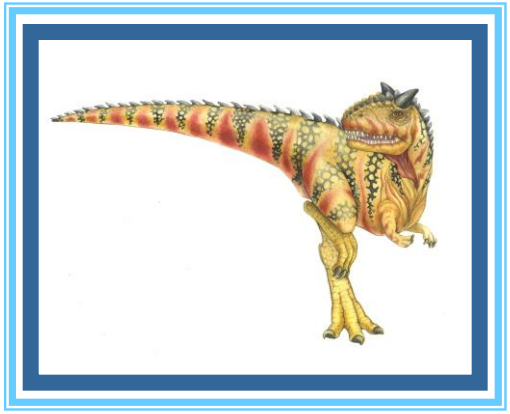
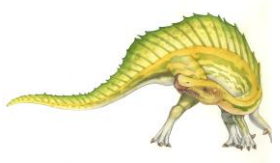
Chapter 6: CPU Scheduling



Chapter 6: CPU Scheduling

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

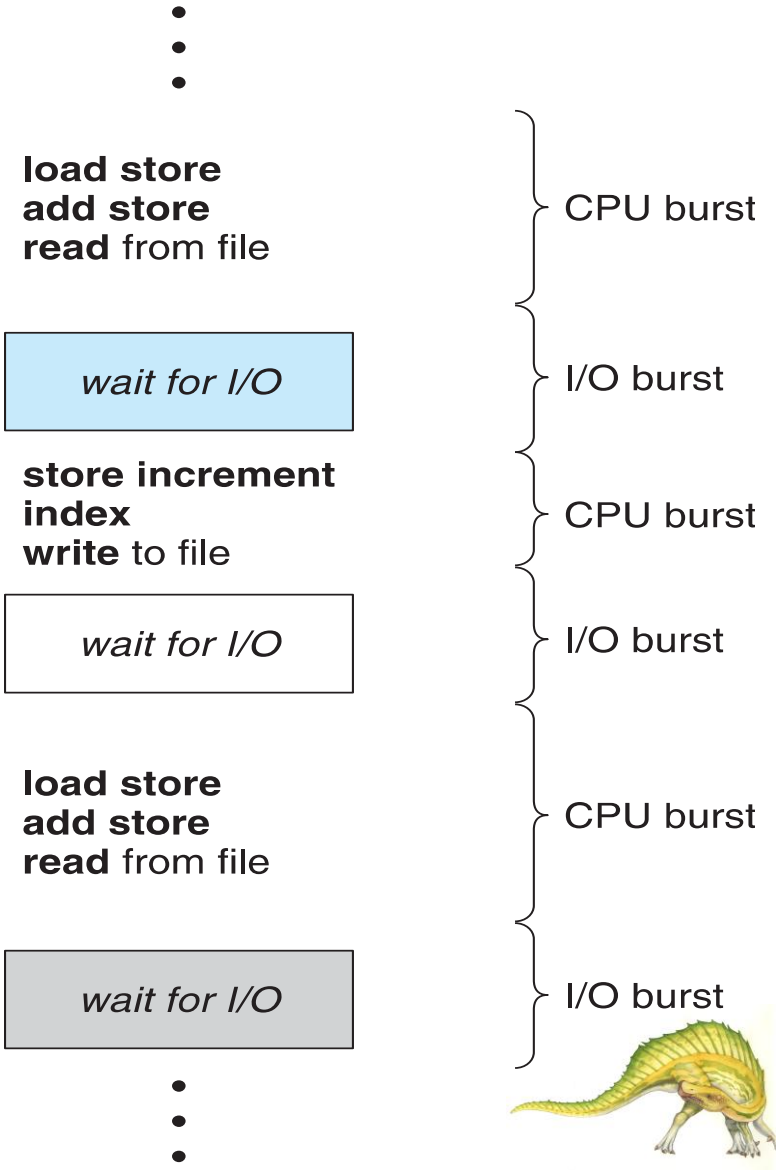


Objectives

- ***CPU scheduling*** is the basis for multi-programmed operating systems
  - ***Process Scheduling***
    - By switching among processes (see Chap-3)
      - Increases productivity of computer
  - ***Thread Scheduling***
    - By switching among kernel threads (see Chap-4)
- To describe various CPU-scheduling algorithms
- To discuss evaluation criteria for selecting a CPU-scheduling algorithm for a particular system
- To examine the scheduling algorithms of several operating systems

Basic Concepts

- **Objective of multiprogramming**: Achieve maximum CPU utilization as possible
  - CPU always running a process
  - No idle CPU
- CPU–I/O Burst Cycle:
  - Process execution consists of a **cycle** of CPU execution and I/O wait
  - When a process is waiting, then assign the CPU to another process
  - See Process Scheduler on Chap-3
- **CPU burst** followed by **I/O burst**
- CPU burst distribution is of main concern



Histogram of CPU-burst Times

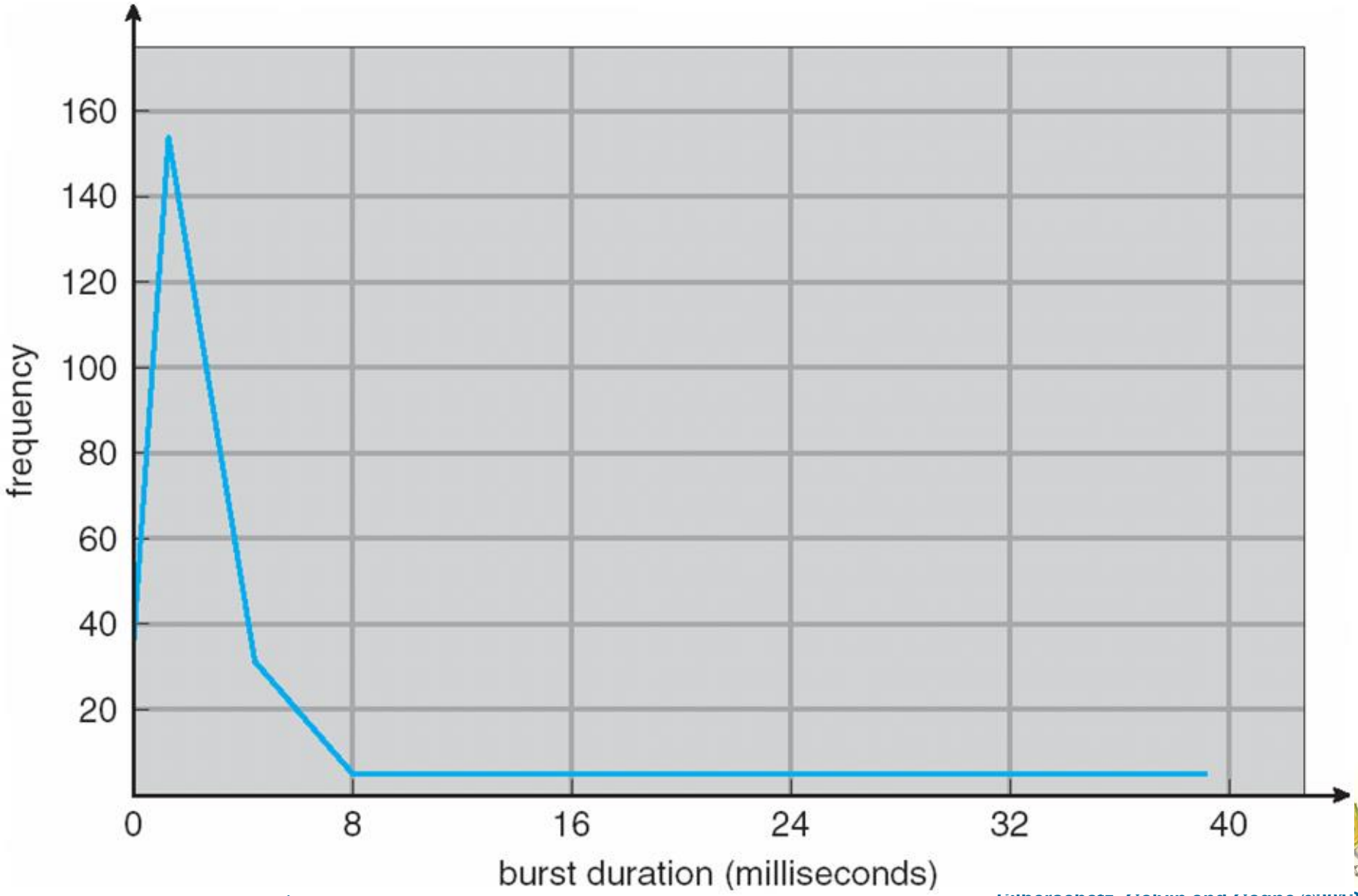
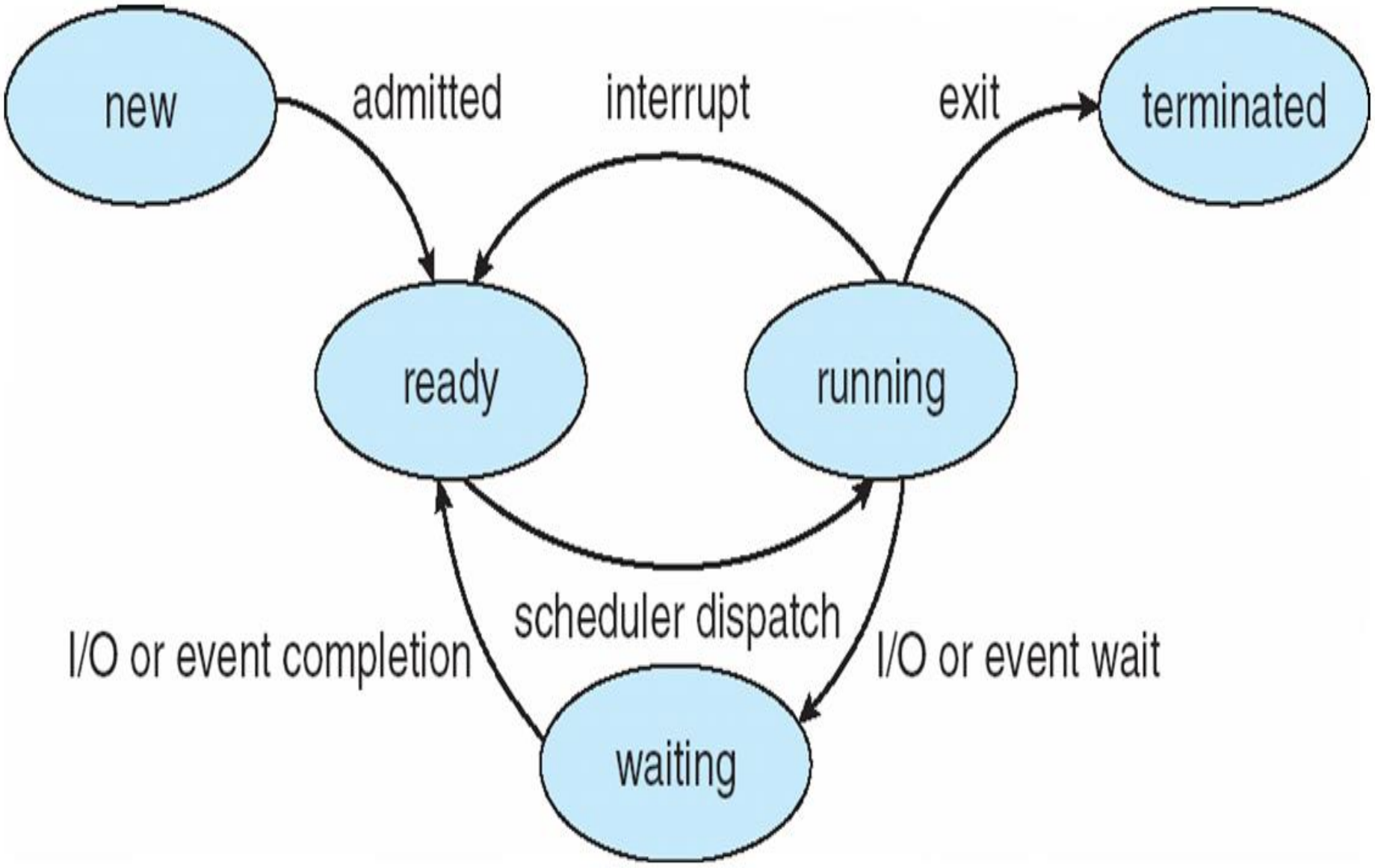
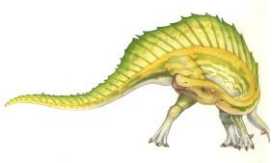


Diagram of Process State



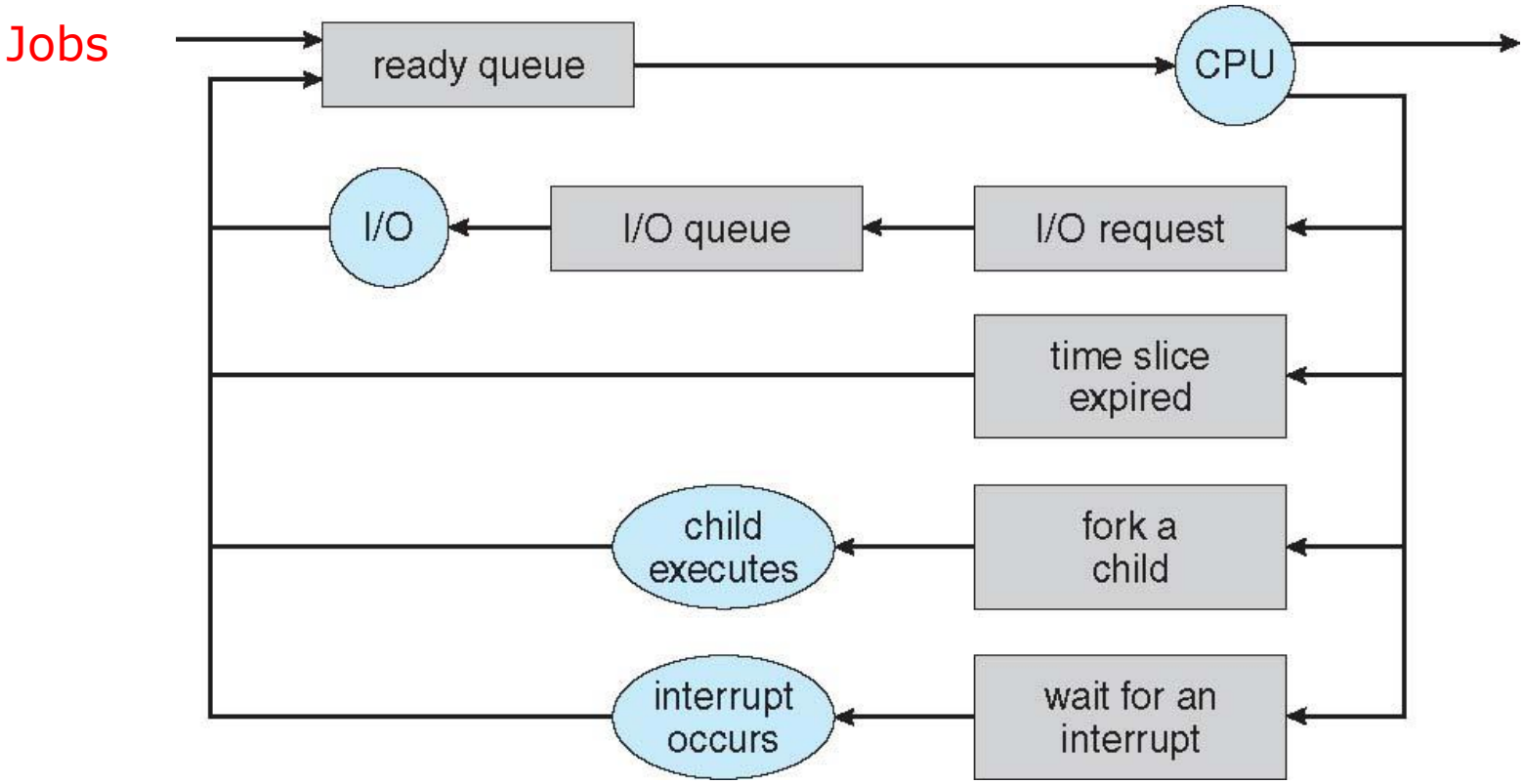
CPU Scheduler

- **Short-term scheduler** selects from among the processes in ready queue and allocates the CPU to one of them
  - Queue may be ordered in various ways: FIFO, LIFO, Random, Priority, … etc
- CPU scheduling decisions may take place when a process:
  1. Switches from running to waiting state; ex: as result of I/O request or `wait()`
  2. Switches from running to ready state; ex: when an interrupt occurs
  3. Switches from waiting to ready; ex: at completion of I/O
  4. Terminates
- Scheduling taking place only under circumstances 1 and 4 is **nonpreemptive**
  - No choice in terms of scheduling; new process must be selected for CPU
    - Process keeps the CPU until it either terminates or switches to waiting state
- All other scheduling is **preemptive**; can result in race conditions (see Chap-5)
  - Consider access to shared data
  - Consider preemption while in kernel mode
  - Consider interrupts occurring during crucial OS activities



Representation of Process Scheduling

- **Queueing diagram** represents queues, resources, flows



Dispatcher

- Dispatcher module gives control of the CPU to the process selected by the short-term scheduler; this involves:
  - Switching context
  - Switching to user mode
  - Jumping to the proper location in the user program to restart that program
- **Dispatch latency** – time it takes for the dispatcher to stop one process and start another running
  - Dispatcher is invoked during every process switch; hence it should be as fast as possible

Scheduling Criteria

- **CPU utilization** – keep the CPU as busy as possible
  - Ranges from 40% to 90% in a real system; i.e., from light to heavy loaded
  - **Scheduling algorithm optimization criteria: *Max CPU utilization***
- **Throughput** – # of processes that complete their execution per time unit
  - Ranges from 10 processes/second to 1 process/hour
  - **Scheduling algorithm optimization criteria: *Max throughput***
- **Turnaround time** – amount of time to execute a particular process
  - Sum of times spent in job pool + ready queue + CPU execution + doing I/O
  - **Scheduling algorithm optimization criteria: *Min turnaround time***
- **Waiting time** – amount of time a process has been waiting in the ready queue
  - Sum of times spent waiting in the ready queue
  - **Scheduling algorithm optimization criteria: *Min waiting time***
- **Response time** – amount of time it takes from when a request was submitted until the first response is produced, not output (for time-sharing environment)
  - The time it takes to start responding to the user; in an interactive system
  - **Scheduling algorithm optimization criteria: *Min response time***

First-Come First-Served (FCFS) Scheduling Algorithm

Process	Burst Time
P_1	24
P_2	3
P_3	3

- Suppose that the processes arrive in the order: P_1 , P_2 , P_3
  - The **Gantt Chart** for the schedule is: [includes start and finish time of each process]

- Waiting time for P_1 = 0; P_2 = 24; P_3 = 27
- Average waiting time: (0 + 24 + 27)/3 = 17
- The simplest scheduling algorithm but usually very bad average waiting time

FCFS Scheduling

Suppose that the processes arrive in the order:

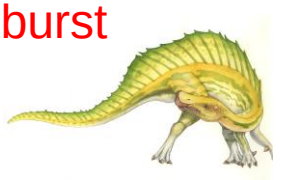
$$P_2 , P_3 , P_1$$

- 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; substantial reduction in waiting time
- Much better than previous case
- **Convoy effect** - short process behind long process;
  - CPU-bound holds CPU while I/O-bounds wait in ready queue for the CPU
    - I/O devices are idle until CPU released… then CPU is idle.. Then … this repeats
  - Consider one CPU-bound and many I/O-bound processes
    - I/O-bounds spend most of the time waiting for CPU-bound to release CPU
    - Result in lower CPU ***and*** device utilization
- ***FCFS is nonpreemptive***: process holds CPU until termination or I/O request

Shortest-Job-First (SJF) Scheduling Algorithm

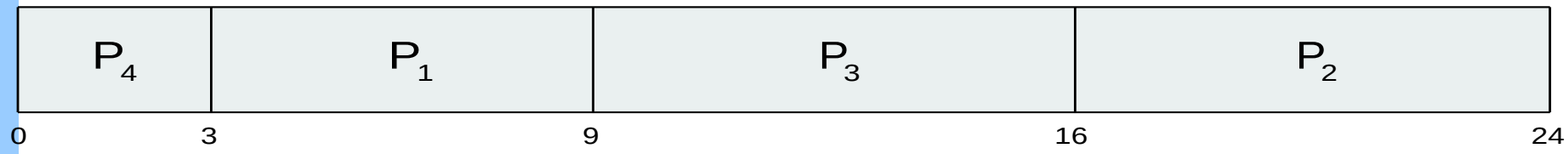
- Associate with each process the length of its next CPU burst
  - Use these lengths to schedule the process with the shortest time
  - Always assign the CPU to the process that has the ***smallest next CPU burst***
  - FCFS breaks the tie if two process have the same next CPU burst length
- Better term: ***Shortest-Next-CPU-Burst-Scheduling*** (SNCB) algorithm
  - But most books use SJF
- SJF is optimal – gives minimum average waiting time for a given set of processes
  - The difficulty is knowing the length of the next CPU request
  - Could ask users to estimate their processes' time limits; for ***job scheduling***
    - CPU scheduling can use these time limits as estimates of CPU burst lengths



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
  - With FCFS it would be 10.25 time units with this order P_1 P_2 P_3 P_4
- Moving a short process before a long one decreases its waiting time more than it increases the long process's waiting time. Thus, decrease of average waiting time

Determining Length of Next CPU Burst

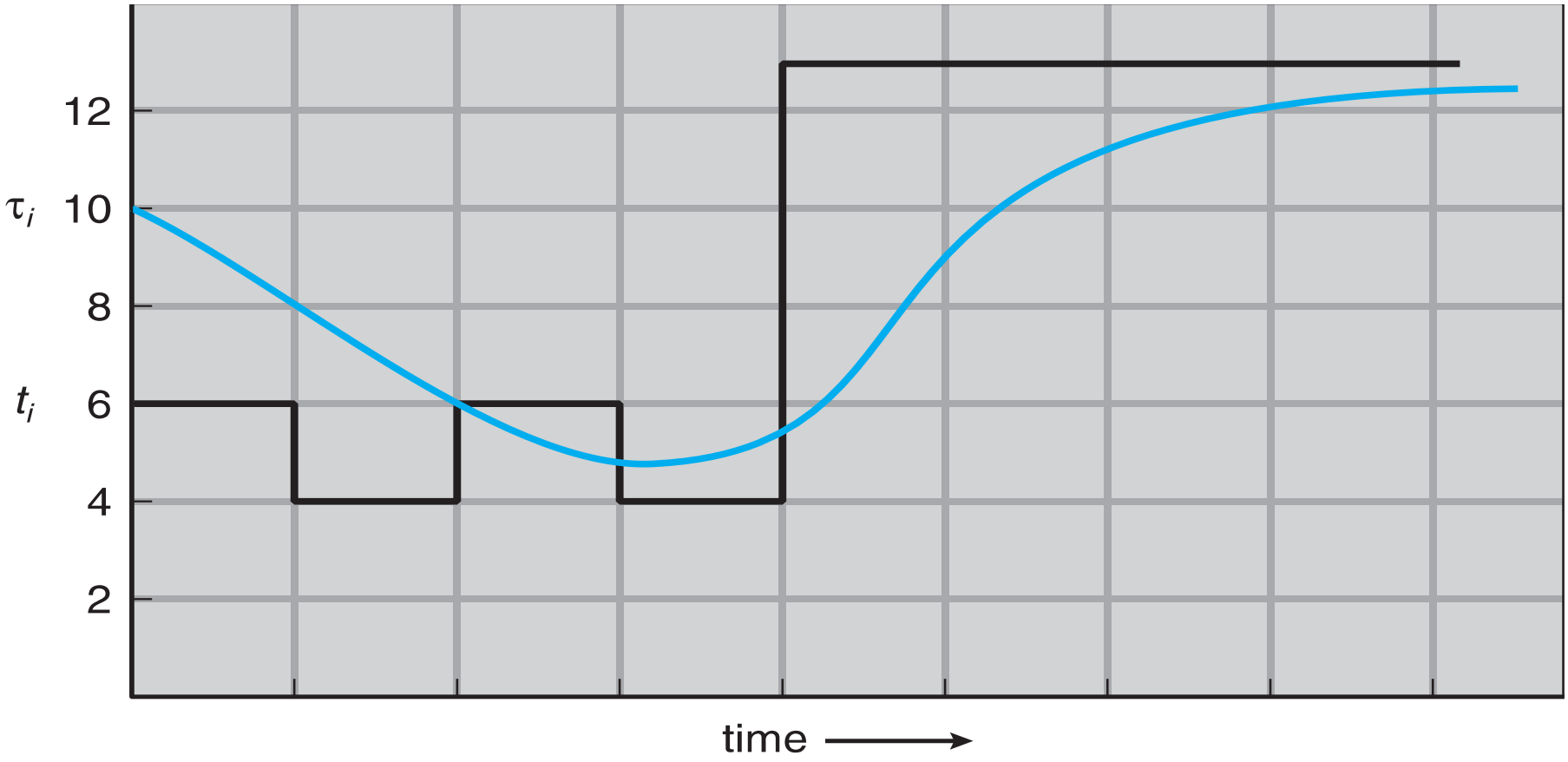
- SJF algo cannot be implemented at the level of the short-term CPU scheduling
  - ***No way*** to know exact length of process's next CPU burst
    - Estimate it using lengths of past bursts: ***next = average of all past bursts***
  - Then pick process with shortest predicted next CPU burst
- Exponential averaging: ***next = average of (past estimate + past actual)***

1. t_n = actual length of n^{th} CPU burst
2. τ_{n+1} = predicted value for the next CPU burst
3. $\alpha, 0 \le \alpha \le 1$
4. Define :

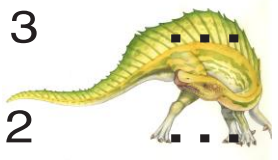
$$\tau_{n=1} = \alpha\, t_n + (1-\alpha)\tau_n .$$

- Relative weight of recent history (t_n) and past history (τ_n): α = ½ **usual** but in [0, 1]
- Preemptive version called **shortest-remaining-time-first** (**SRTF** algorithm)
  - The next burst of new process may be shorter than that left of current process
    - Currently running process will be preempted

Prediction of the Length of the 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
- $\alpha = 1$
  - $\tau_{n+1} = t_n$
  - Only the actual last CPU burst counts
- If we expand the formula, we get:

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

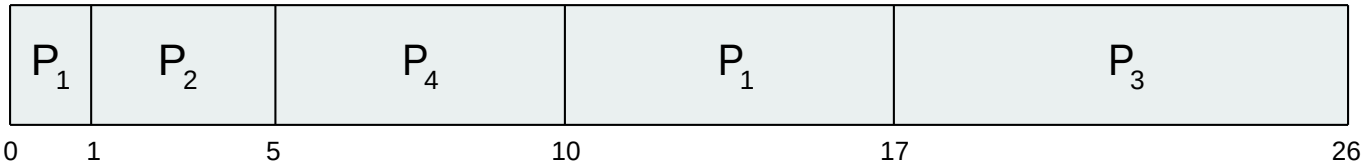
- Since both α and $(1 - \alpha)$ are less than or equal to 1, each successive term has less weight than its predecessor

Example of Shortest-Remaining-Time-First

- 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

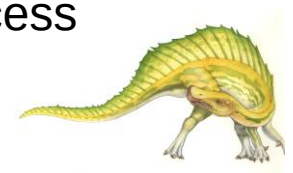


- Average waiting time = [(10-1) + (1-1) + (17-2) + (5-3)]/4 = 26/4 = 6.5 time units

  - Nonpreemptive SJF will result in 7.75 time units

Priority Scheduling

- A priority number (integer) is associated with each process
- The CPU is allocated to the process with the highest priority
  - We assume that ***smallest integer ≡ highest priority***
    - **Internally** defined priority: based on criteria within OS. Ex: memory needs
    - **Externally** defined priority: based on criteria outside OS. Ex: paid process
  - Preemptive: a running process is preempted by a new higher priority process
  - Nonpreemptive: running process holds CPU until termination or I/O request
- SJF is priority scheduling: priority is the inverse of predicted next CPU burst time
- Problem ≡ **Starvation** – some low priority process may never get the CPU
  - Process wait indefinitely in the ready queue due to stream of high priority processes
- Solution ≡ **Aging** – as time progresses increase the priority of the process
  - Example: do ***priority = priority – 1*** every 15 minutes

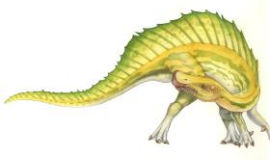


Example of Priority Scheduling

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

- Average waiting time = 8.2 msec



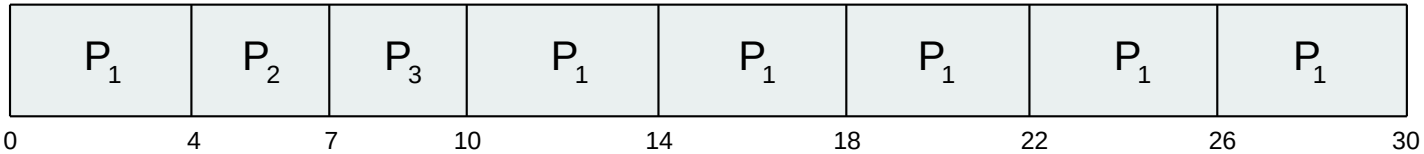
Round Robin (RR)

- Each process gets a small unit of CPU time (**time quantum or time slice** q), usually 10-100 milliseconds. After this time has elapsed, the process is preempted and added to the end of the ready queue.
  - Ready queue is treated as a circular queue
- If there are n processes in the ready queue and the 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 $(n-1) \times q$ time units.
- Timer interrupts at every quantum to schedule next process
- Performance depends on heavily on the size of the time quantum
  - If q very large $\Rightarrow$ RR scheduling = FCFS scheduling
  - If q very small $\Rightarrow$ q must be large with respect to context switch
    - Otherwise overhead of number of context switches will be is too high

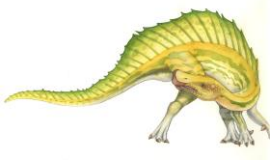
Example of RR with Time Quantum = 4

Process	Burst Time
P_1	24
P_2	3
P_3	3

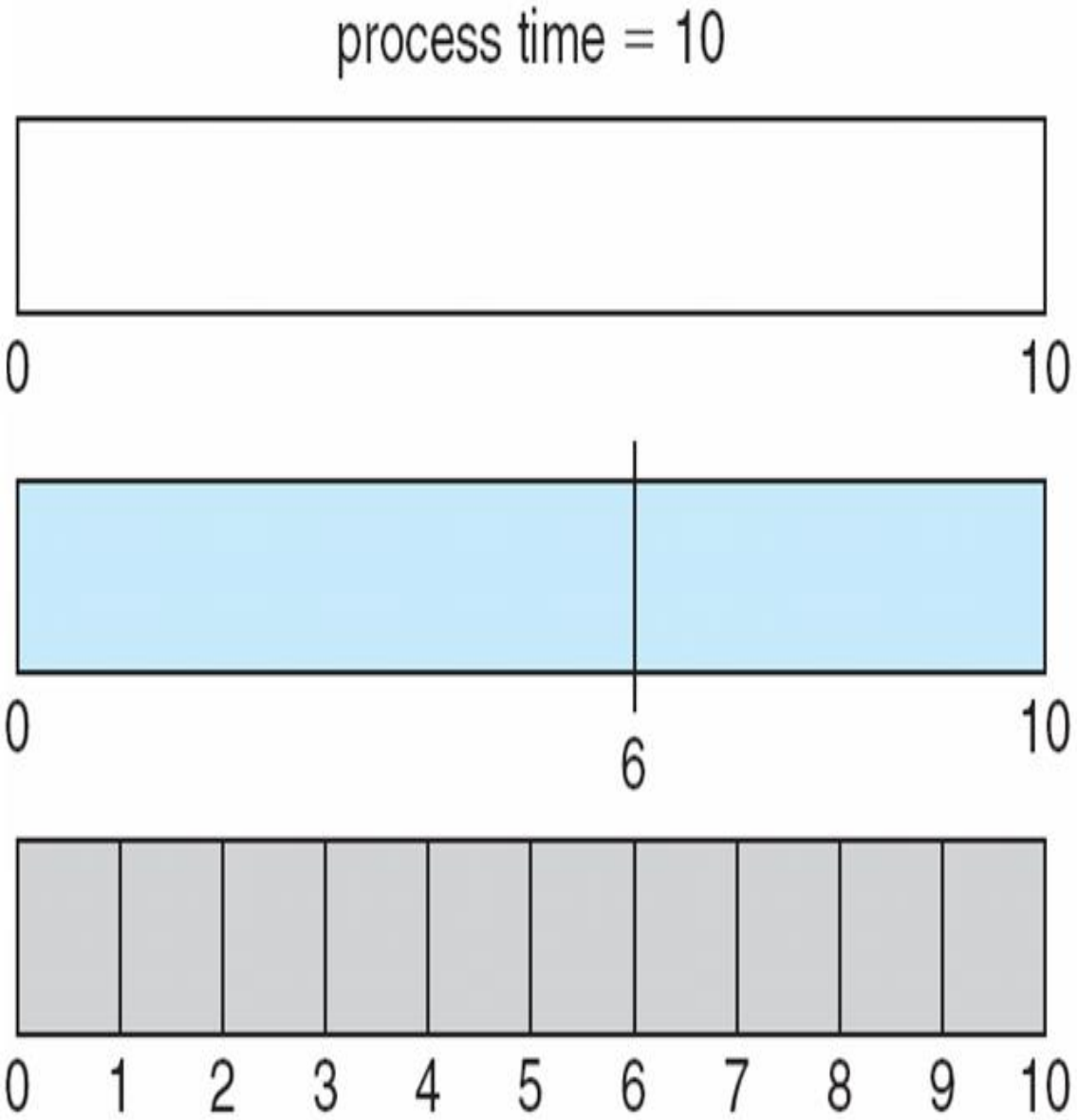
- The Gantt chart is:



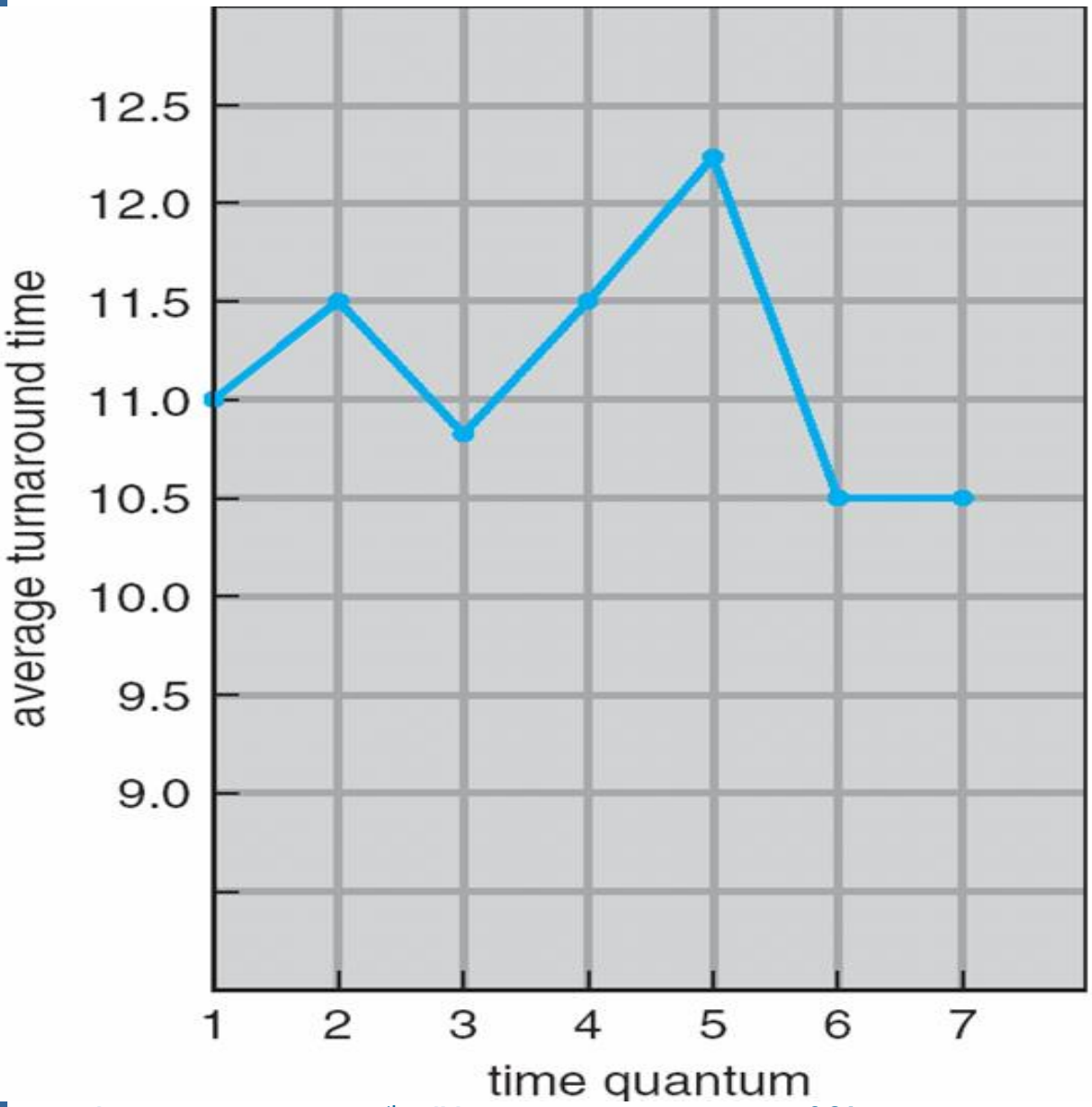
- Average waiting time under RR scheduling is often long
- Typically, higher average turnaround than SJF, but better ***response***
- *q* should be large compared to context switch time
- *q* usually between 10ms to 100ms, context switch < 10 μsec



Time Quantum and Context Switch Time



Turnaround Time Varies With The Time Quantum



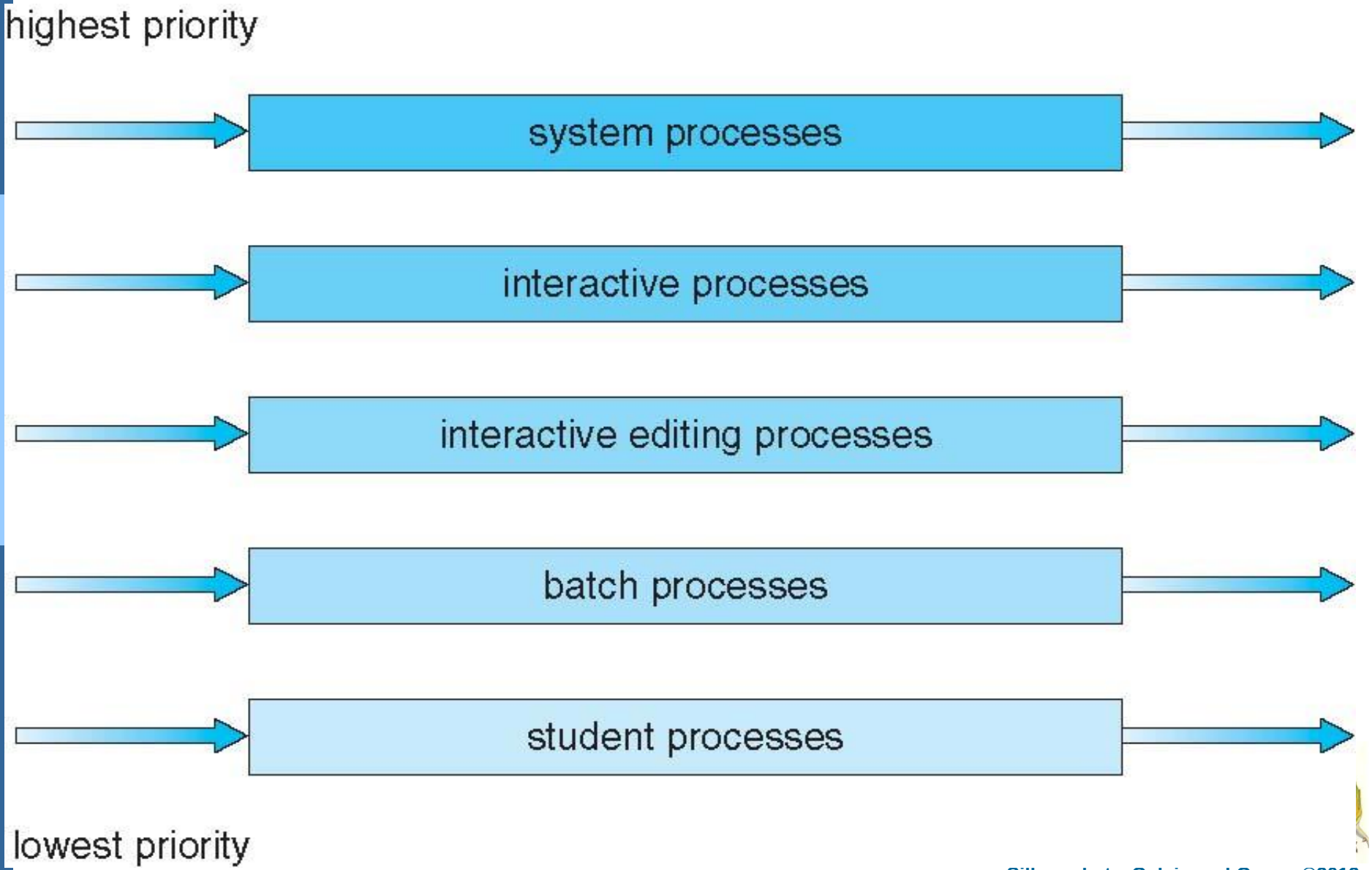
process	time
P_1	6
P_2	3
P_3	1
P_4	7

80% of CPU bursts should be shorter than q

Multilevel Queue

- Ready queue is partitioned into separate queues; e.g., two queues containing
  - **foreground** (interactive) processes
    - May have externally defined priority over background processes
  - **background** (batch) processes
- Process permanently associated to a given queue; no move to a different queue
- Each queue has its own scheduling need, and hence, different algorithm:
  - foreground – RR or …
  - background – FCFS or …
- Must schedule among the queues too (not just processes):
  - 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;
    - 80% to foreground in RR, and, 20% to background in FCFS

Multilevel Queue Scheduling

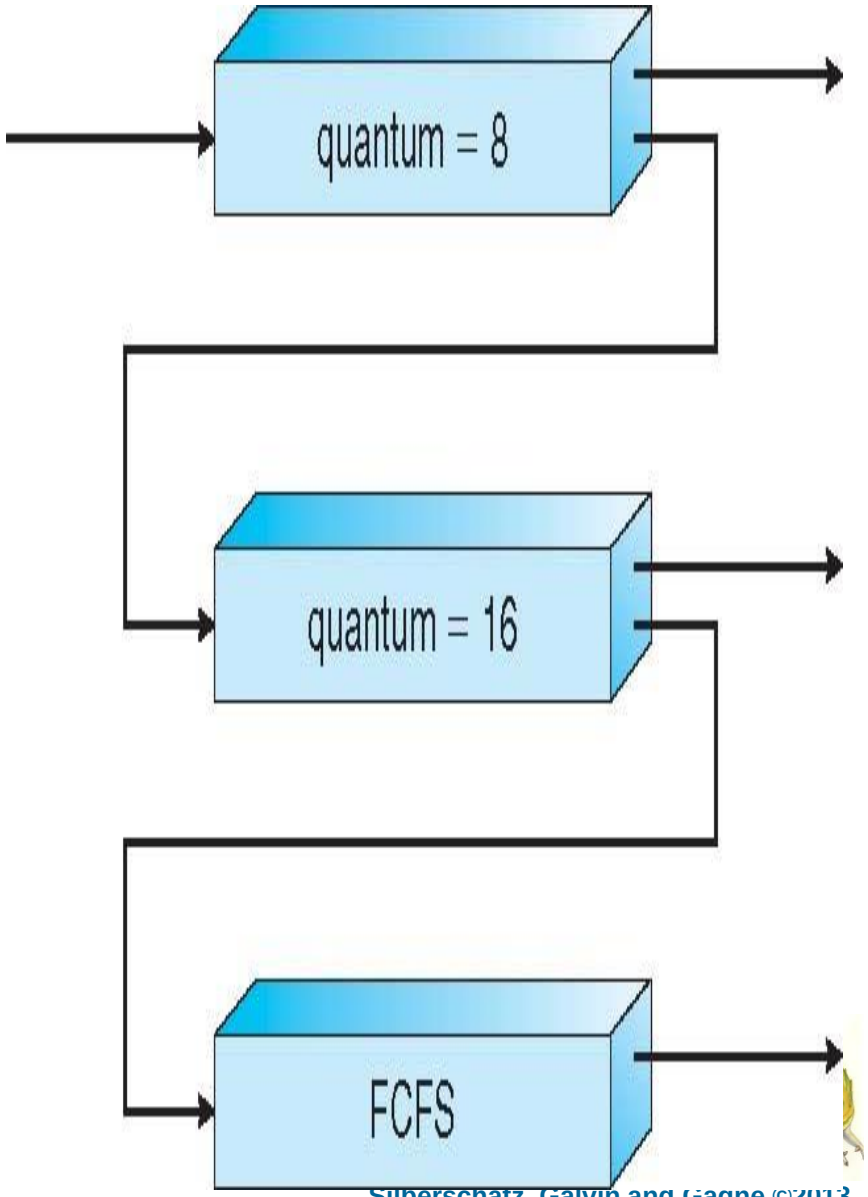


Multilevel Feedback Queue

- A process can move between the queues; aging can be implemented this way
  - Move heavy foreground CPU-bound process to the background queue
  - Move starving background process to the foreground queue
- 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
  - … etc

Example of Multilevel Feedback Queue

- Three queues:
  - Q_0 – RR with time quantum 8 milliseconds
    - Highest priority. Preempts Q_1 and Q_2 proc's
  - Q_1 – RR time quantum 16 milliseconds
    - Medium priority. Preempts processes in Q_2
  - Q_2 – FCFS
    - Lowest priority
- Scheduling
  - A new job enters queue Q_0 which uses RR-8
    - 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 receives 16 additional milliseconds
    - If it still does not complete, it is preempted and moved to queue Q_2

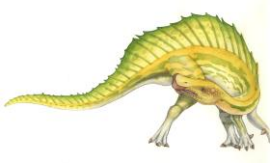


Thread Scheduling

- Distinction between user-level and kernel-level threads
- When OS support threading, then ***kernel threads*** are scheduled; not processes
- On OS using many-to-one or many-to-many mapping models, the thread library schedules ***user threads*** to run on an available lightweight process (LWP)
  - Known as **process-contention scope (PCS)** since scheduling competition is within the process; i.e., among user threads belonging to the same process
    - Which user thread goes to the kernel…?
  - Typically done via user thread priorities set by the programmer
- Kernel thread scheduled onto available CPU is **system-contention scope (SCS)**
  - Competition is among all kernel threads in the system
  - Also on OS using one-to-one mapping model

Pthread Scheduling

- API allows specifying either PCS or SCS during thread creation
  - PTHREAD_SCOPE_PROCESS schedules threads using PCS scheduling
  - PTHREAD_SCOPE_SYSTEM schedules threads using SCS scheduling
- Can be limited by OS – Linux and Mac OS X only allow PTHREAD_SCOPE_SYSTEM

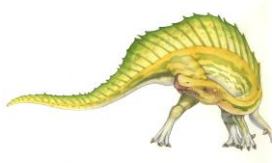


Pthread Scheduling API

```
#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");
   }
```

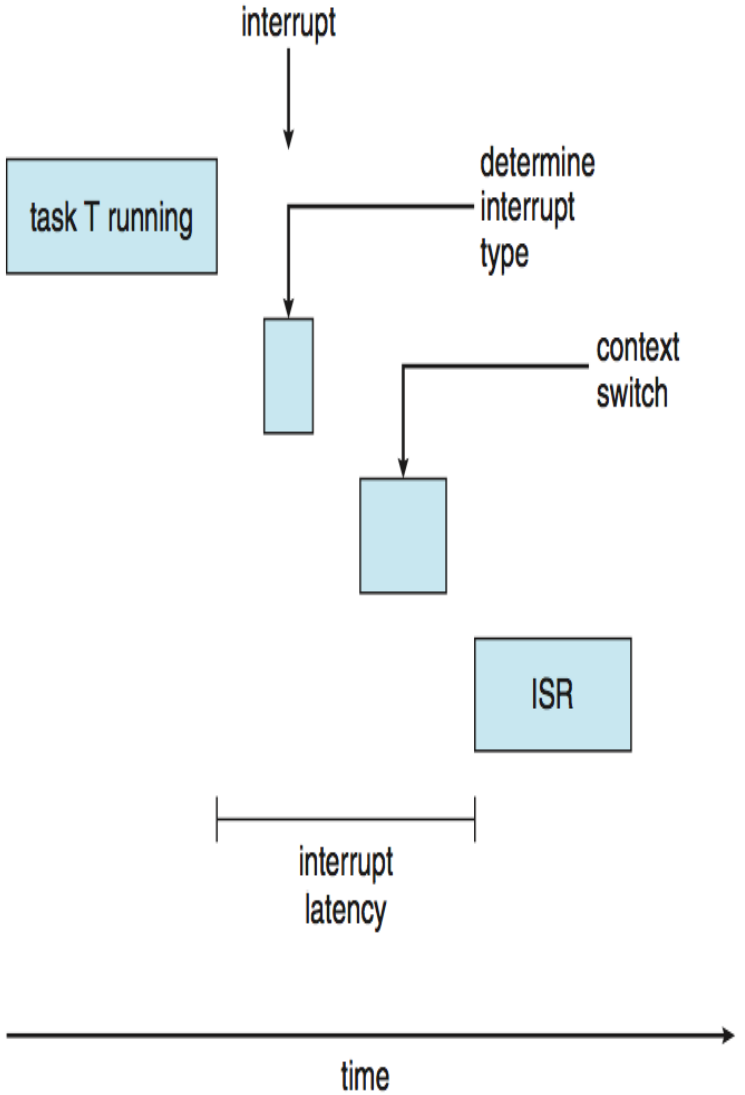
Pthread Scheduling API

```
   /* 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);
}
```



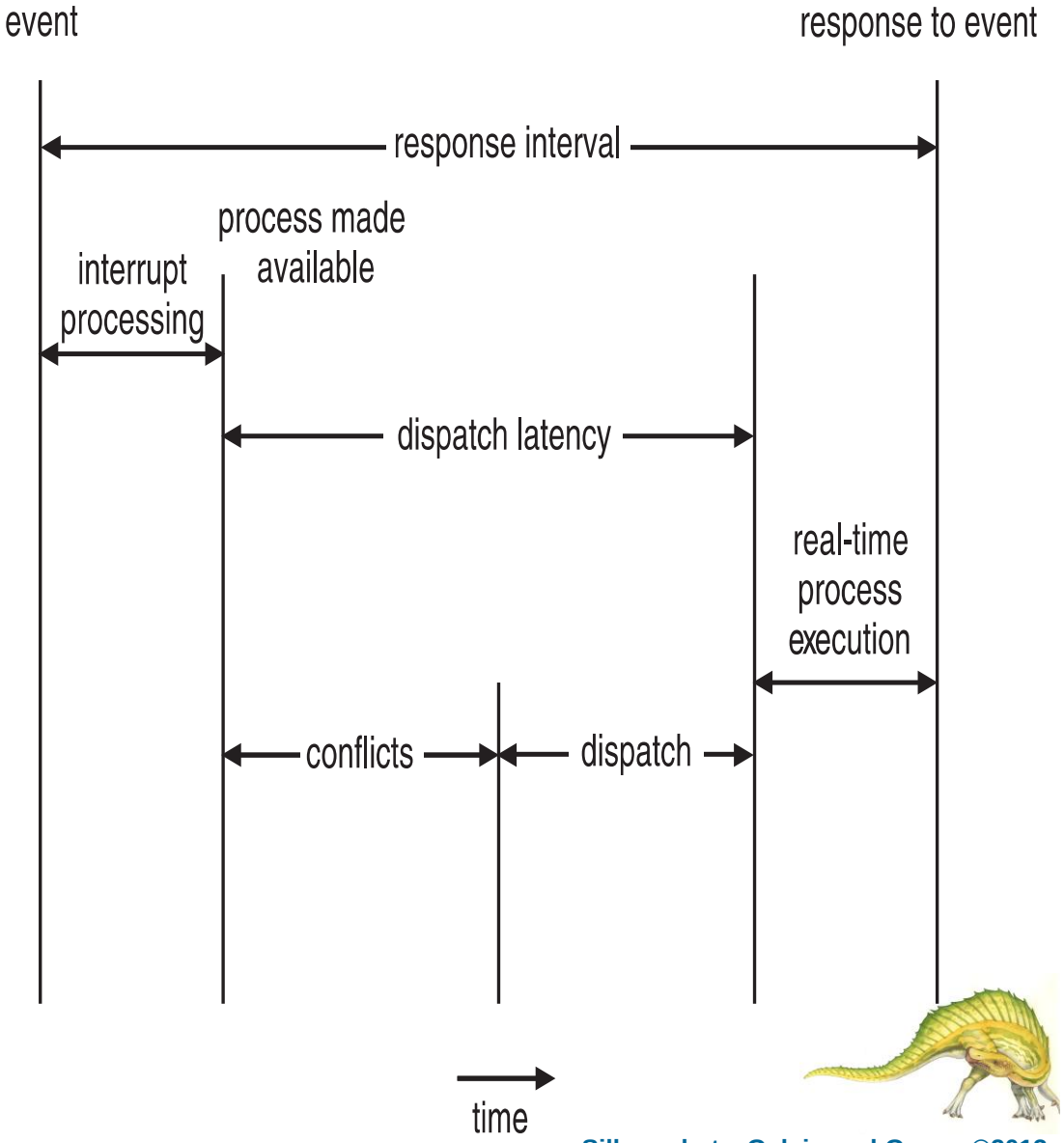
Real-Time CPU Scheduling

- Can present obvious challenges
- **Soft real-time systems** – no guarantee as to when critical real-time process will be scheduled
- **Hard real-time systems** – task must be serviced by its deadline
- **Event latency**: [occurred to serviced]
- Two types of event latencies affect performance
  1. Interrupt latency – time from arrival of interrupt to start of routine that services interrupt
  2. Dispatch latency – time for dispatcher to take current process off CPU and switch to another
- Want to minimize both latencies



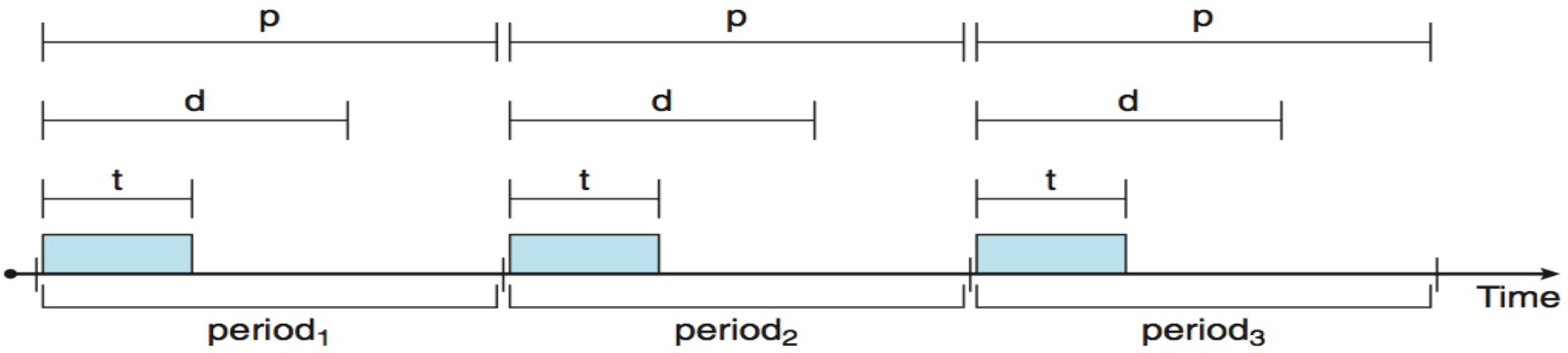
Real-Time CPU Scheduling [Dispatch Latency]

- Conflict phase of dispatch latency:
  1. Preemption of any process running in kernel mode
  2. Release by low-priority processes of resources needed by a high-priority process



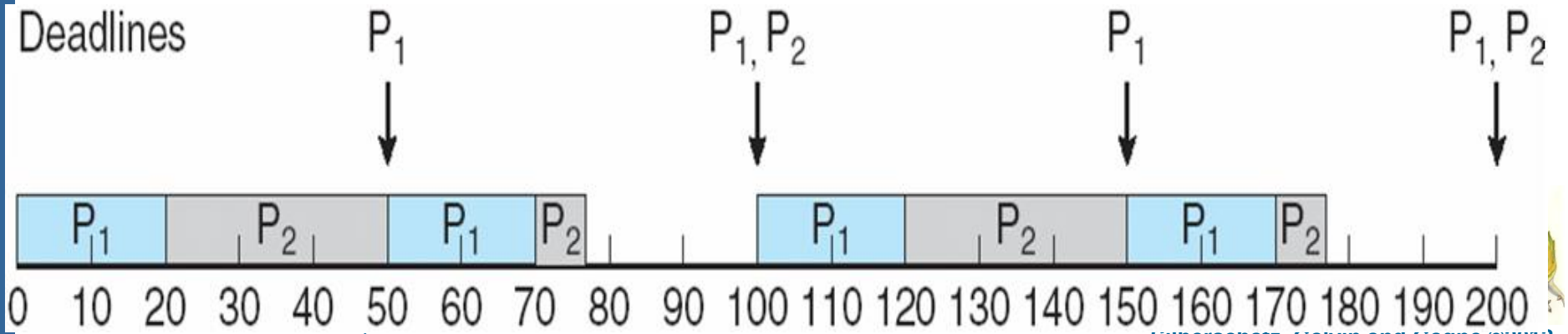
Priority-Based Scheduling

- Real-time OS responds ***immediately*** to a real-time process when it request CPU
  - Real-time scheduler must support ***preemptive*** priority-based algorithm
    - But only guarantees soft real-time functionality
  - ***Hard real-time systems*** must also provide ability to meet task deadlines
    - **Periodic** processes require CPU at constant intervals
      - Each process has a processing time *t*, a deadline *d*, and a period *p*
      - $0 \leq t \leq d \leq p$
      - **Rate** of periodic task is $1/p$
    - **Admission-control**: process announces it requirements, then scheduler admits the process if it can complete it on time, or, reject it if it cannot serviced it by deadline



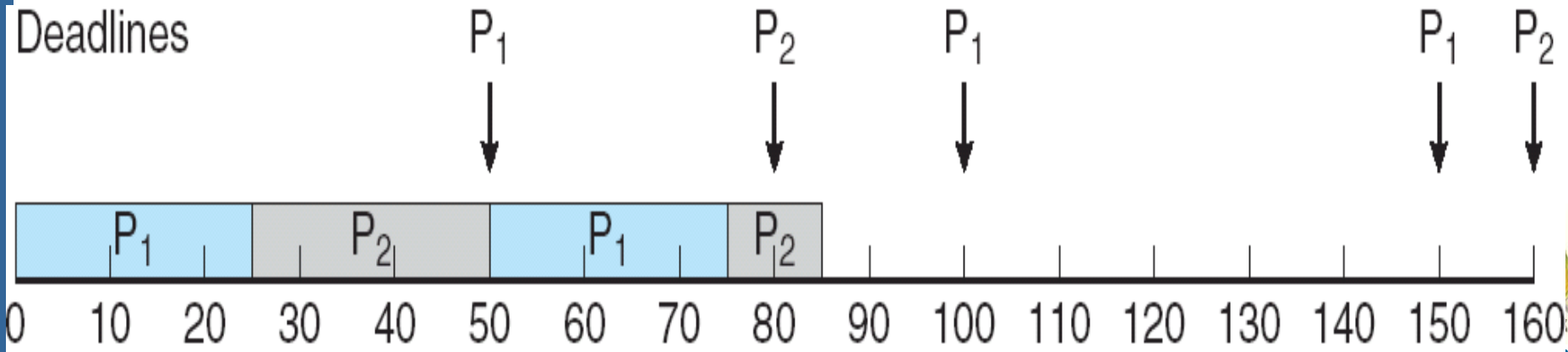
Rate-Monotonic Scheduling

- Static priority policy with preemption
  - A priority is assigned to a process based on the inverse of its period *p*
    - Shorter periods = higher priority; Longer periods = lower priority
    - Assumes processing time *t* is the same for each CPU burst
    - Rationale: higher priority to tasks that require the CPU more often
- Example:
  - Process P_1 : $t_1 = 20$, d_1 = complete CPU burst by start of next period, $p_1 = 50$.
  - Process P_2 : $t_2 = 35$, d_2 = complete CPU burst by start of next period, $p_2 = 100$.
  - P_1 is assigned a higher priority than P_2.
  - **CPU utilization = t_i / p_i**. Thus total CPU utilization = 20/50 + 35/100 = 75%



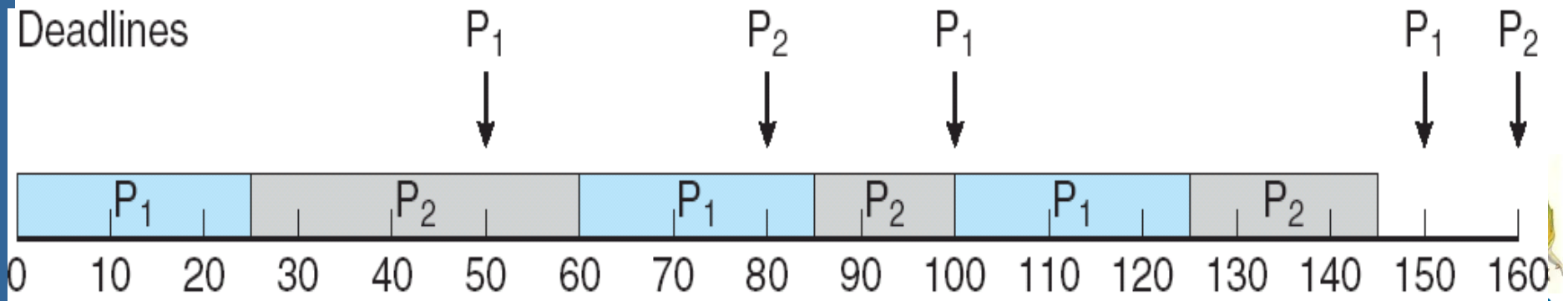
Missed Deadlines with Rate-Monotonic Scheduling

- Process P_1 : $t_1 = 25$, d_1 = complete CPU burst by start of next period, $p_1 = 50$.
- Process P_2 : $t_2 = 35$, d_2 = complete CPU burst by start of next period, $p_2 = 80$.
- Total CPU utilization = 25/50 + 35/80 = **94%**
- P_1 is assigned a higher priority than P_2.
- **P_2 has missed the deadline for completion of its CPU burst at time 80**
- Reason - **bounded** CPU utilization; worst case for *N* processes: $B = N(2^{1/N} - 1)$
  - Bounded at $B = 83\%$ for $N = 2$ processes
  - ***Theory***: cannot guarantee that processes can be scheduled to meet their deadlines if **actual CPU utilization > *Bound B***
    - 94% > 83% in the example above



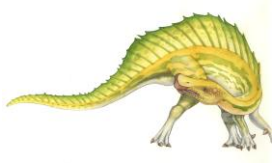
Earliest Deadline First Scheduling (EDF)

- Dynamic priority policy with preemption
  - Priorities are **dynamically** assigned according to deadlines:
    - Earlier deadline = higher priority; later deadline = lower priority
  - New runnable process must announce its deadline requirements to scheduler
    - Scheduler will **adjust current priorities** accordingly
  - Process need not be periodic
  - CPU burst time need not be constant
- Process P_1 : $t_1 = 25$, d_1 = complete CPU burst by start of next period, $p_1 = 50$.
- Process P_2 : $t_2 = 35$, d_2 = complete CPU burst by start of next period, $p_2 = 80$.
- P_1 is **initially** assigned a higher priority than P_2 since $d_1 = 50 \leq d_2 = 80$



Proportional Share Scheduling

- *T* shares of time are allocated among all processes in the system
- An application receives *N* shares of time where *N* < *T*
- This ensures each application will receive ***N*** / *T* of the total processor time
- Example: three processes A, B and C, with *T* = 100 shares
  - A, B and C assigned each 50, 15 and 20 shares, respectively
  - Thus A will have 50% of total processor times, and so on with B and C
- Scheduler must use admission-control policy:
  - Admit a request only if sufficient shares are available
  - Example: If new process D needs 30 share then scheduler will deny him CPU
    - Current total share is 50 + 15 + 20 = 85
    - Only a new process with share < 15 can be scheduled



POSIX Real-Time Scheduling

- The 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

- 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)`

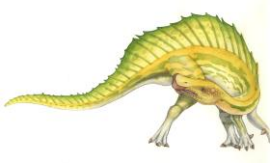
POSIX Real-Time Scheduling API

```
#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.)

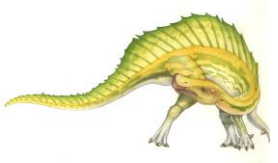
```
   /* set the scheduling policy - FIFO, RR, or OTHER */
   if (pthread_attr_setschedpolicy(&attr, SCHED_FIFO) != 0)
      fprintf(stderr, "Unable to set policy.\n");
   /* create the threads */
   for (i = 0; i < NUM_THREADS; i++)
      pthread_create(&tid[i],&attr,runner,NULL);
   /* now join on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

- Linux scheduling
- Windows scheduling
- Solaris scheduling

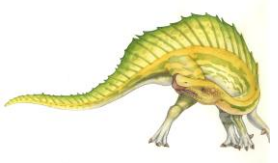


Linux Scheduling Through Version 2.5

- 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
  - 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

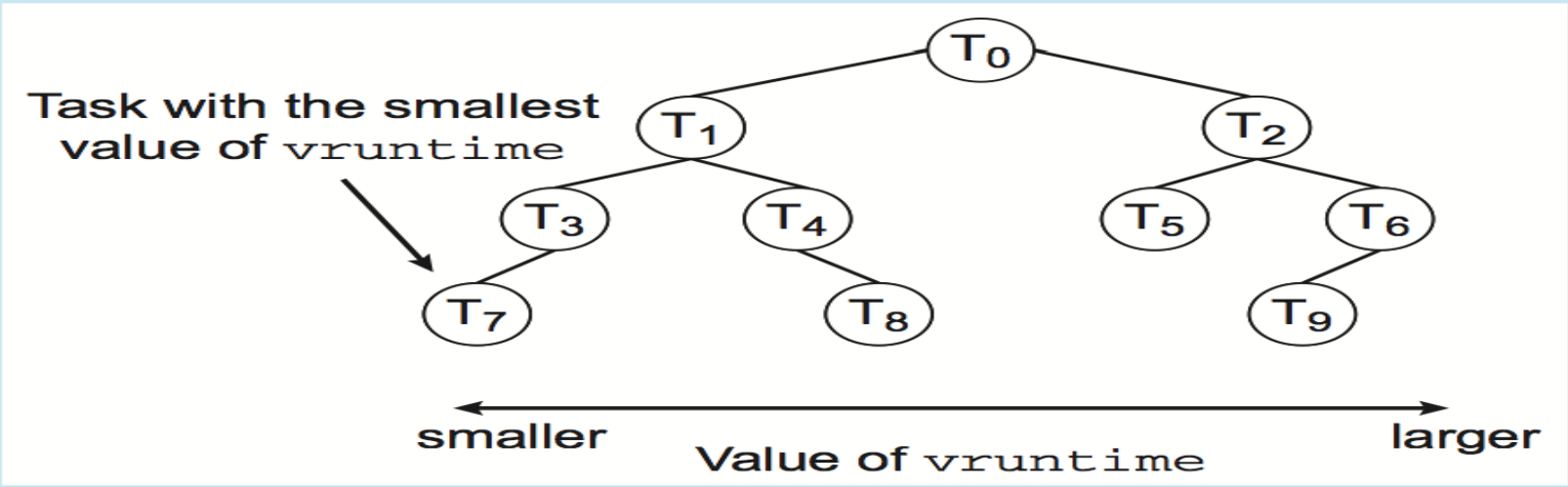
Linux Scheduling in Version 2.6.23 +

- ***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
- 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
- To decide next task to run, scheduler picks task with lowest virtual run time



CFS Performance

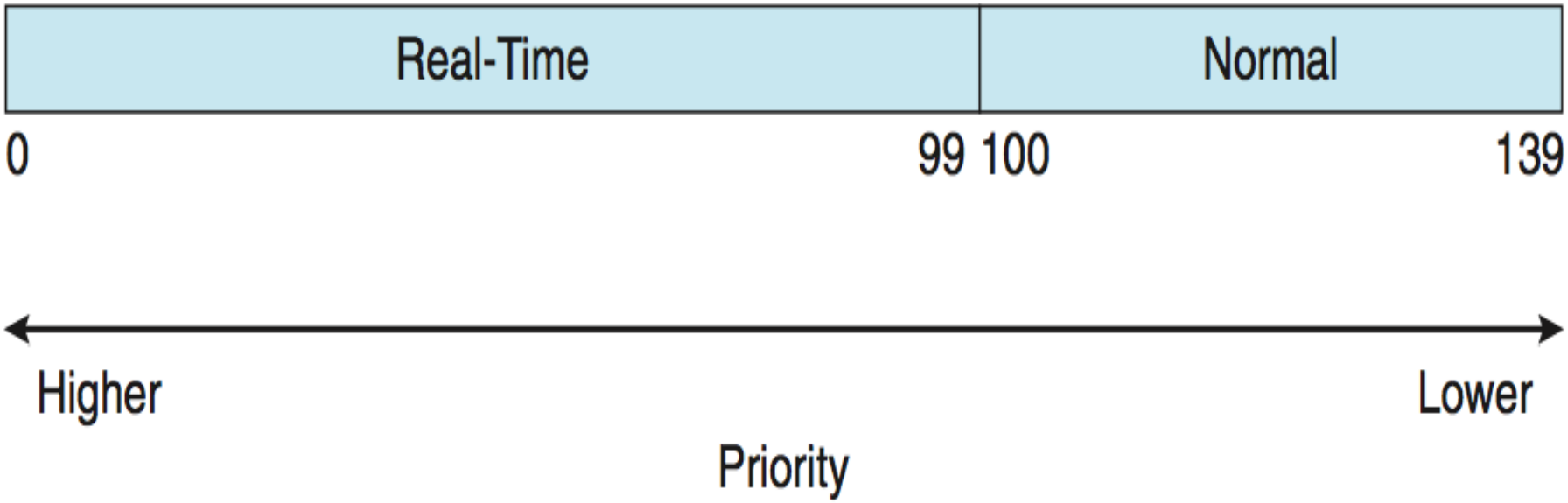
The Linux CFS scheduler provides an efficient algorithm for selecting which task to run next. Each runnable task is placed in a red-black tree—a balanced binary search tree whose key is based on the value of `vruntime`. This tree is shown below:



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.

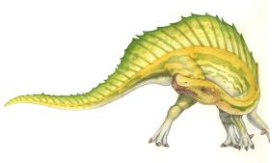
Linux Scheduling

- 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



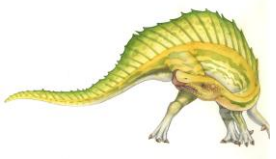
Windows Scheduling

- 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**



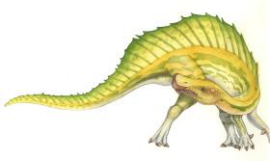
Windows Priority Classes

- 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

- 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

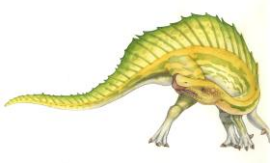


Windows Priorities

	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

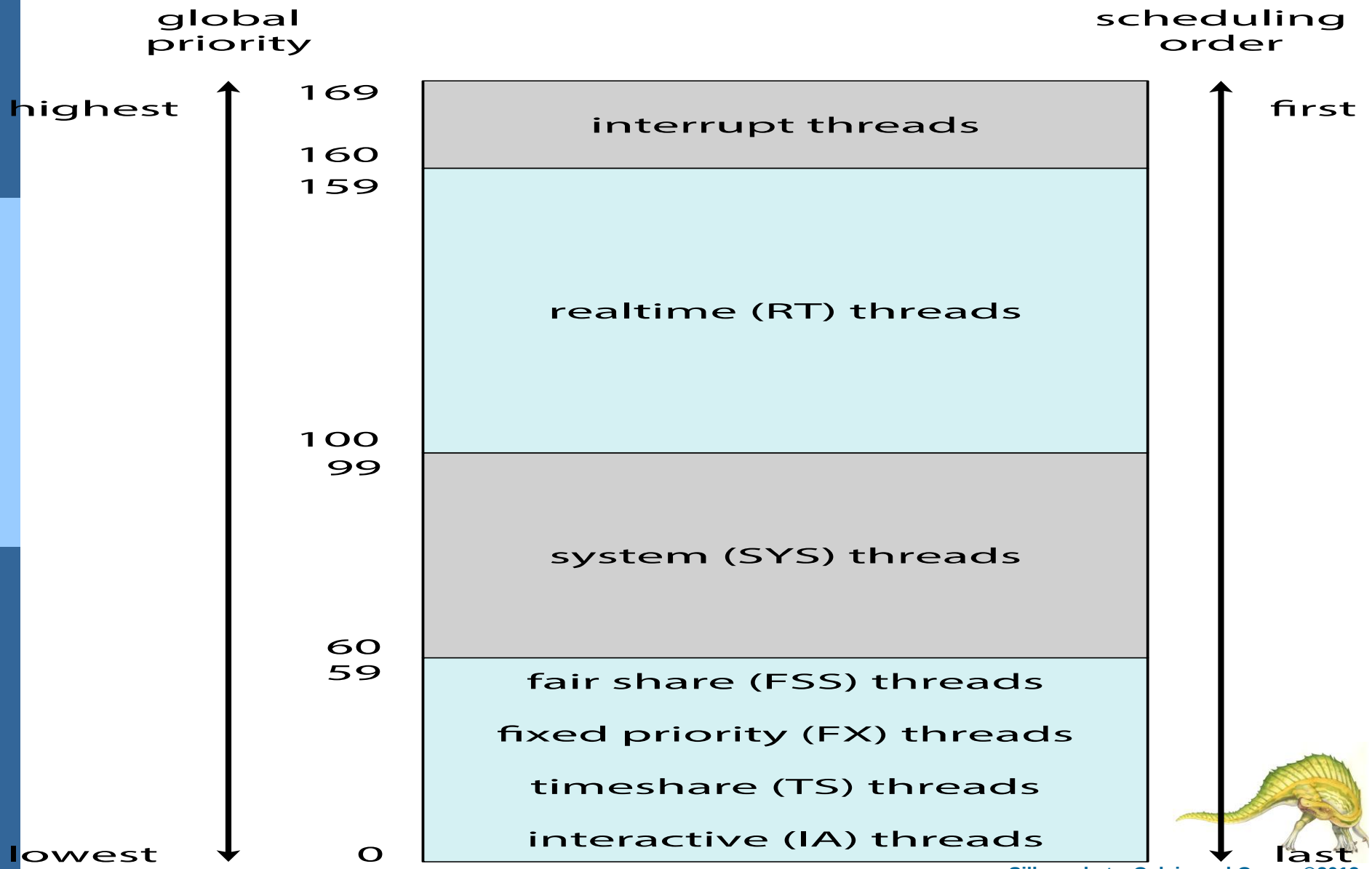
- Priority-based scheduling
- Six classes available
  - Time sharing (default) (TS)
  - Interactive (IA)
  - Real time (RT)
  - System (SYS)
  - Fair Share (FSS)
  - Fixed priority (FP)
- Given thread can be in one class at a time
- Each class has its own scheduling algorithm
- Time sharing is multi-level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



Solaris Dispatch Table

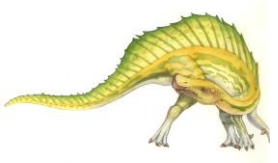
priority	time quantum	time quantum expired	return from sleep
0	200	0	50
5	200	0	50
10	160	0	51
15	160	5	51
20	120	10	52
25	120	15	52
30	80	20	53
35	80	25	54
40	40	30	55
45	40	35	56
50	40	40	58
55	40	45	58
59	20	49	59

Solaris Scheduling

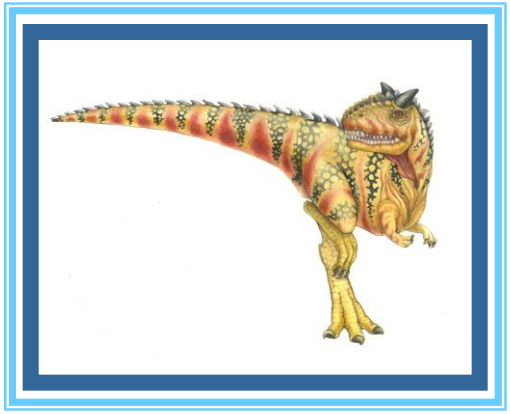


Solaris Scheduling

- 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



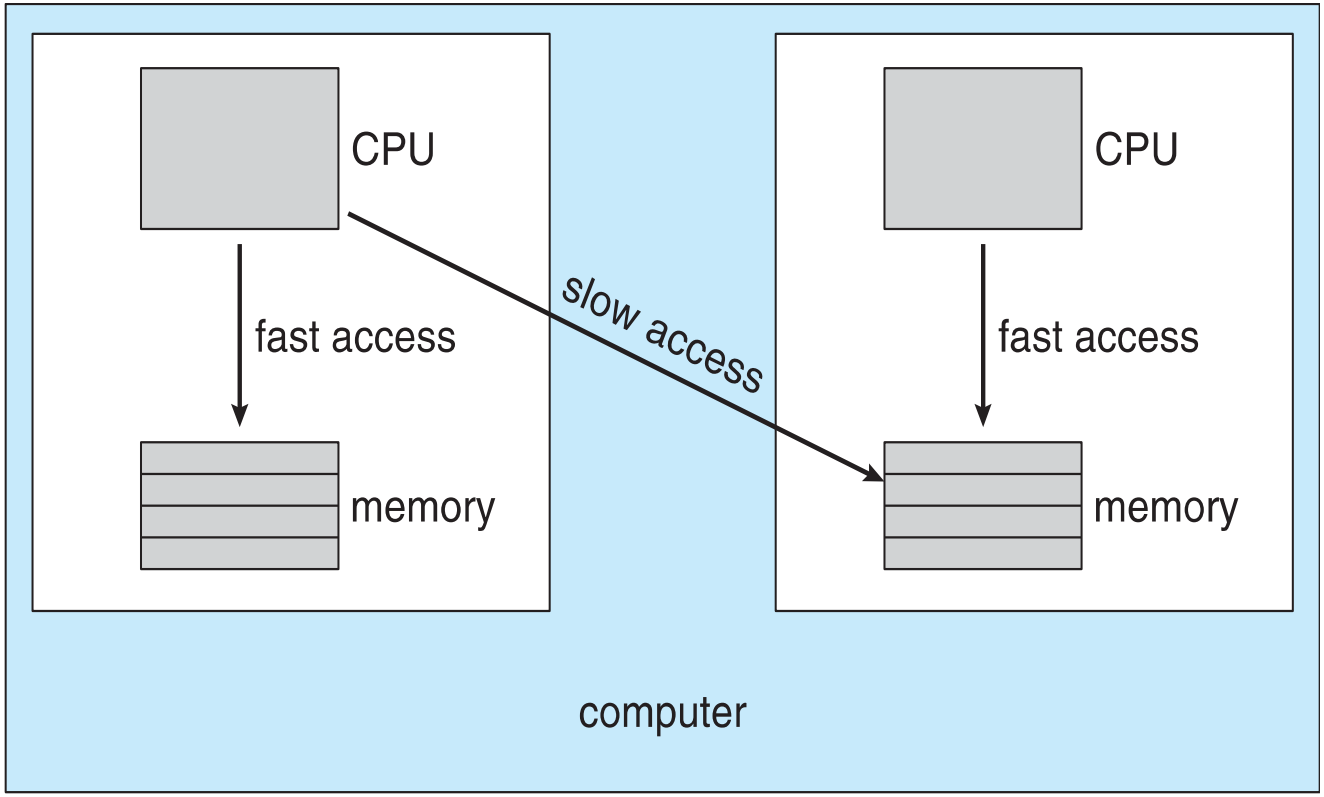
End of Chapter 6



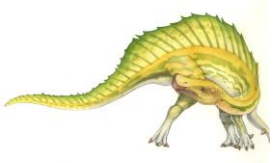
Multiple-Processor Scheduling

- CPU scheduling more complex when multiple CPUs are available
- **Homogeneous processors** within a multiprocessor
- **Asymmetric multiprocessing** – only one processor accesses the system data structures, alleviating the need for data sharing
- **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
- **Processor affinity** – process has affinity for processor on which it is currently running
  - **soft affinity**
  - **hard affinity**
  - Variations including **processor sets**

NUMA and CPU Scheduling

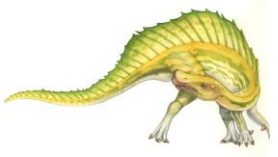


Note that memory-placement algorithms can also consider affinity



Multiple-Processor Scheduling – Load Balancing

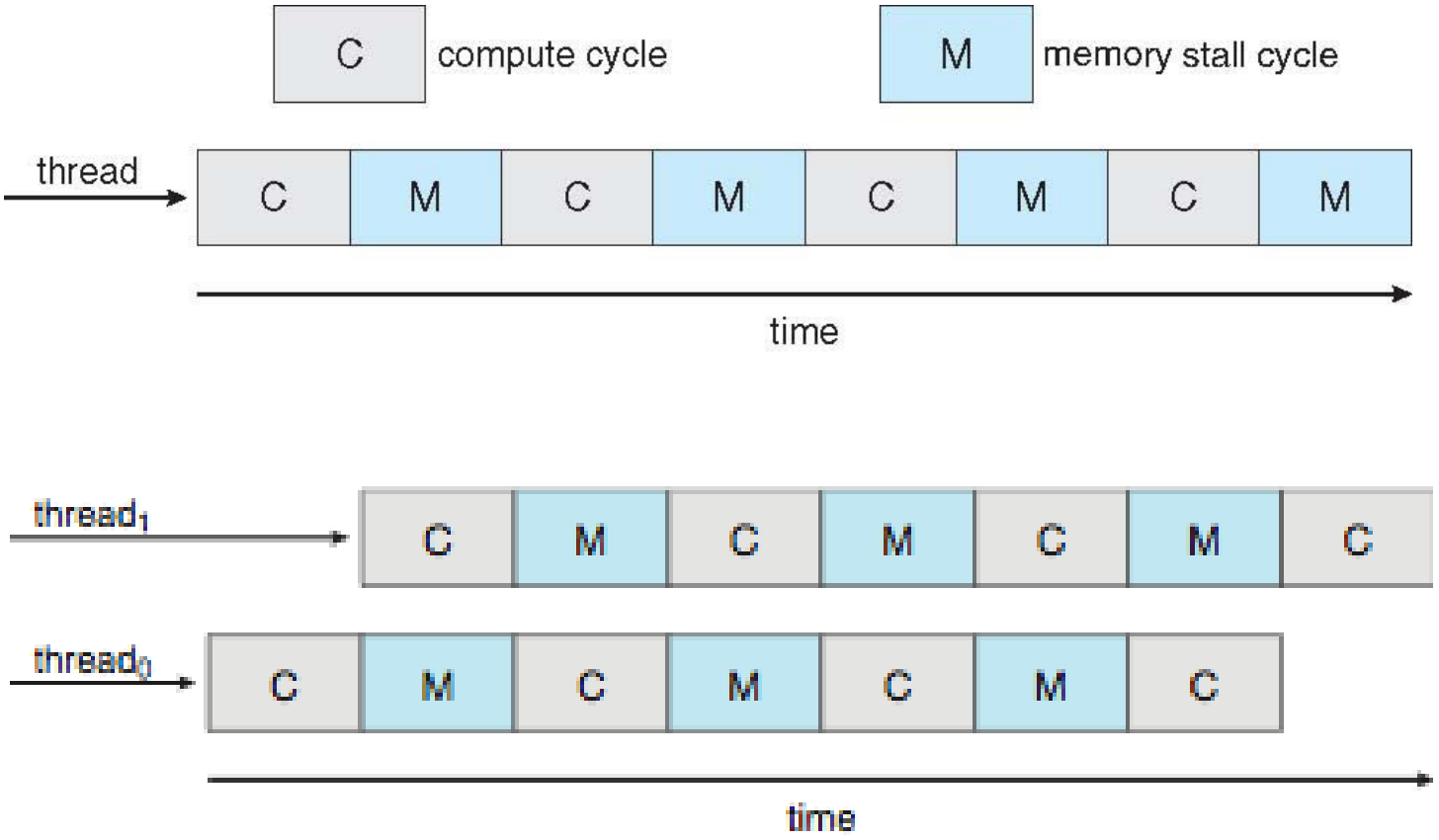
- If SMP, need to keep all CPUs loaded for efficiency
- **Load balancing** attempts to keep workload evenly distributed
- **Push migration** – periodic task checks load on each processor, and if found pushes task from overloaded CPU to other CPUs
- **Pull migration** – idle processors pulls waiting task from busy processor

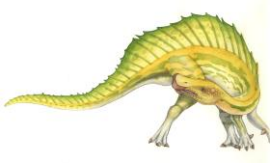


Multicore Processors

- 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

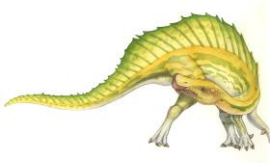
Multithreaded Multicore System





Virtualization and Scheduling

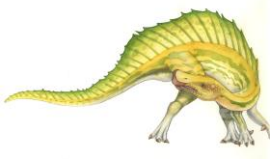
- Virtualization software schedules multiple guests onto CPU(s)
- Each guest doing its own scheduling
  - Not knowing it doesn't own the CPUs
  - Can result in poor response time
  - Can effect time-of-day clocks in guests
- Can undo good scheduling algorithm efforts of guests



Algorithm Evaluation

- How to select CPU-scheduling algorithm for an OS?
- Determine criteria, then evaluate algorithms
- **Deterministic modeling**
  - Type of **analytic evaluation**
  - Takes a particular predetermined workload and defines the performance of each algorithm for that workload
- Consider 5 processes arriving at time 0:

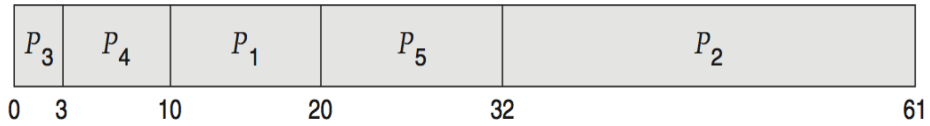
Process	Burst Time
P_1	10
P_2	29
P_3	3
P_4	7
P_5	12



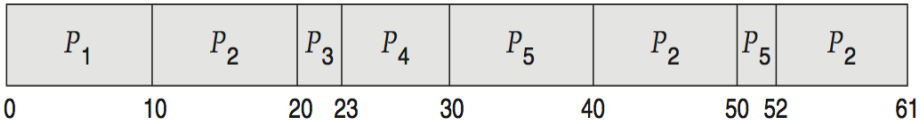
Deterministic Evaluation

- 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:

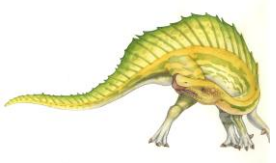


  - RR is 23ms:



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:

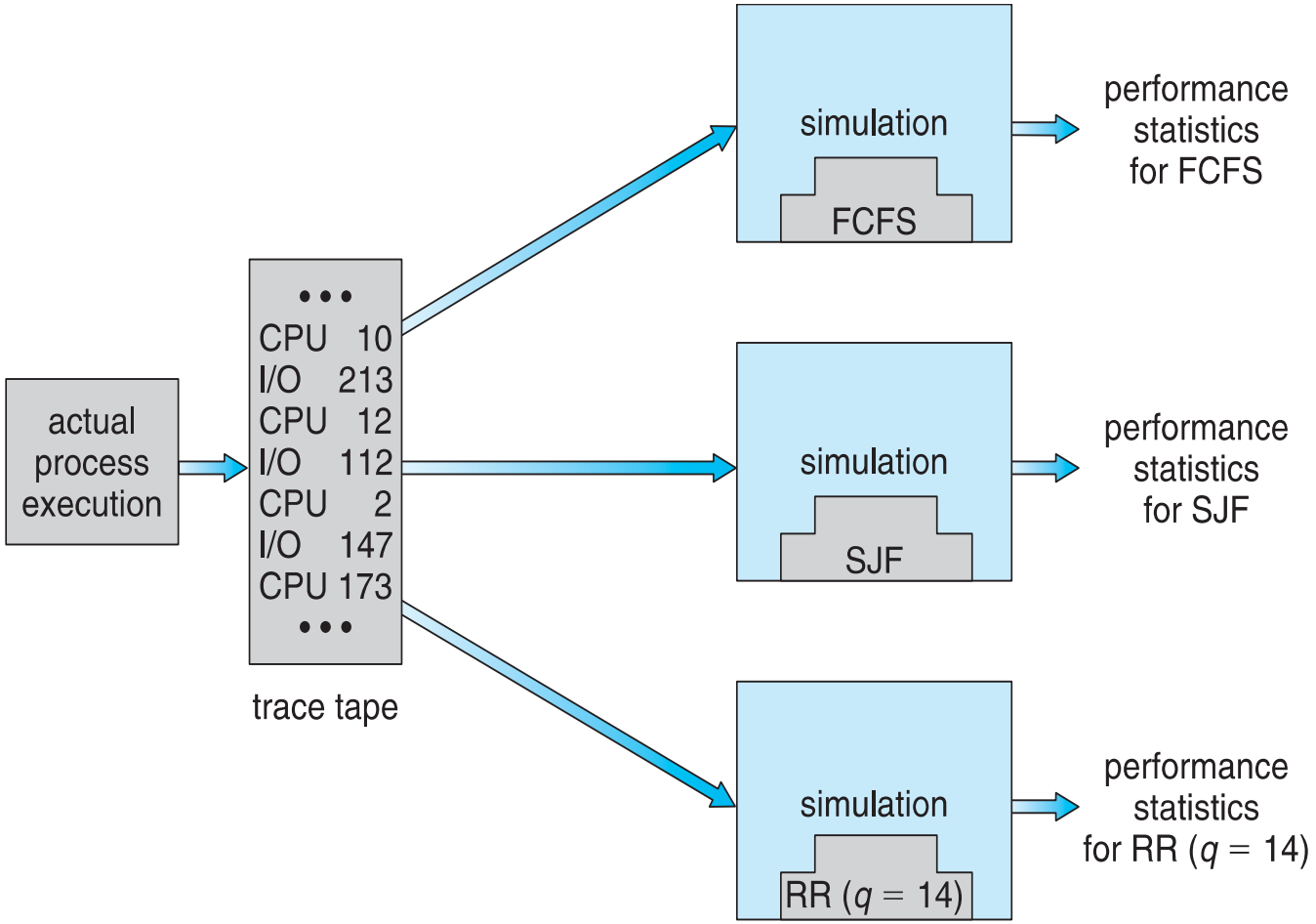
  $n = \lambda \times W$

  - 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