication (IPC) (shared & message passing)



What Learned So Far? (cont.)

- Lecture 4
  - Thread concept
  - Single-thread vs. Multi-threaded process
  - User threads vs. kernel threads
  - Multi-threading models
  - Thread libraries
  - Issues with multi-threading

Lecture 5: CPU Scheduling

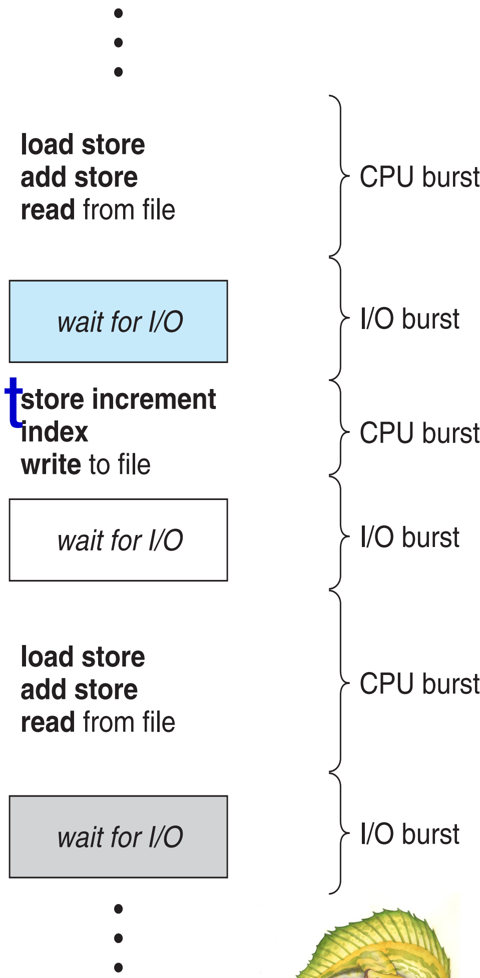
- Basic Concepts
- Scheduling Criteria
- Scheduling Algorithms
- Thread Scheduling
- Real-Time CPU Scheduling
- Multiple-Processor Scheduling
- Operating Systems Examples

Objectives

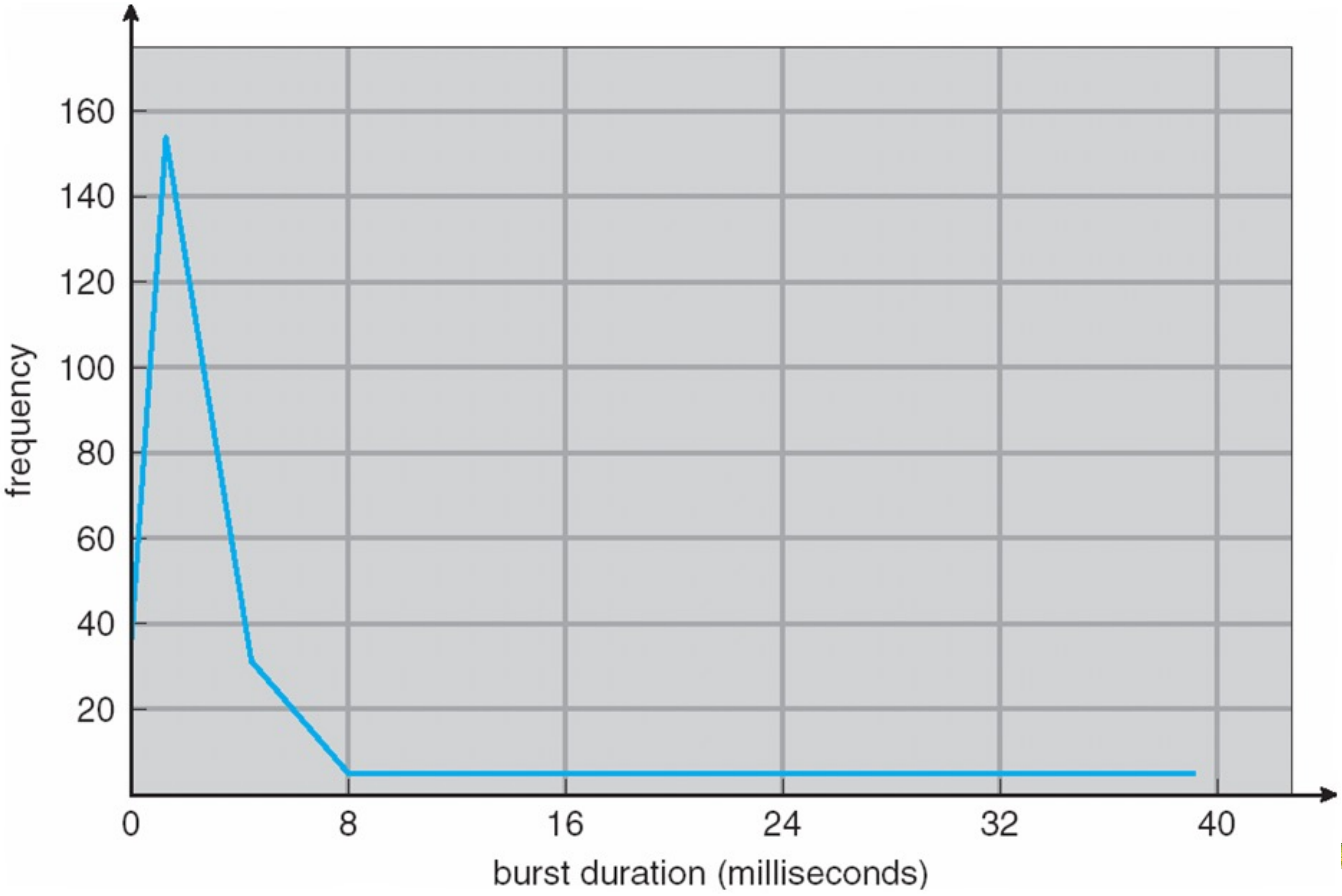
- To Introduce CPU Scheduling
  - Basis for multi-programmed OSs
- To Describe Various CPU-Scheduling Algorithms
- To Discuss Evaluation Criteria for Selecting a CPU-Scheduling Algorithm for a Particular System

Basic Concepts

- Maximum CPU Utilization Obtained with Multiprogramming
- CPU–I/O Burst Cycle
  - Process execution consists of a **cycle** of CPU execution and I/O wait
- **CPU Burst** followed by **I/O Burst**
- Major Concerns
  - CPU burst **Duration**
  - CPU burst **distribution**



Histogram of CPU-Burst Times

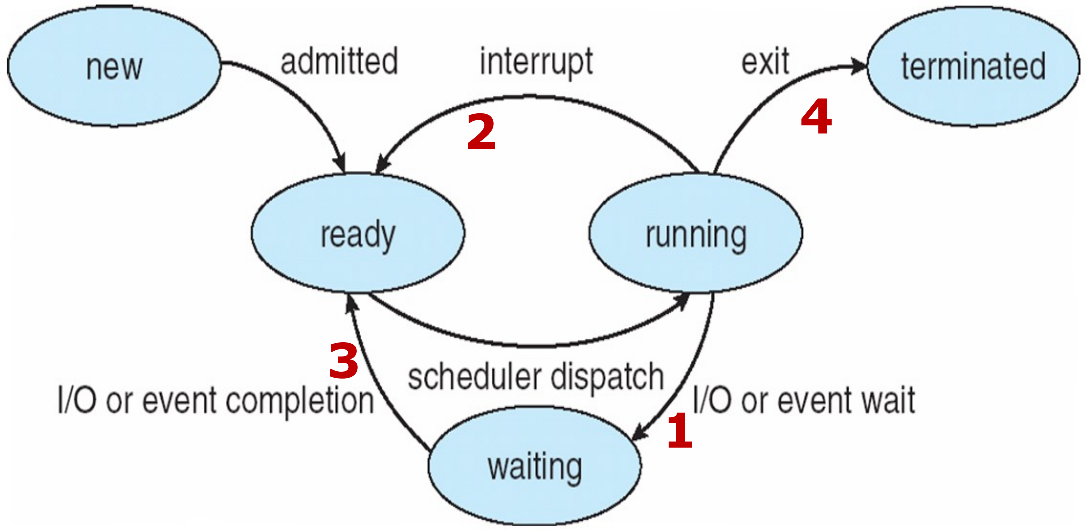


CPU Scheduler

- **Short-Term Scheduler** selects from among processes in ready queue, and allocates CPU to one of them
  - Queue may be ordered in various ways
    - Not necessarily a FIFO
- CPU Scheduling Decisions may Take Place when a process:
  1. Switches from running to waiting state
  2. Switches from running to ready state
  3. Switches from waiting to ready state
  4. Terminates

CPU Scheduler (cont.)

- **Non-Preemptive Scheduling**
  - Scheduling only under case 1 and case 4
  - **aka, cooperating scheduling**
  - Used in Windows 3.x
- All other Scheduling is **Preemptive**
  - Used in Win95/WinNT, Mac OS X



CPU Scheduler (cont.)

- **Non-Preemptive** Scheduling
  - Simple to implement
  - Low context switch flexibility
  - Process hogging very likely to happen
- **Preemptive** Scheduling
  - More flexibility in context switching
  - More complex to implement
  - More issues for implementation
    - Consider access to shared data
    - Consider preemption while in kernel mode
    - Consider interrupts occurring during crucial OS activities

Dispatcher

- Dispatcher Module
  - Gives Control of CPU to process Selected by short-term scheduler
- Dispatcher Involves:
  - Switching context
  - Switching to user mode
  - Jumping to proper location in user program to restart that program
- **Dispatch latency**
  - Time it takes for dispatcher to stop one process and start another running

Scheduling Criteria

- **CPU Utilization** – keep CPU as busy as possible
- **Throughput** – # of processes that complete their execution per time unit
- **Turnaround Time** – amount of time to execute a particular process
- **Waiting Time** – amount of time a process has been waiting in ready queue
- **Response Time** – amount of time it takes from when a request was submitted until first response is produced, not output (for time-sharing environment)

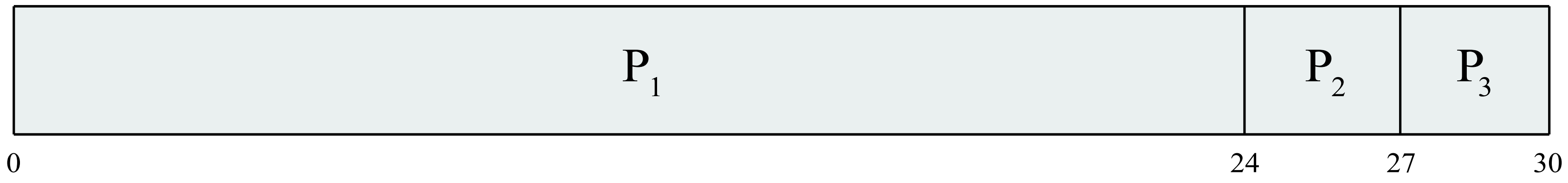
Scheduling Algorithm Optimization Criteria

- Max CPU Utilization
- Max Throughput
- Minimize Response Time
  - Average response time
  - Maximum response time
  - Variance of response time
- Minimize Turnaround Time
  - Average, maximum, variance
- Minimize Waiting Time
  - Average, maximum, variance

First- Come, First-Served (FCFS) Scheduling

Process	Burst Time
P_1	24
P_2	3
P_3	3

- Suppose: processes arrive in order: P_1 , P_2 , P_3
  The Gantt Chart for the schedule is:



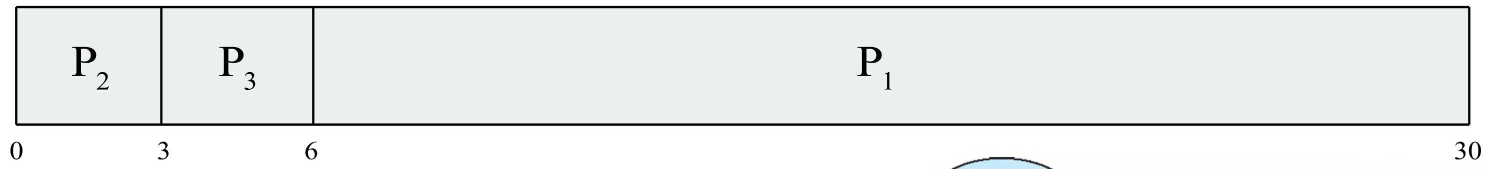
- Waiting time for P_1 = 0; P_2 = 24; P_3 = 27
- Average waiting time: (0 + 24 + 27)/3 = 17
- FCFS: non-preemptive ➔ not suited for time-sharing or real-time systems

FCFS Scheduling (Cont.)

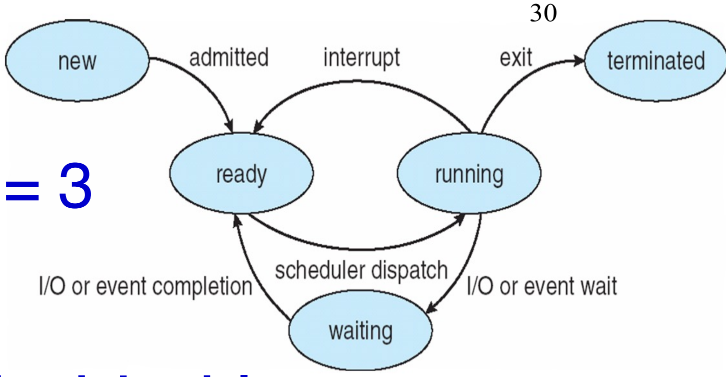
Suppose that processes arrive in following order:

$$P_2 , P_3 , P_1$$

- Gantt chart for this 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



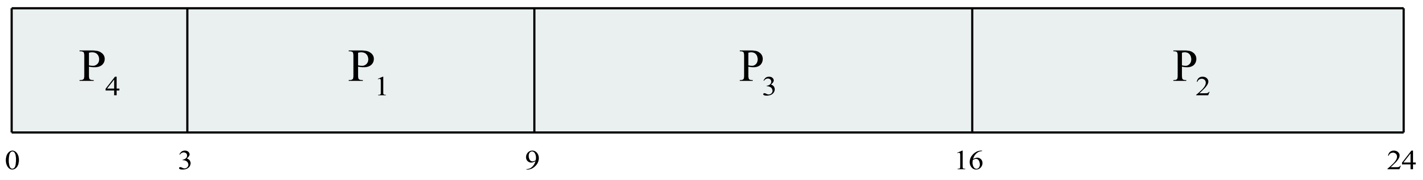
- **Convoy Effect:** short process behind long process
  - Consider one CPU-bound and many I/O-bound processes

Shortest-Job-First (SJF) Scheduling

- Associate with each Process Length of its Next CPU Burst
  - Use these lengths to schedule process with shortest time
- SJF is optimal
  - Gives minimum average waiting time for a given set of processes
  - Difficulty is knowing length of next CPU request
  - Possible starvation
    - Why?

Example of SJF

Process	Burst Time
P_1	6
P_2	8
P_3	7
P_4	3



- SJF Scheduling Chart
- Average waiting time = (3 + 16 + 9 + 0) / 4 = 7

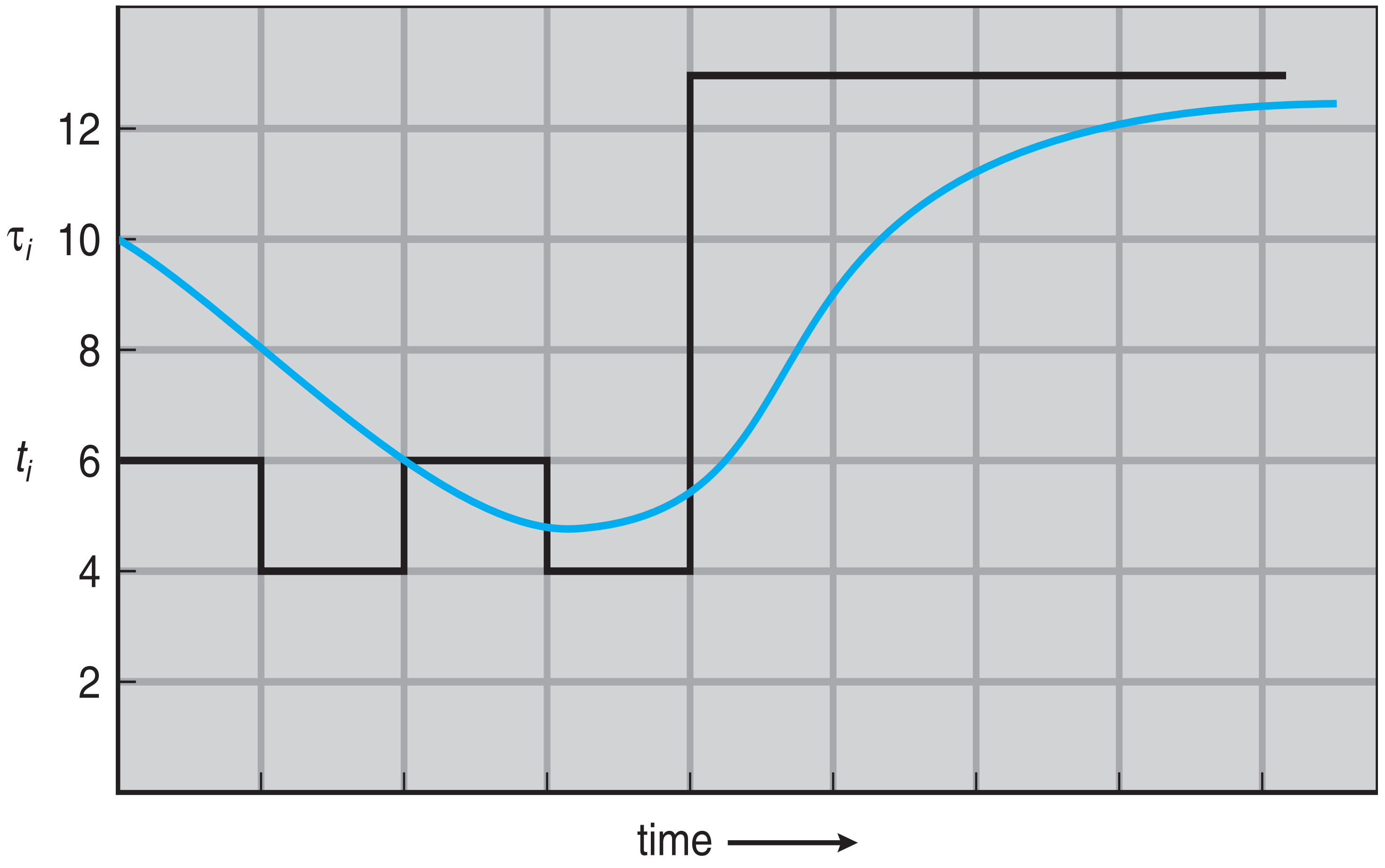
Determining Length of Next CPU Burst

- Can only Estimate Length – should be Similar to Previous one
  - Then pick process with shortest predicted next CPU burst

1. $t_n =$ actual length of n^{th} CPU burst
2. $\tau_{n+1} =$ predicted value for the next CPU burst
3. $\alpha, 0 \leq \alpha \leq 1$
4. Define: $\tau_{n+1} = \alpha\, t_n + (1-\alpha)\tau_n.$

- Can be done by using length of previous CPU bursts, using exponential averaging
- Commonly, α set to ½
- Preemptive version called **shortest-remaining-time-first**

Prediction of Length of Next CPU Burst



CPU burst (t_i)		6	4	6	4	13	13	13	…
"guess" (τ_i)	10	8	6	6	5	9	11	12	…

Examples of Exponential Averaging

- $\alpha = 0$
  - $\tau_{n+1} = \tau_n$
  - Recent history does not count
- $\alpha = 1$
  - $\tau_{n+1} = \alpha\, t_n$
  - Only actual last CPU burst counts

Examples of Exponential Averaging

- If we expand formula, we get:

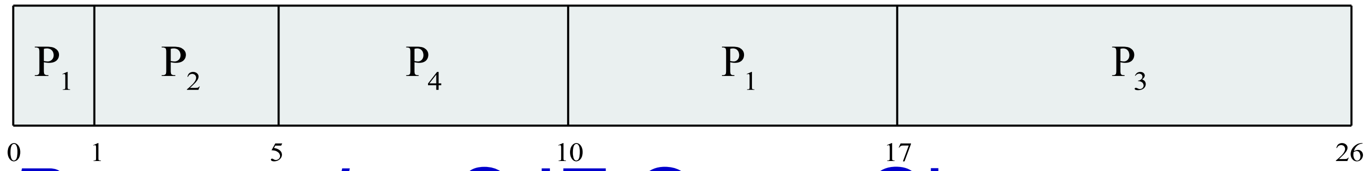
$$\tau_{n+1} = \alpha\, t_n + (1-\alpha)\alpha\, t_{n-1} + \ldots$$
$$+ (1-\alpha)^j \alpha\, t_{n-j} + \ldots$$
$$+ (1-\alpha)^{n+1} \tau_0$$

- Since both α and $(1 - \alpha)$ are less than or equal to 1, each successive term has less weight than its predecessor

Example of Shortest-remaining-time-first

- Varying Arrival Times and Preemption to Analysis

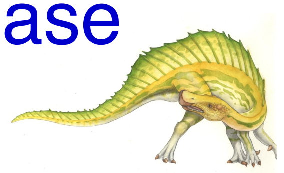
Process	*Arrival* Time	Burst Time
P_1	0	8
P_2	1	4
P_3	2	9
P_4	3	5



- *Preemptive* SJF Gantt Chart
  - Can have either preemptive or non-preemptive
- Average waiting time
  - Preemptive: [(10-1)+(1-1)+(17-2)+5-3)]/4 = 26/4 = 6.5 ms
  - Non-preemptive: 7.75 ms

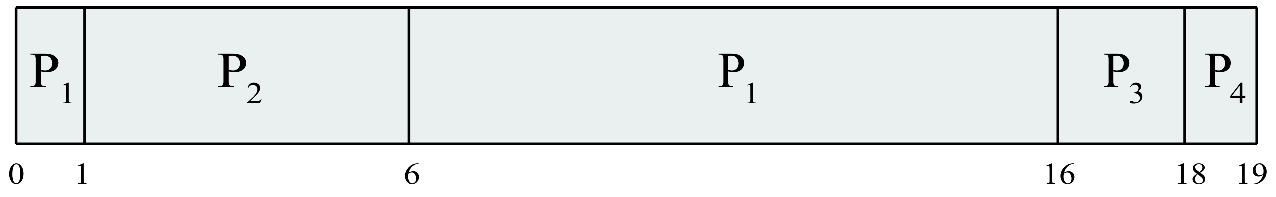
Priority Scheduling

- A Priority Number (Integer) is Associated with each Process
- CPU is allocated to Process with Highest Priority (smallest integer ≡ highest priority)
  - Preemptive
  - Non-preemptive
- **SJF** is Priority Scheduling where Priority is Inverse of Predicted Next CPU Burst Time
- Problem ≡ **Starvation** – low priority processes may never execute (e.g., IBM mainframe at MIT)
- Solution ≡ **Aging** – as time progresses increase process priority



Example of Priority Scheduling

Process	Burst Time	Priority
P_1	10	3
P_2	1	1
P_3	2	4
P_4	1	5



- Priority scheduling Gantt Chart
- Average waiting time = 8.2 msec (?)

Round Robin (RR)

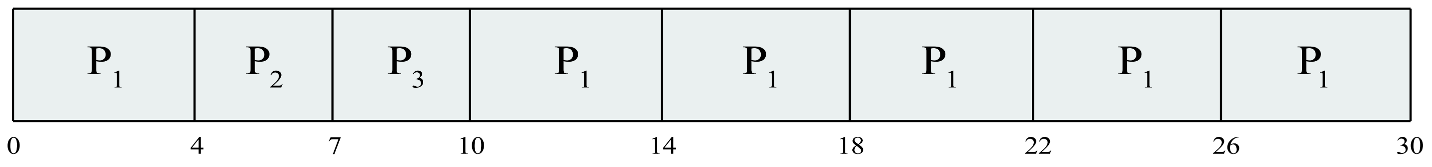
- Each Process Gets a Small Unit of CPU Time (**time quantum** *q*)
  - Usually 10-100 milliseconds
  - After this time has elapsed, process is preempted and added to end of ready queue
  - If there are *n* processes in ready queue and time quantum is *q*, then each process gets **1/*n*** of CPU time in chunks of at most *q* time units at once
  - No process waits more than $(n-1)q$ time units

Round Robin (RR) (cont.)

- Timer Interrupts every Quantum to Schedule next Process
- Performance
  - *q* very large ⇒ FIFO (FCFS)
  - *q* small ⇒ *q* must be large with respect to context switch, otherwise overhead is too high
- Real Example: Control Data Corporation (CDC)
  - 10 uP with only one set of HW and 10 sets of RF
    - Seems like 10 slow uPs rather than one fast uP
    - In reality, each slow uP provides better performance than one tenth of fast uP due to memory accesses

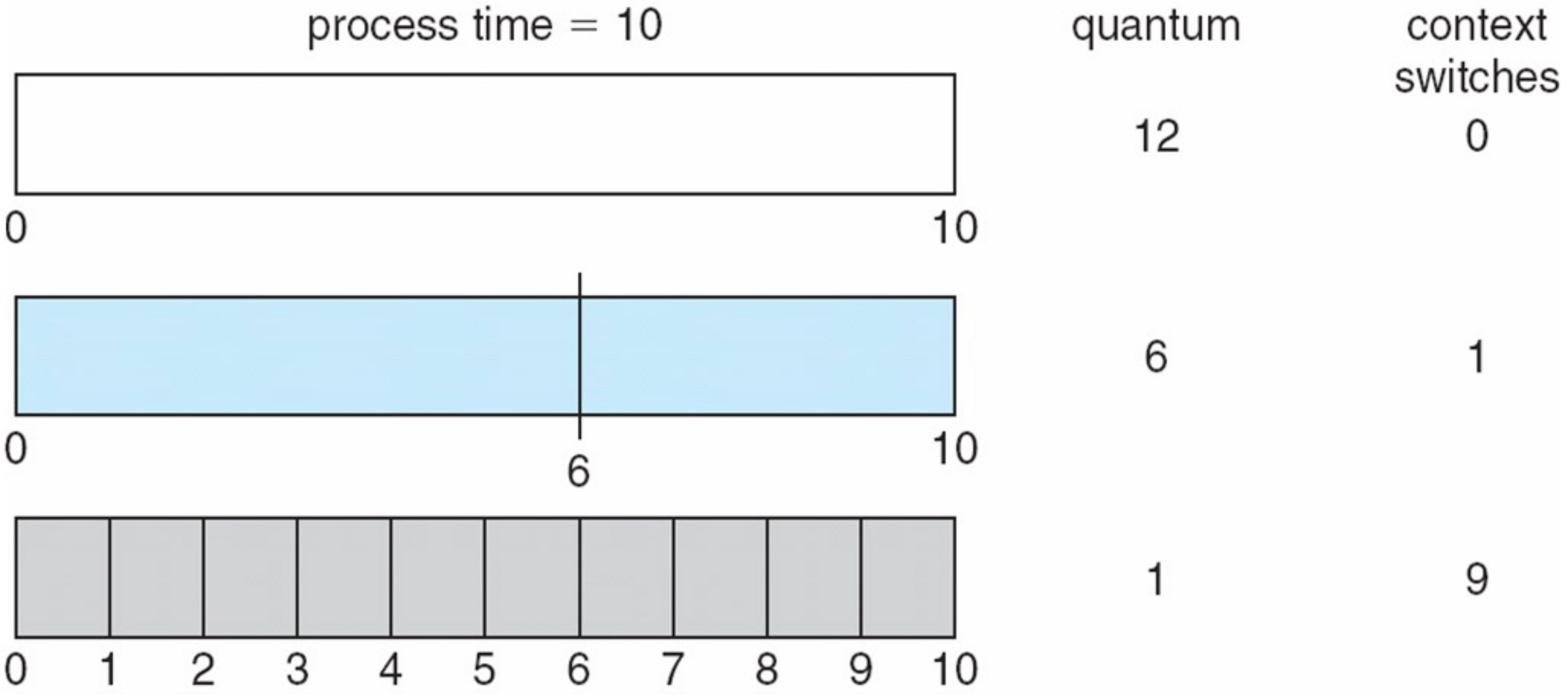
Example of RR with Time Quantum = 4

Process	Burst Time
P_1	24
P_2	3
P_3	3

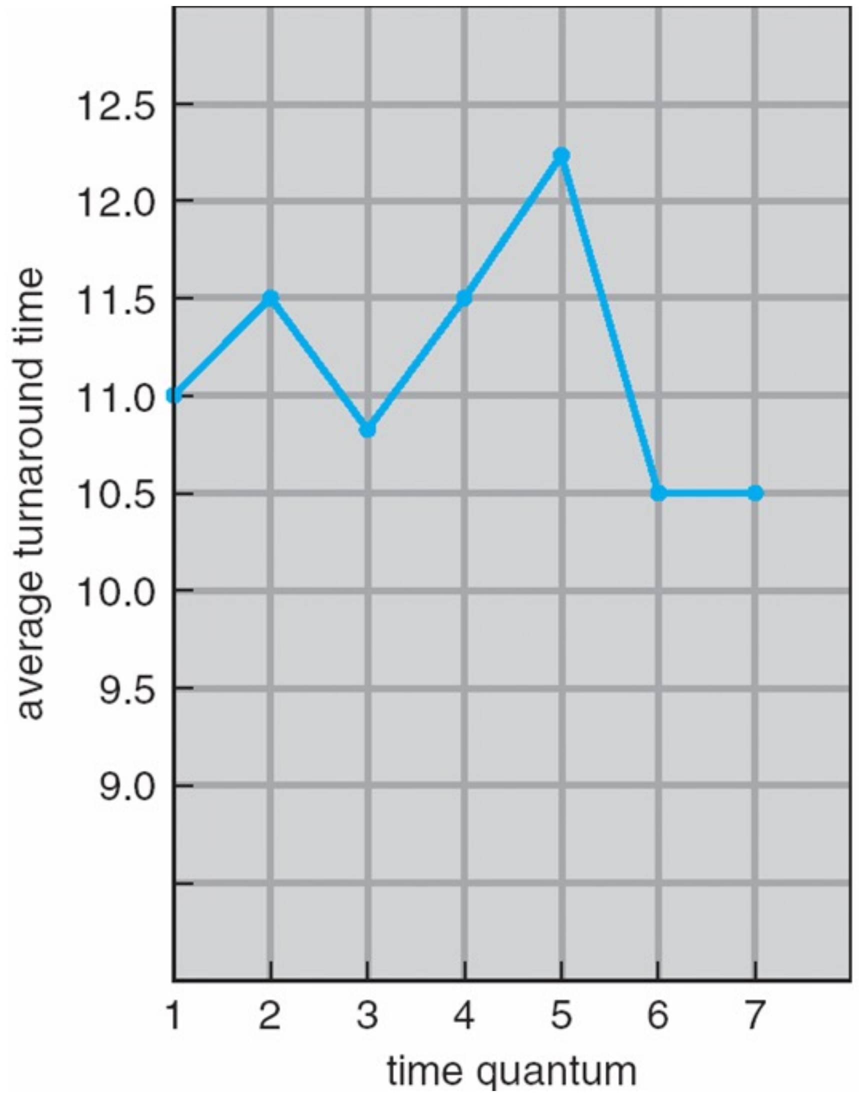


- Higher average turnaround than SJF
- Better ***response***
- q should be large compared to context switch time
- q usually 10ms to 100ms, context switch < 10 usec

Time Quantum and Context Switch Time



Turnaround Time Varies With Time Quantum



process	time
P_1	6
P_2	3
P_3	1
P_4	7

Rule of Thumb: 80% of CPU bursts should be shorter than q

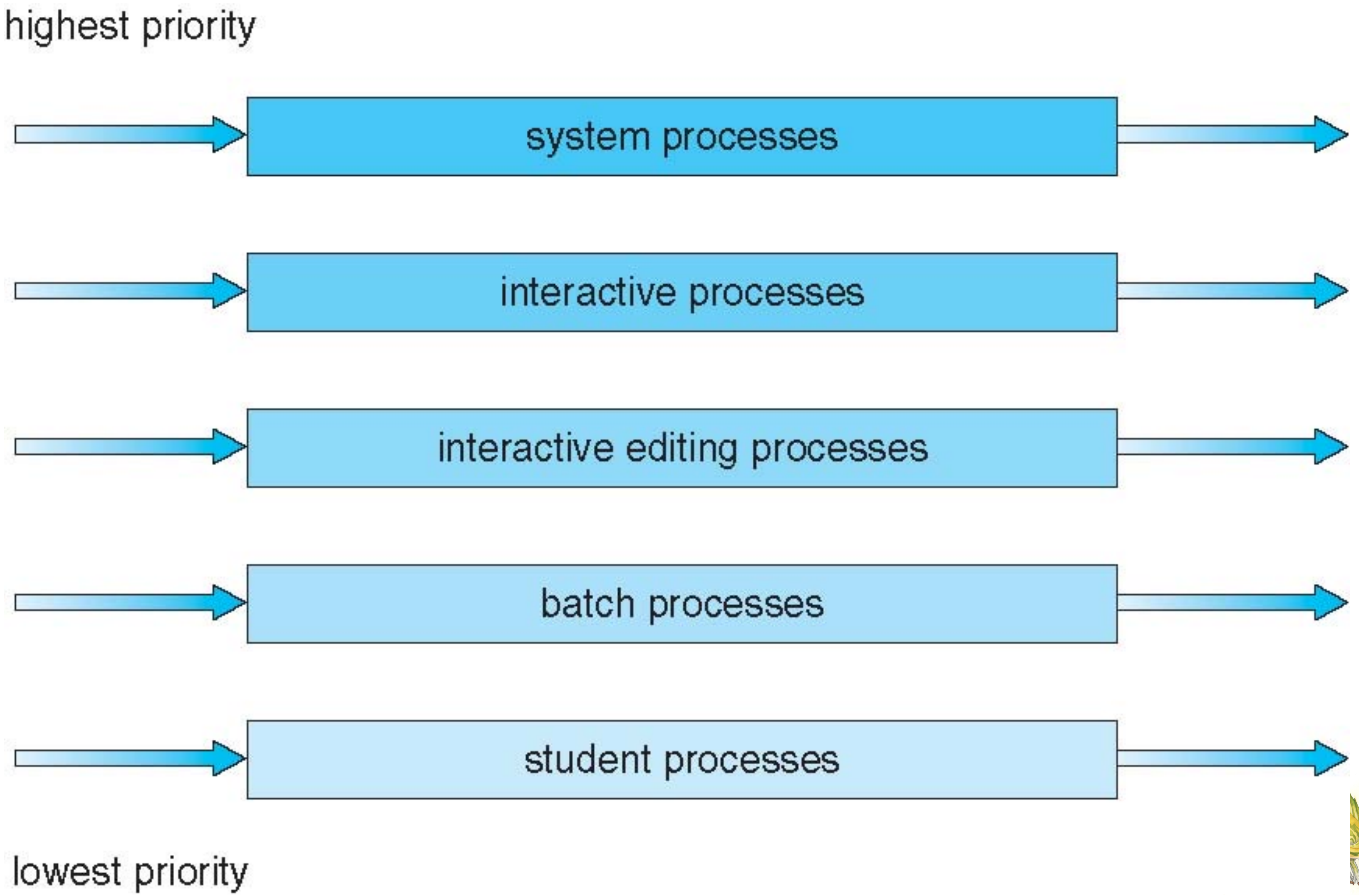
Multilevel Queue

- Processes can be Classified into Different Groups
  - Processes have different response-time requirements ➔ different scheduling needs
  - E.g., ready queue partitioned into two queues
    - **Foreground** (interactive)
    - **Background** (batch)
- Processes Permanently Assigned to one Queue

Multilevel Queue

- Each Queue has its own Scheduling Algorithm
  - Foreground – RR
  - Background – FCFS
- Scheduling must be done between Queues
  - Fixed priority scheduling; (i.e., serve all from foreground then from background).
    - Possibility of starvation
  - Time slice – each queue gets a certain amount of CPU time which it can schedule amongst its processes; i.e., 80% to foreground in RR
  - 20% to background in FCFS

Multilevel Queue Scheduling

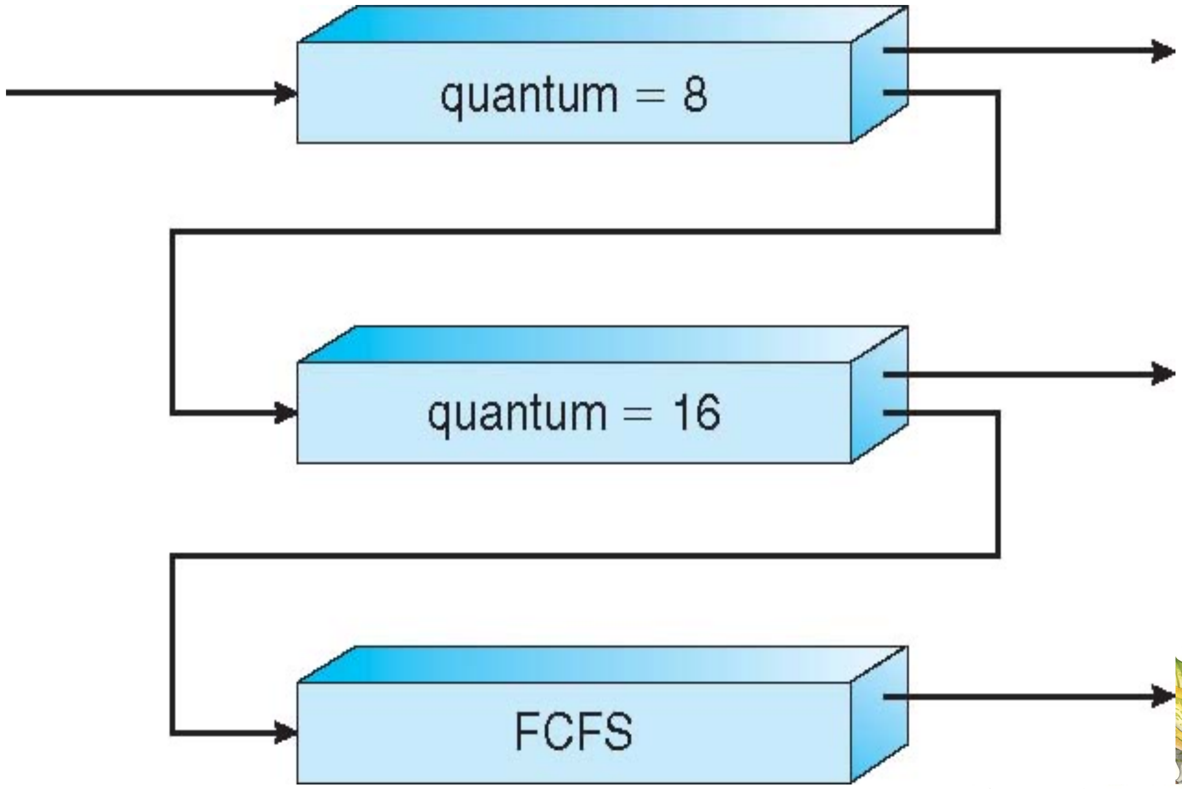


Multilevel Feedback Queue

- A Process can Move between Various Queues
  - Aging can be implemented this way
- Multilevel-Feedback-Queue Scheduler defined by 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**

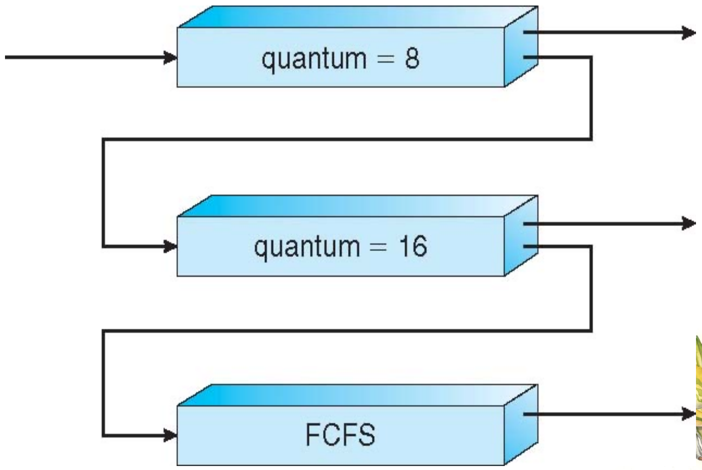
Example: Multilevel Feedback Queue

- Three Queues
  - Q_0 – RR with time quantum 8ms
  - Q_1 – RR time quantum 16 ms
  - Q_2 – FCFS



Example of Multilevel Feedback Queue

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



Thread Scheduling

- Distinction between User- and Kernel-Level Threads
- When Threads Supported, Threads Scheduled, Not Processes
- Many-to-One and Many-to-Many Models
  - Thread library schedules user-level threads to run on LWP (still may not be running on CPU)
  - Known as **process-contention scope (PCS)** since scheduling competition is within process
  - Typically done via priority set by programmer

Thread Scheduling (cont.)

- Kernel Thread Scheduled onto Available CPU
  - **Aka, System-Contention Scope (SCS)**
  - Competition among all threads in system

Pthread Scheduling

- API allows Specifying either **PCS** or **SCS** during Thread Creation
  - PTHREAD_SCOPE_PROCESS schedules threads using PCS scheduling
    - Schedules user-level threads onto available LWPs on systems implementing many-to-many model
  - PTHREAD_SCOPE_SYSTEM schedules threads using SCS scheduling
    - Binds an LWP for each user-level thread on many-to-many systems (similar to one-to-one policy)
- Can be Limited by OS: Linux and Mac OS X only allow PTHREAD_SCOPE_SYSTEM
  - When one-to-one model is used

Pthread Scheduling API

```
#include <pthread.h>
#include <stdio.h>
#define NUM_THREADS 5
int main(int argc, char *argv[]) {
   int i, scope;
   pthread_t tid[NUM THREADS];
   pthread_attr_t attr;
   /* get the default attributes */
   pthread_attr_init(&attr);
   /* first inquire on the current scope */
   if (pthread_attr_getscope(&attr, &scope) != 0)
      fprintf(stderr, "Unable to get scheduling scope\n");
   else {
      if (scope == PTHREAD_SCOPE_PROCESS)
         printf("PTHREAD_SCOPE_PROCESS");
      else if (scope == PTHREAD_SCOPE_SYSTEM)
         printf("PTHREAD_SCOPE_SYSTEM");
      else
         fprintf(stderr, "Illegal scope value.\n");
   }
```

Pthread Scheduling API

```
    /* set the scheduling algorithm to PCS or SCS */
    pthread_attr_setscope(&attr, PTHREAD_SCOPE_SYSTEM);
    /* create the threads */
    for (i = 0; i < NUM_THREADS; i++)
       pthread_create(&tid[i],&attr,runner,NULL);
    /* now join on each thread */
    for (i = 0; i < NUM_THREADS; i++)
       pthread_join(tid[i], NULL);
}
/* Each thread will begin control in this function */
void *runner(void *param)
{
    /* do some work ... */
    pthread_exit(0);
}
```

Multiple-Processor Scheduling

- CPU Scheduling more Complex when Multiple CPUs are Available
- **Homogeneous Processors** within a Multiprocessor
  - As apposed to heterogeneous architecture
- **Asymmetric Multiprocessing**
  - All scheduling decisions, I/O processing, and system activities handled by a single processor
    - Hence, only one processor accesses system data structures, alleviating need for data sharing
    - Simple design & implementation

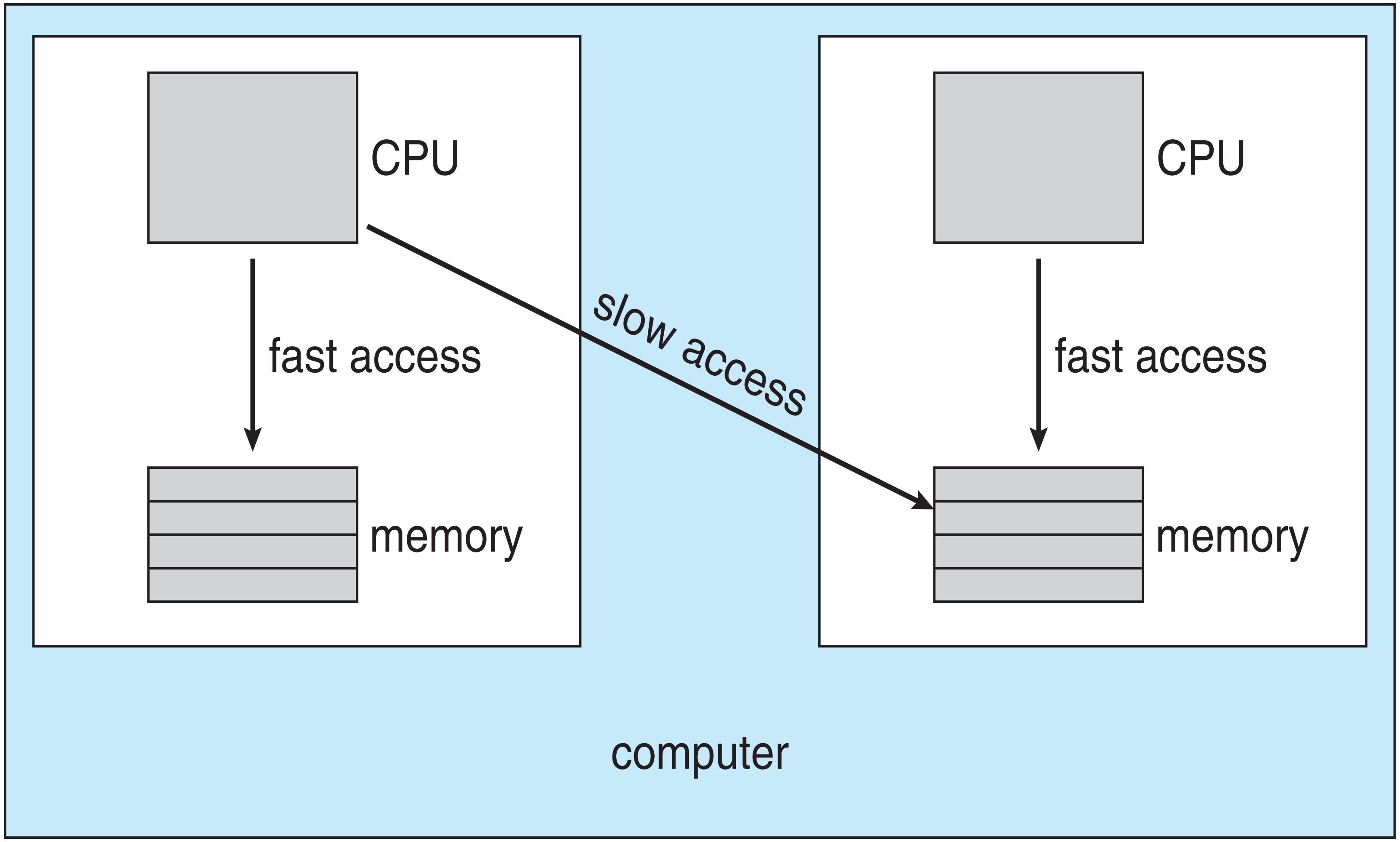
Multiple-Processor Scheduling (cont.)

- **Symmetric Multiprocessing (SMP)**
  - Each processor is self-scheduling
  - All processes in common ready queue OR each has its own private queue of ready processes
  - Currently, most common
  - Supported by all well-known OSs
    - WinXP, Win 2000, Solaris, Linux, Mac OS X

Multiple-Processor Scheduling (cont.)

- **Processor Affinity:** Process has affinity for processor on which it is currently running
- Why to Have Affinity?
  - Process migration very costly; it requires
    - Cache invalidation in source processor
    - Cache repopulation in destination processor
- Types of Affinity
  - **Soft affinity**
  - **Hard affinity** (e.g., supported in Linux)
    - Processor sets (e.g., Solaris)

NUMA and CPU Scheduling



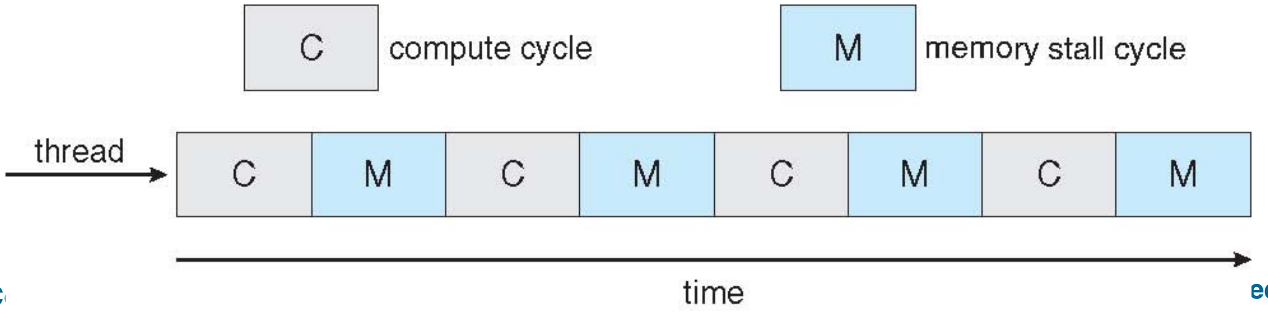
- Main Memory can affect processor affinity
- Memory placement algorithms can consider affinity

Multiple-Processor Scheduling – Load Balancing

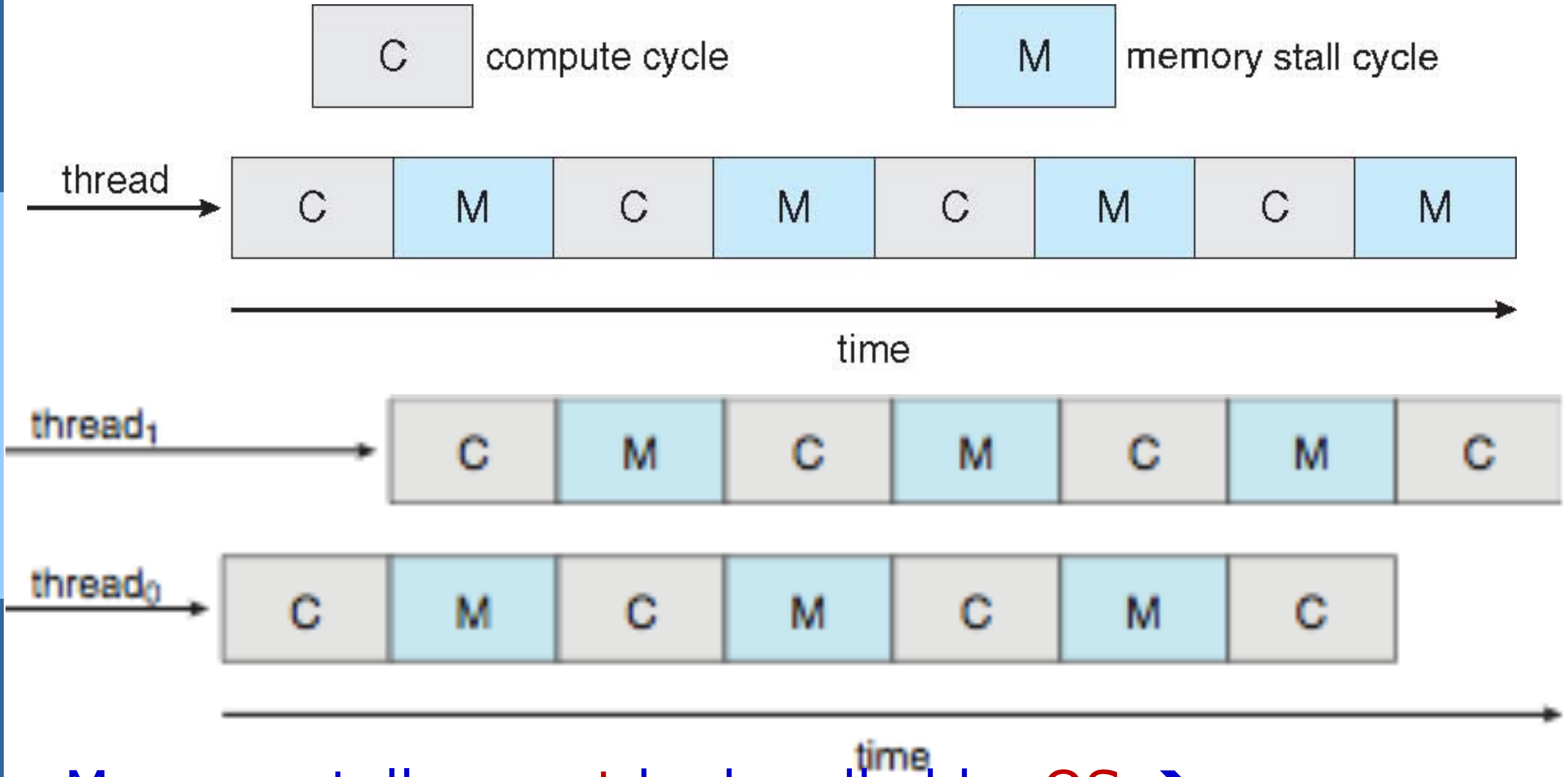
- If SMP, Need to Keep all CPUs Loaded for Efficiency
- **Load Balancing** attempts to keep workload evenly distributed; 2 approaches as follows:
- **Push Migration** – periodic task checks load on each processor, and if found pushes task from overloaded CPU to other CPUs
- **Pull Migration** – idle processors pulls waiting task from busy processor
  - Load balancing may contradict processor affinity

Multicore Processors

- Recent Trend to Place Multiple Cores on Same Physical Chip
  - Faster and consumes less power
- Memory Stall Significantly Affects Performance
- Multiple Threads per Core also Growing
  - E.g., UltraSPARC T1: 8 cores, 4 threads per core ➔ 32 logical processors from OS perspective
  - Takes advantage of memory stall to make progress on another thread while memory retrieve happens



Multithreaded Multicore System



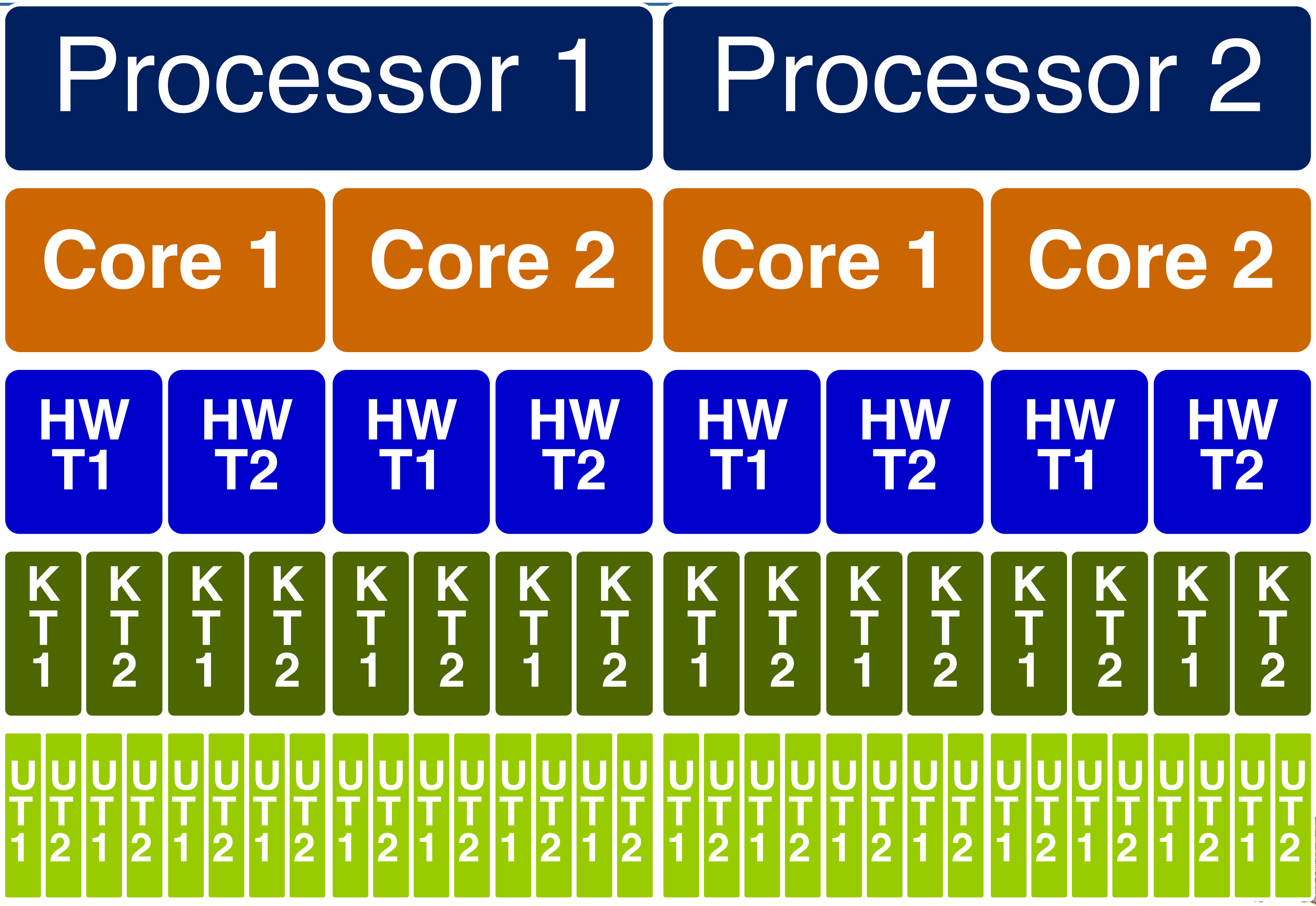
Memory stall cannot be handled by OS ➔

Needs HW threading support to switch to another thread to keep CPU cores as busy as possible

Conclusion: Levels of Scheduling

- **Three Levels** of Scheduling
- **L2**: Scheduling User Threads to Kernel Threads (Scheduling user threads to LWPs)
  - Managed by threading lib or application programmer
  - Not exist in one-to-one models
- **L1**: Scheduling Kernel Threads to HW Threads
  - Aka, logical Processors
  - Typically managed by OS
- **L0**: Scheduling HW Threads for Execution on HW Cores
  - Managed by HW controller

Processing Hierarchies



Real-Time CPU Scheduling

- Real-Time System
  - Results must be produced within a specified deadline period
    - In addition, computing results must be correct
  - Can present obvious challenges
- **Soft Real-Time Systems**
  - Only guarantees that a process will be given a preference over non-critical processes
  - No guarantee as to when critical real-time process will be scheduled
- **Hard Real-Time Systems**
  - Task must be serviced by its deadline

Features of Real-Time Kernels

- Windows XP over 40M lines of Source Code
- A RT OS very Simple Design
  - Written in 10K~100K LoC
  - Not supporting many features of typical OSes
    - Variety of peripheral devices, protection/security mechanisms, and multiple users
- Why RT OS does not Provide Advanced Features?
  - RT OSes are designed for single-purpose apps
  - Additional features require large memory and fast CPU
  - Supporting advanced features increases cost of RT systems

Implementing RT OS

- RT OS Must Support and Implement Three main Features
  - Preemptive, priority-based scheduling
  - Preemptive kernel
  - Minimized latency

Implementing RT OS (cont.)

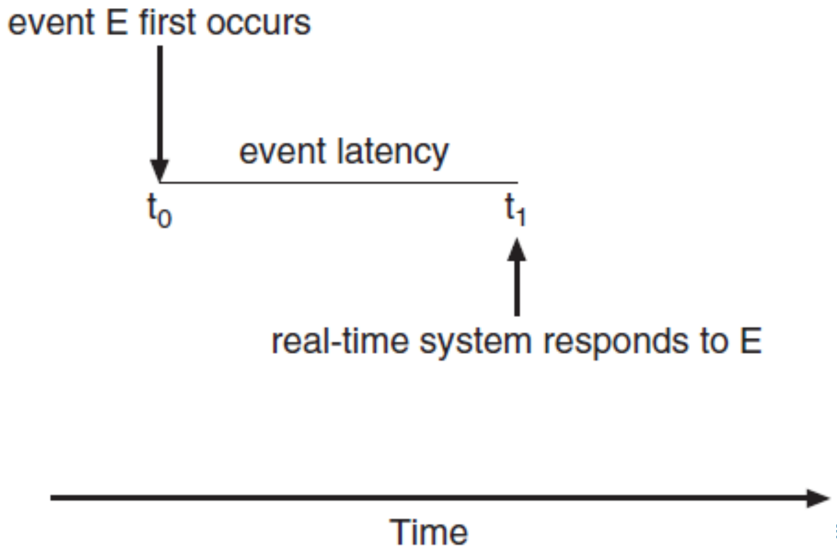
- Preemptive, Priority-based Scheduling
  - Can assign different priorities to processes
  - A process currently running on CPU will be preempted if a higher-priority process arrives
  - Most OSes support priority scheduling
    - Win-XP has 32 different priority levels
  - This feature only guarantees soft RT functionality

Implementing RT OS (cont.)

- Preemptive Kernel
  - Non-preemptive kernels disallow preemption of a process running in kernel mode
    - A kernel-mode process will run until it exists kernel mode
  - Designing preemptive kernels is quite difficult
  - Desktop OSes such as Win-XP are non-preemptive since apps such as word or excel do not require such quick response
  - Preemptive kernels is mandatory in hard RT systems

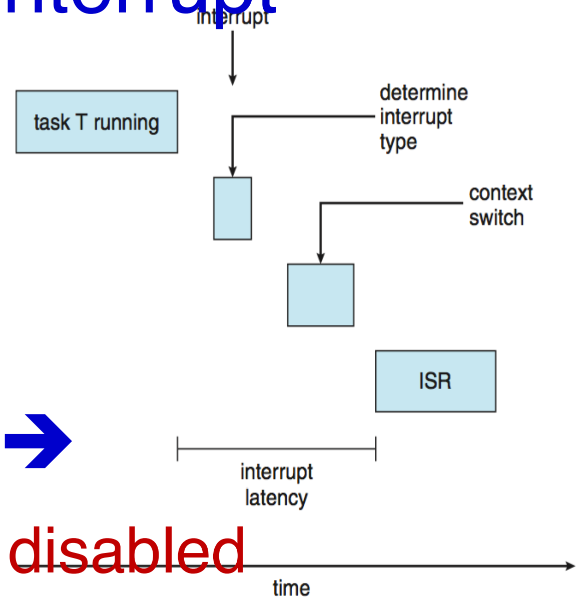
Implementing RT OS (cont.)

- Minimizing Latency
  - Try to minimize response time of events
  - Different events require different latency requirements
    - E.g.1, antilock brake system: 3~5 ms
    - E.g.2, airline radar controller: 1~10s



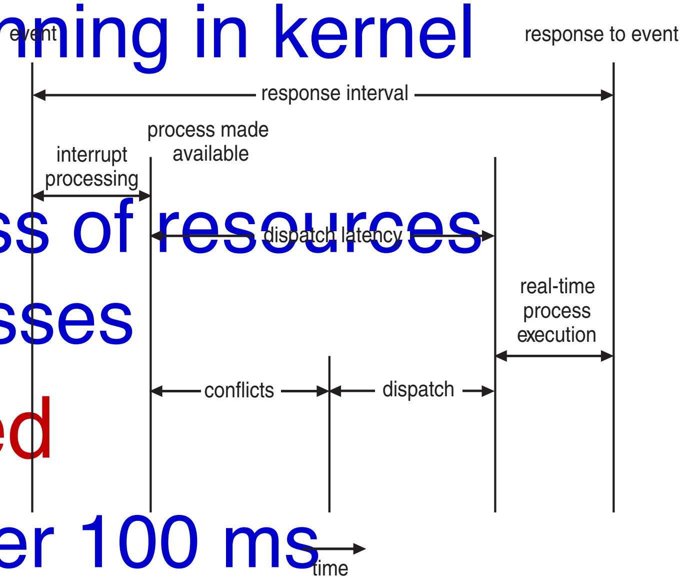
Real-Time CPU Scheduling

- Two Types of Latencies affect Performance
  - Interrupt latency and dispatch latency
- Interrupt latency: Time from Arrival of interrupt to start of routine that services interrupt
  - Completing current instruction
  - Determine type of interrupt
  - Save state of current process
  - Interrupt disabled time interval ➔
    - Amount of time interrupts may be disabled
      - While kernel data structures are being updated



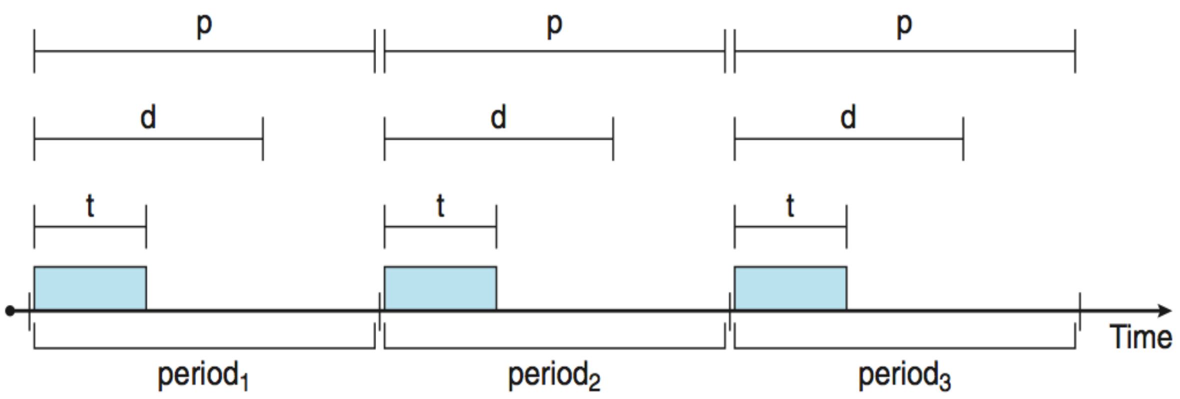
Real-Time CPU Scheduling (cont.)

- Dispatch latency – time for schedule to take current process off CPU and switch to another
  - Effective approach is using preemptive kernel
- Issues in Dispatch Latency
  1. Preemption of any process running in kernel mode
  2. Release by low-priority process of resources needed by high-priority processes
- Effect of Preemption Disabled
  - In solaris, dispatch latency over 100 ms
  - When enabled, less than 1 ms



Priority-based Scheduling

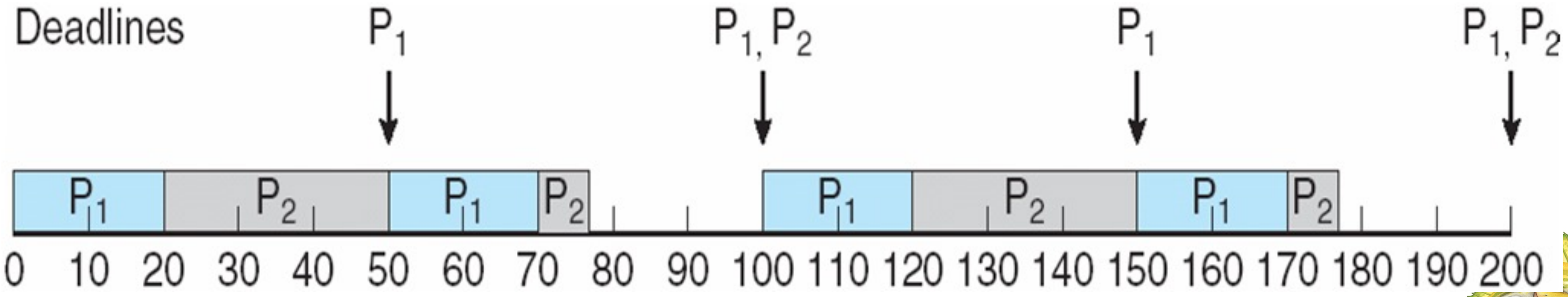
- For Real-Time Scheduling, Scheduler must support Preemptive, Priority-based Scheduling
  - But only guarantees soft real-time
- For Hard Real-Time must also Provide ability to **Meet Deadlines**
- Process **ic**
  ones re
  - Has p
  - 0 ≤ *t* ≤
  - **Rate**



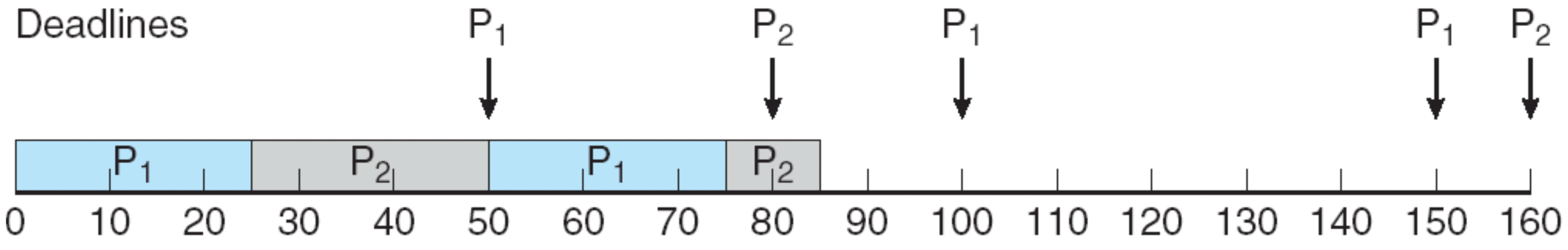
- Admission Control: either admits or rejects request

Rate Monotonic Scheduling

- A Priority is Assigned based on Inverse of its Period (Static Priority policy)
- Shorter periods = higher Priority
- Longer periods = lower Priority
- P_1 is assigned a Higher Priority than P_2
- Example: p1=50, p2=100, t1=20, t2=35



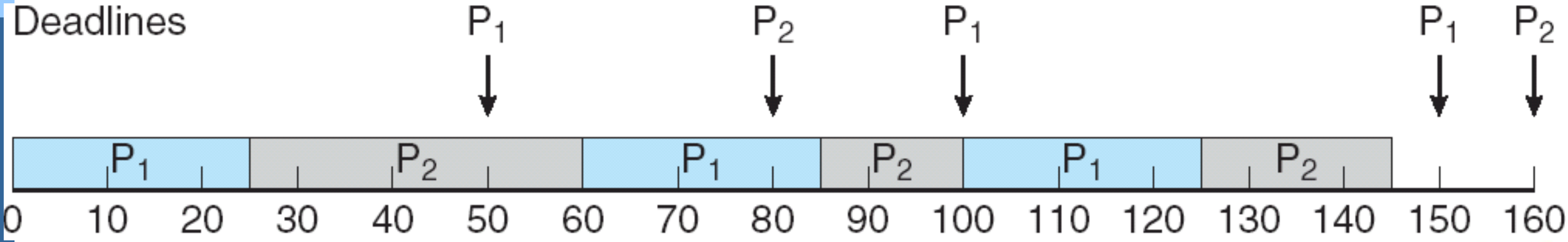
Missed Deadlines with Rate Monotonic Scheduling



Another Example: p1=50, p2=80, t1=25, t2=35

Earliest Deadline First Scheduling (EDF)

- Priorities are Assigned according to Deadlines (Dynamic Scheduling Policy)
  - Earlier deadline, higher priority
  - Later deadline, lower priority
- Example: p1=50, p2=80, t1=25, t2=35



POSIX Real-Time Scheduling

- POSIX.1b standard
- API provides functions for managing real-time threads
- Defines two scheduling classes for real-time threads:
  1. SCHED_FIFO - threads are scheduled using a FCFS strategy with a FIFO queue. There is no time-slicing for threads of equal priority
  2. SCHED_RR - similar to SCHED_FIFO except time-slicing occurs for threads of equal priority

POSIX Real-Time Scheduling (cont.)

- Defines Two Functions for Getting and Setting Scheduling policy

1. `pthread_attr_getsched_policy(pthread_attr_t *attr, int *policy)`
2. `pthread_attr_setsched_policy(pthread_attr_t *attr, int policy)`

POSIX Real-Time Scheduling API

```
#include <pthread.h>
#include <stdio.h>
#define NUM_THREADS 5
int main(int argc, char *argv[])
{
   int i, policy;
   pthread_t tid[NUM_THREADS];
   pthread_attr_t attr;
   /* get the default attributes */
   pthread_attr_init(&attr);
   /* get the current scheduling policy */
   if (pthread_attr_getschedpolicy(&attr, &policy) != 0)
      fprintf(stderr, "Unable to get policy.\n");
   else {
      if (policy == SCHED_OTHER) printf("SCHED_OTHER\n");
      else if (policy == SCHED_RR) printf("SCHED_RR\n");
      else if (policy == SCHED_FIFO)
printf("SCHED_FIFO\n");
   }
```

POSIX Real-Time Scheduling API (Cont.)

```
   /* set the scheduling policy - FIFO, RR, or OTHER */
   if (pthread_attr_setschedpolicy(&attr, SCHED_FIFO) != 0)
      fprintf(stderr, "Unable to set policy.\n");
   /* create the threads */
   for (i = 0; i < NUM_THREADS; i++)
      pthread_create(&tid[i],&attr,runner,NULL);
   /* now join on each thread */
   for (i = 0; i < NUM_THREADS; i++)
      pthread_join(tid[i], NULL);
}

/* Each thread will begin control in this function */
void *runner(void *param)
{
   /* do some work ... */
   pthread_exit(0);
}
```

Operating System Examples

- Reading Assignments
  - Linux scheduling
  - Windows scheduling
  - Solaris scheduling

Linux Scheduling Through Version 2.5

- Prior to kernel version 2.5, ran Variation of Standard UNIX Scheduling Algorithm
- Version 2.5 Moved to Constant Order *O*(1) Scheduling Time
  - Preemptive, priority based
  - Two priority ranges: time-sharing and real-time
  - **Real-time** range from 0 to 99 and **nice** value from 100 to 140
  - Map into global priority with numerically lower values indicating higher priority
  - Higher priority gets larger q

Linux Scheduling Through Version 2.5

- Version 2.5 Moved to Constant Order *O*(1) Scheduling Time
  - Task run-able as long as time left in time slice (**active**)
  - If no time left (**expired**), not run-able until all other tasks use their slices
  - All run-able tasks tracked in per-CPU **runqueue** data structure
    - Two priority arrays (active, expired)
    - Tasks indexed by priority
    - When no more active, arrays are exchanged
  - Worked well, but poor response times for interactive processes

Linux Scheduling in Version 2.6.23 +

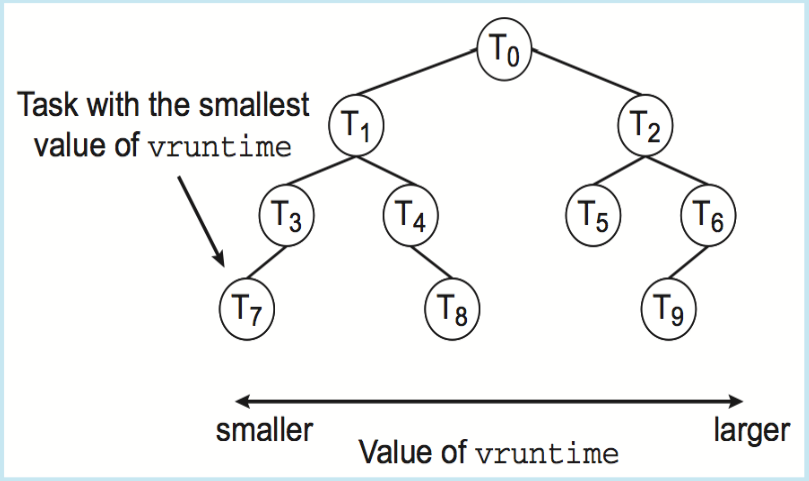
- ***Completely Fair Scheduler*** (CFS)
- **Scheduling Classes**
  - Each has specific priority
  - Scheduler picks highest priority task in highest scheduling class
  - Rather than quantum based on fixed time allotments, based on proportion of CPU time
  - 2 scheduling classes included, others can be added
    1. default
    2. real-time

Linux Scheduling in Version 2.6.23 + (cont.)

- Quantum calculated based on **nice value** from -20 to +19
  - Lower value is higher priority
  - Calculates **target latency** – interval of time during which task should run at least once
  - Target latency can increase if say number of active tasks increases
- CFS scheduler maintains per task **virtual run time** in variable **`vruntime`**
  - Associated with decay factor based on priority of task – lower priority is higher decay rate
  - Normal default priority yields virtual run time = actual run time

CFS Performance

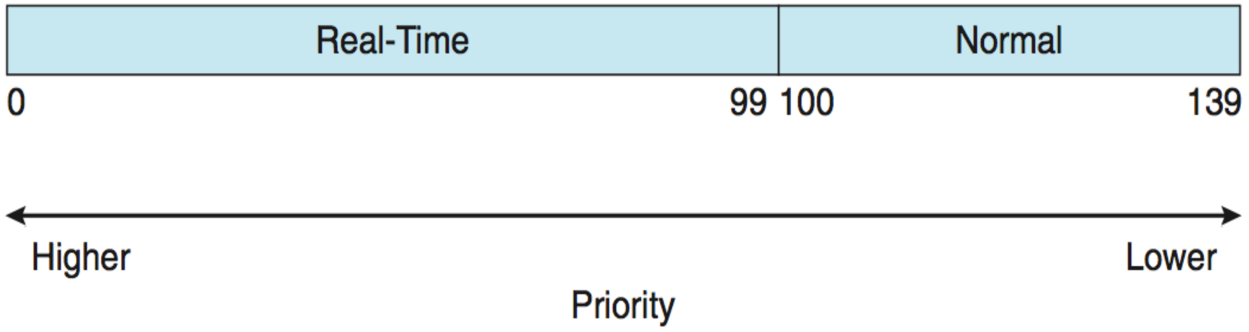
The Linux CFS scheduler provides an efficient algorithm for selecting which task to run next. Each runnable task is placed in a red-black tree—a balanced binary search tree whose key is based on the value of `vruntime`. This tree is shown below:



When a task becomes runnable, it is added to the tree. If a task on the tree is not runnable (for example, if it is blocked while waiting for I/O), it is removed. Generally speaking, tasks that have been given less processing time (smaller values of `vruntime`) are toward the left side of the tree, and tasks that have been given more processing time are on the right side. According to the properties of a binary search tree, the leftmost node has the smallest key value, which for the sake of the CFS scheduler means that it is the task with the highest priority. Because the red-black tree is balanced, navigating it to discover the leftmost node will require $O(lg N)$ operations (where N is the number of nodes in the tree). However, for efficiency reasons, the Linux scheduler caches this value in the variable `rb_leftmost`, and thus determining which task to run next requires only retrieving the cached value.

Linux Scheduling (Cont.)

- Real-Time Scheduling according to POSIX.1b
  - Real-time tasks have static priorities
- Real-Time plus Normal Map into Global Priority Scheme
- Nice value of -20 maps to Global priority 100
- Nice value of +19 maps to priority 139



Windows Scheduling

- Windows uses priority-based preemptive scheduling
- Highest-priority thread runs next
- **Dispatcher** is scheduler
- Thread runs until (1) blocks, (2) uses time slice, (3) preempted by higher-priority thread
- Real-time threads can preempt non-real-time
- 32-level priority scheme
- **Variable class** is 1-15, **real-time class** is 16-31
- Priority 0 is memory-management thread
- Queue for each priority
- If no run-able thread, runs **idle thread**

Windows Priority Classes

- Win32 API identifies several priority classes to which a process can belong
  - REALTIME_PRIORITY_CLASS, HIGH_PRIORITY_CLASS, ABOVE_NORMAL_PRIORITY_CLASS,NORMAL_PRIORITY_CLASS, BELOW_NORMAL_PRIORITY_CLASS, IDLE_PRIORITY_CLASS
  - All are variable except REALTIME
- A thread within a given priority class has a relative priority
  - TIME_CRITICAL, HIGHEST, ABOVE_NORMAL, NORMAL, BELOW_NORMAL, LOWEST, IDLE
- Priority class and relative priority combine to give numeric priority
- Base priority is NORMAL within the class
- If quantum expires, priority lowered, but never below base

Windows Priority Classes (cont.)

- If wait occurs, priority boosted depending on what was waited for
- Foreground window given 3x priority boost
- Windows 7 added **user-mode scheduling** (**UMS**)
  - Applications create and manage threads independent of kernel
  - For large number of threads, much more efficient
  - UMS schedulers come from programming language libraries like C++ **Concurrent Runtime** (ConcRT) framework

Windows Priorities

	real-time	high	above normal	normal	below normal	idle priority
time-critical	31	15	15	15	15	15
highest	26	15	12	10	8	6
above normal	25	14	11	9	7	5
normal	24	13	10	8	6	4
below normal	23	12	9	7	5	3
lowest	22	11	8	6	4	2
idle	16	1	1	1	1	1

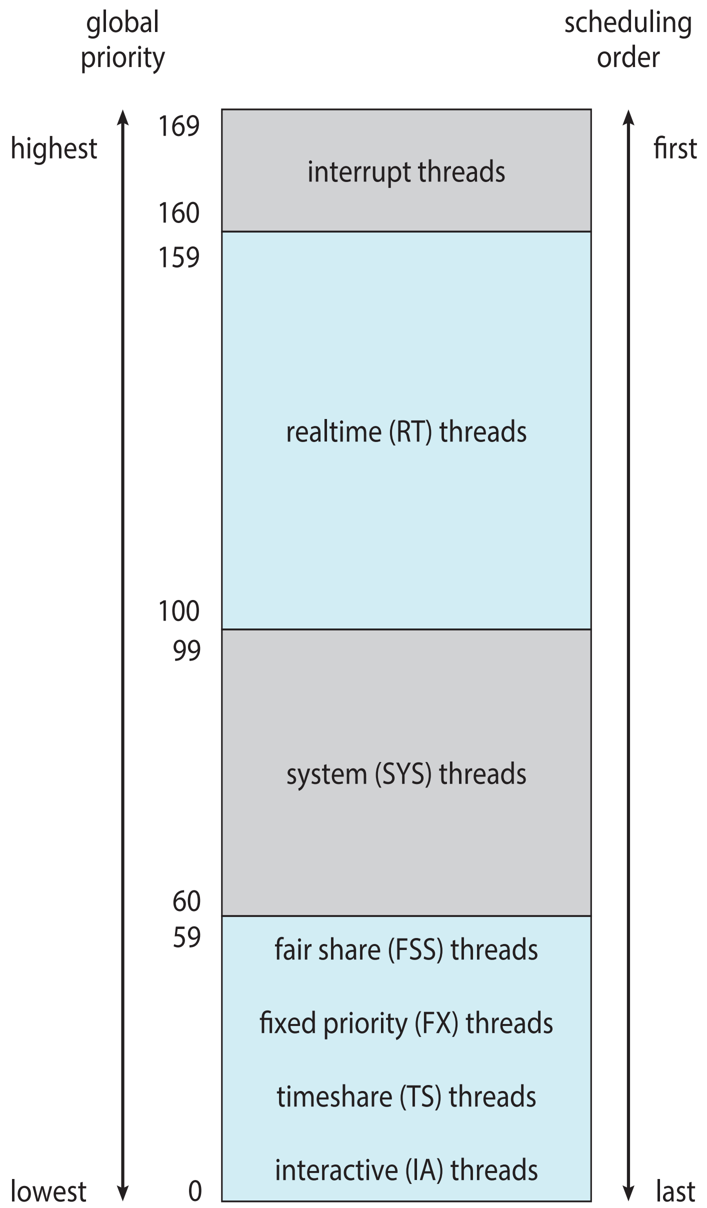
Solaris

- Priority-based Scheduling
- Six classes available
  - Time sharing (default) (TS)
  - Interactive (IA)
  - Real time (RT)
  - System (SYS)
  - Fair Share (FSS)
  - Fixed priority (FP)
- Given thread can be in one class at a time
- Each class has its own scheduling algorithm
- Time sharing is multi-level feedback queue
  - Loadable table configurable by sysadmin

Solaris Dispatch Table

priority	time quantum	time quantum expired	return from sleep
0	200	0	50
5	200	0	50
10	160	0	51
15	160	5	51
20	120	10	52
25	120	15	52
30	80	20	53
35	80	25	54
40	40	30	55
45	40	35	56
50	40	40	58
55	40	45	58
59	20	49	59

Solaris Scheduling

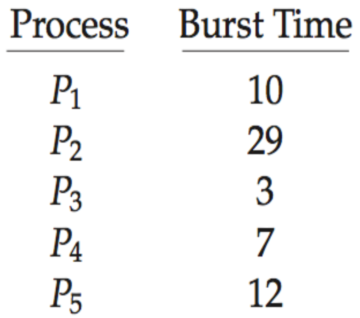


Solaris Scheduling (Cont.)

- Scheduler converts class-specific priorities into a per-thread global priority
  - Thread with highest priority runs next
  - Runs until (1) blocks, (2) uses time slice, (3) preempted by higher-priority thread
  - Multiple threads at same priority selected via RR

Algorithm Evaluation

- How to select CPU-scheduling algorithm for an OS?
- Determine criteria, then evaluate algorithms
- **Deterministic modeling**
  - Type of **analytic evaluation**
  - Takes a redetermined workload and defines ance of each algorithm for that wor

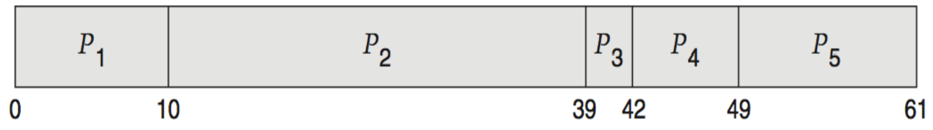


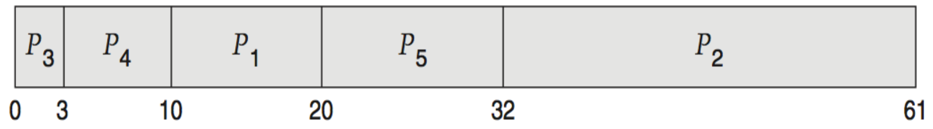
Process	Burst Time
P_1	10
P_2	29
P_3	3
P_4	7
P_5	12

- Consider 5 processes arriving at time 0:

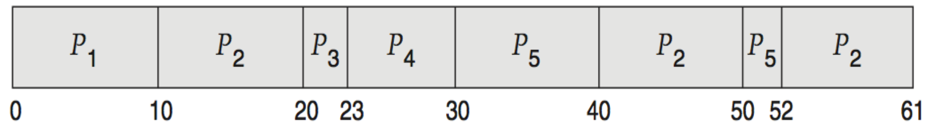
Deterministic Evaluation

- For each algorithm, calculate minimum average waiting time
- Simple and fast, but requires exact numbers for input, applies only to those inputs
  - FCS is 28ms:





  - Non-preemptive SFJ is 13ms:



Queueing Models

- Describes the arrival of processes, and CPU and I/O bursts probabilistically
  - Commonly exponential, and described by mean
  - Computes average throughput, utilization, waiting time, etc
- Computer system described as network of servers, each with queue of waiting processes
  - Knowing arrival rates and service rates
  - Computes utilization, average queue length, average wait time, etc

Little's Formula

- n = average queue length
- W = average waiting time in queue
- λ = average arrival rate into queue
- Little's law – in steady state, processes leaving queue must equal processes arriving, thus:

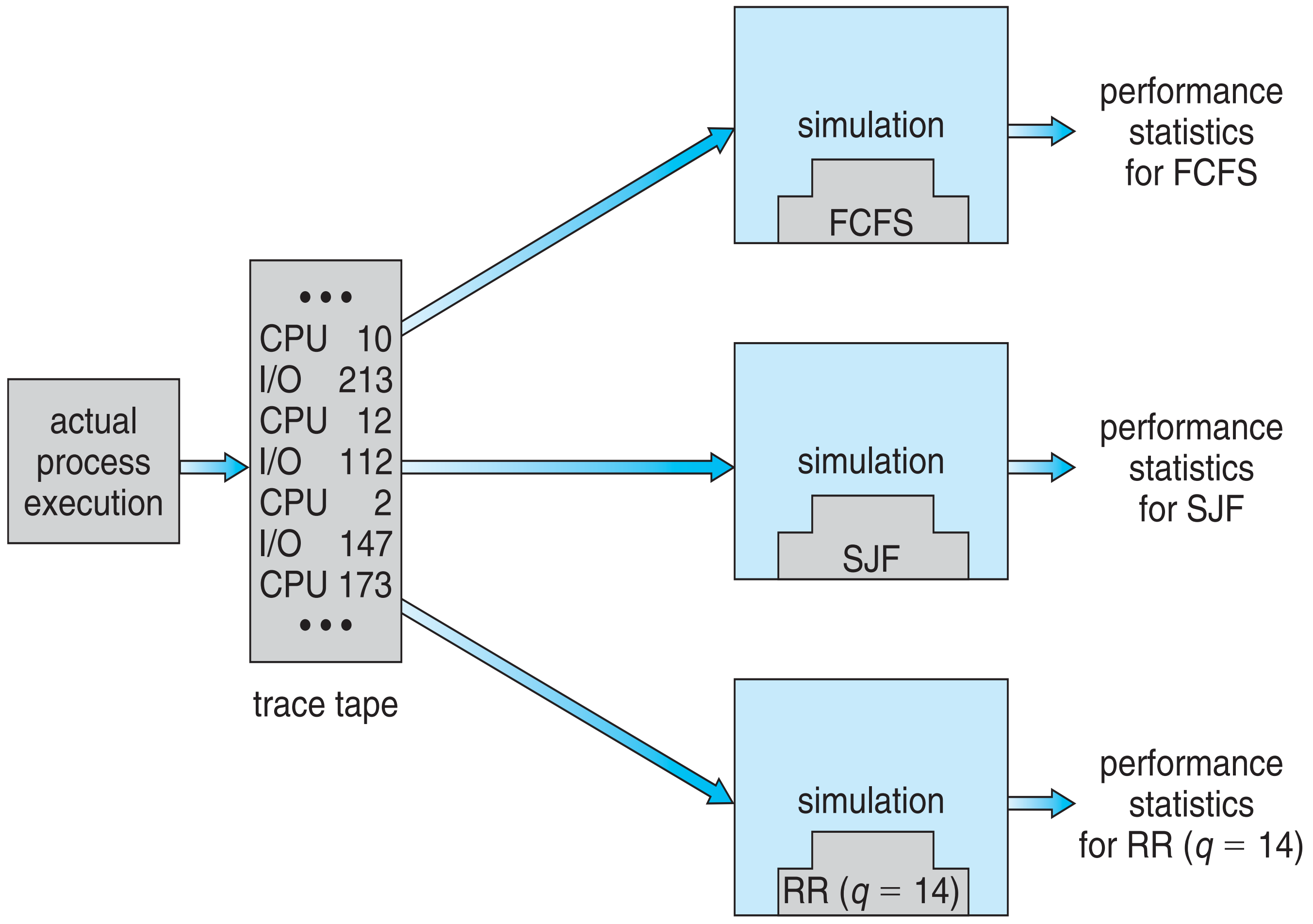
  $$n = \lambda \times W$$

  - Valid for any scheduling algorithm and arrival distribution
- For example, if on average 7 processes arrive per second, and normally 14 processes in queue, then average wait time per process = 2 seconds

Simulations

- Queueing models limited
- **Simulations** more accurate
  - Programmed model of computer system
  - Clock is a variable
  - Gather statistics indicating algorithm performance
  - Data to drive simulation gathered via
    - Random number generator according to probabilities
    - Distributions defined mathematically or empirically
    - Trace tapes record sequences of real events in real systems

Evaluation of CPU Schedulers by Simulation



Implementation

- Even simulations have limited accuracy
- Just implement new scheduler and test in real systems
  - High cost, high risk
  - Environments vary
- Most flexible schedulers can be modified per-site or per-system
- Or APIs to modify priorities
- But again environments vary

End of Lecture 5