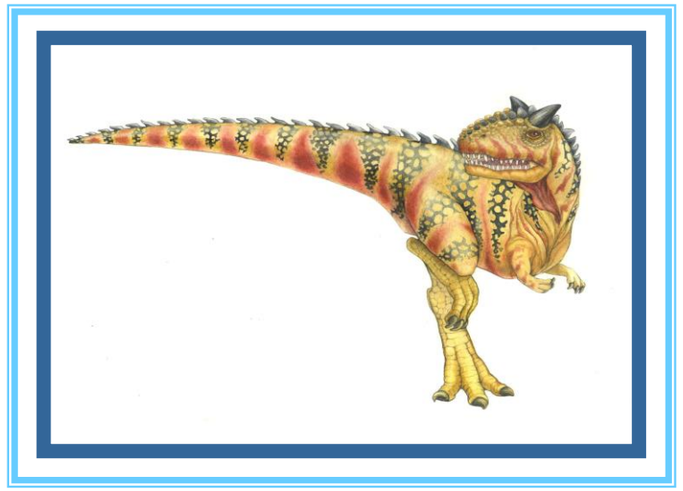
Chapter 5: Process Scheduling

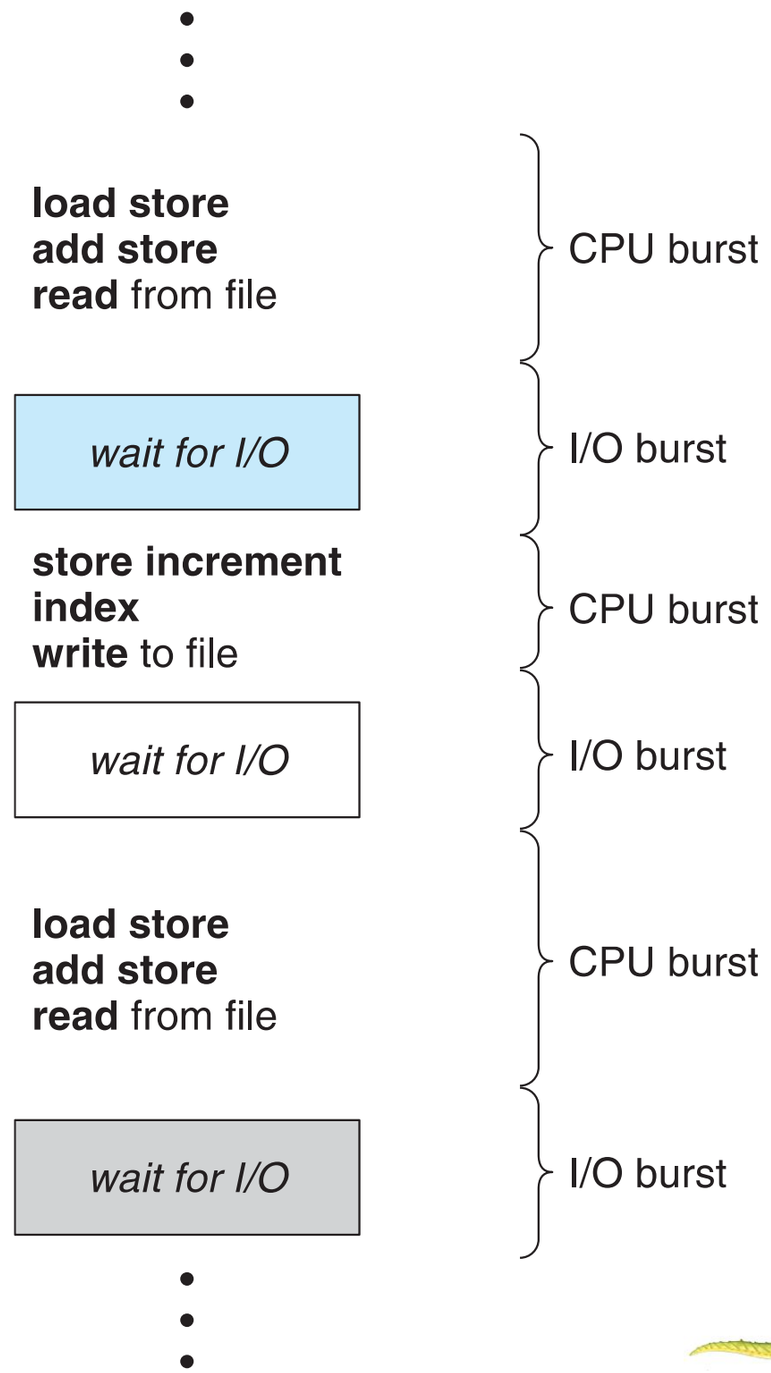


Chapter 5: Process Scheduling

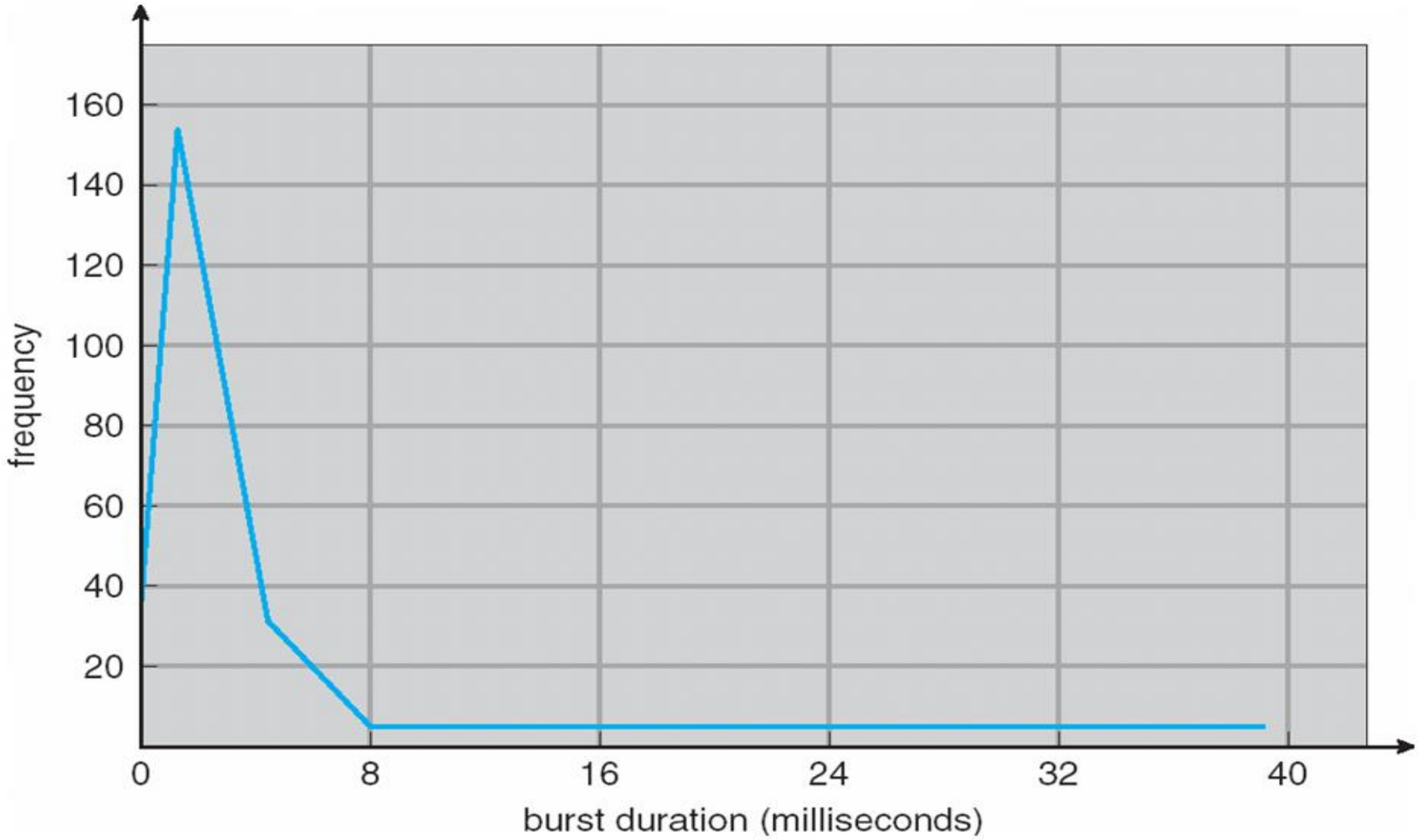
- Basic Concepts
- Scheduling Criteria
- Scheduling Algorithms
- Thread Scheduling
- Multiple-Processor Scheduling
- Real-Time CPU Scheduling
- Operating Systems Examples
- Algorithm Evaluation
- **Objectives**
  - To introduce CPU scheduling, which is the basis for multiprogrammed operating systems
  - 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

- Maximum CPU utilization obtained with multiprogramming
- Process execution consists of a **cycle** of CPU execution and I/O wait
- **CPU burst** followed by **I/O burst**
- CPU burst distribution is of main concern



Histogram of CPU-burst Times



- An I/O-bound program typically has many short CPU bursts.
- A CPU-bound program might have a few long CPU bursts.
- The distribution of CPU burst can be important in the selection of an appropriate CPU-scheduling algorithm.

CPU Scheduler

- **Short-term scheduler (or CPU scheduler)** selects from among the processes in ready queue, and allocates the CPU to one of them
  - Queue may be ordered in various ways (not necessary to be 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
  4. Terminates
- Scheduling occurs only under 1 and 4 is **nonpreemptive (or cooperative),** otherwise; the scheduling is **preemptive**.
- Preemptive scheduling has some issues
  - preempted while access to shared data
  - preempted while in kernel mode

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

Scheduling Criteria

- **CPU utilization** – keep the 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 (from the time of process submission to the time of completion)
- **Waiting time** – amount of time a process has been waiting in the ready queue
- **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)
- Scheduling algorithm optimization criteria
  - Max CPU utilization, Max throughput
  - Min turnaround time, Min waiting time, Min response time

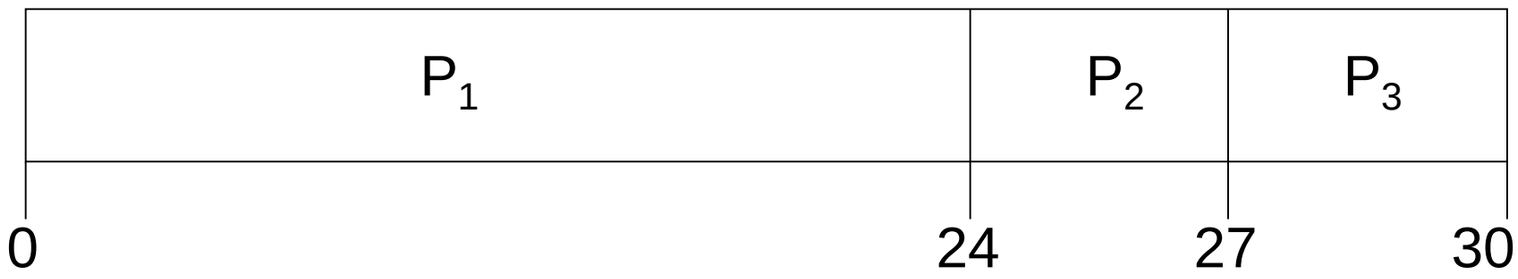
Chapter 5: Process Scheduling

- Basic Concepts
- Scheduling Criteria
- Scheduling Algorithms
- Thread Scheduling
- Multiple-Processor Scheduling
- Real-Time CPU Scheduling
- Operating Systems Examples
- Algorithm Evaluation
- **Objectives**
  - To introduce CPU scheduling, which is the basis for multiprogrammed operating systems
  - 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

First-Come, First-Served (FCFS) Scheduling

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:



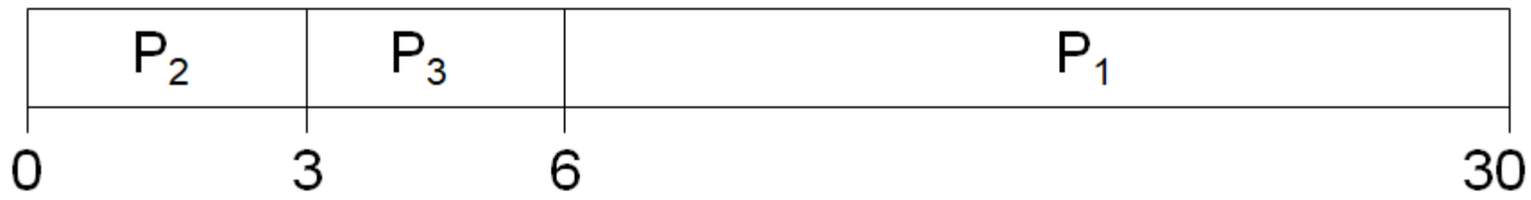
- Waiting time for P_1 = 0; P_2 = 24; P_3 = 27
- Average waiting time: (0 + 24 + 27)/3 = 17

FCFS Scheduling (Cont.)

Suppose that the processes arrive in the order:

$$P_2 , P_3 , P_1$$

- The Gantt chart for the schedule is:



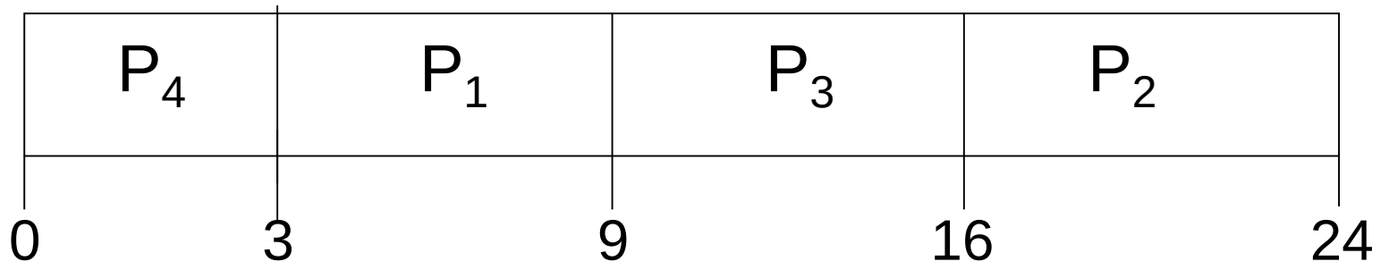
- Waiting time for $P_1 = 6; P_2 = 0, P_3 = 3$
- Average waiting time: (6 + 0 + 3)/3 = 3
- Much better than previous case
- **Convoy effect** - short process behind long process
  - Consider one CPU-bound and many I/O-bound processes
- FCFS is a non-preemptive algorithm

Shortest-Job-First (SJF) Scheduling

- Associate with each process the length of its next CPU burst
  - Use these lengths to schedule the process with the shortest time
- 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 the user
- e.g.

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

- To estimate the length of next CPU burst – should be similar to previous one
  - Then pick process with shortest predicted next CPU burst
- Can be done by using the length of previous CPU bursts, using exponential averaging

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

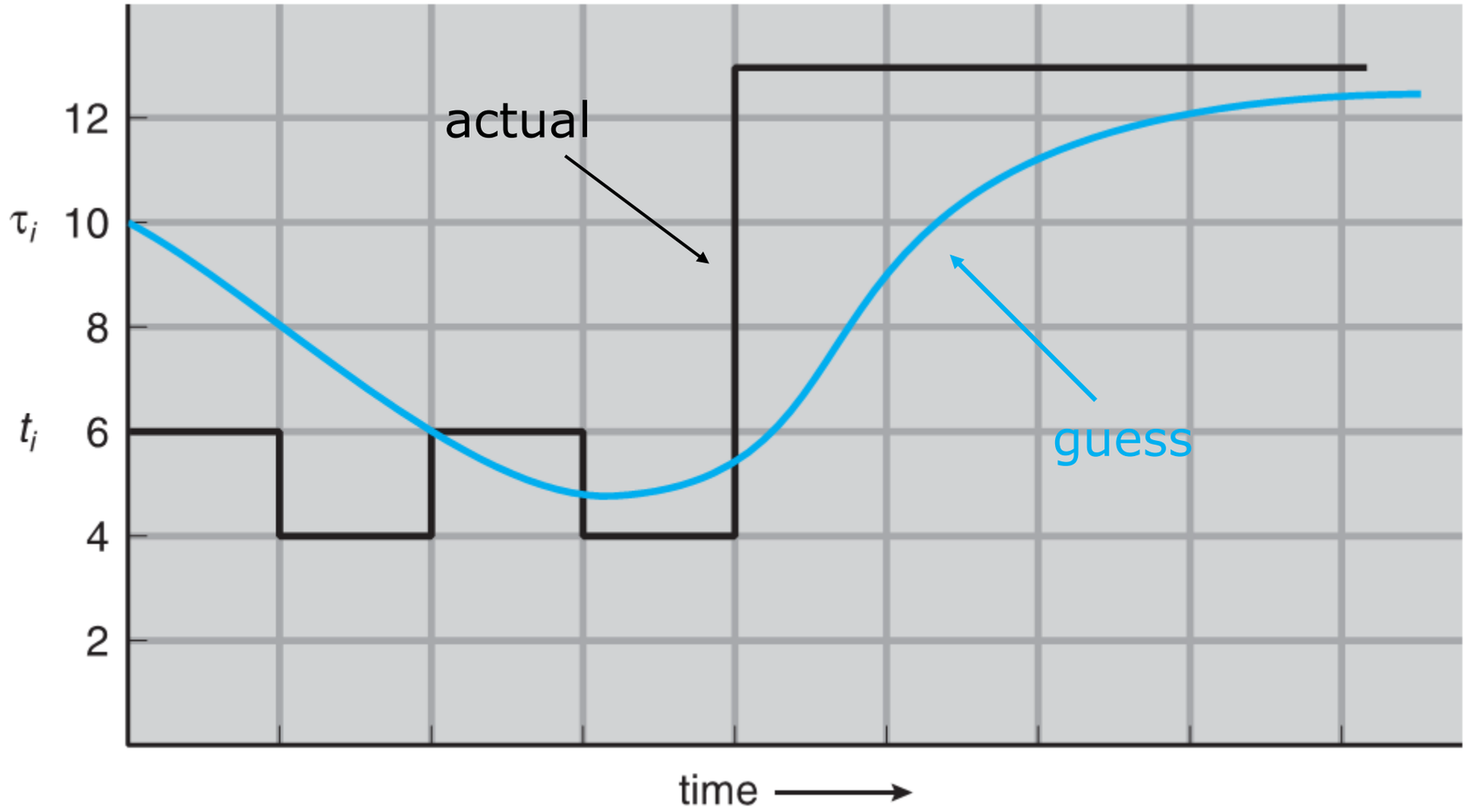
$\alpha\, t_{n-1} + (1-\alpha)\tau_{n-1}$

Expand ➔ $\tau_{n+1} = \alpha t_n + (1-\alpha)\alpha t_{n-1} + (1-\alpha)^2 \alpha t_{n-2} + \cdots + (1-\alpha)^{n+1}\tau_0$

- $\alpha = 0$ ➔ $\tau_{n+1} = \tau_n$ ➔ Recent history does not count
- $\alpha = 1$ ➔ $\tau_{n+1} = t_n$ ➔ Only the last (most recent) CPU burst counts
- Commonly, α set to ½

Shortest-job-first(SJF) can be **preemptive** or **non-preemptive**.
The preemptive version is also called **shortest-remaining-time-first.**

Prediction of the Length of the Next CPU Burst



α = 0.5

CPU burst (t_i)	6	4	6	4	13	13	13	...
"guess" (τ_i)	10							...

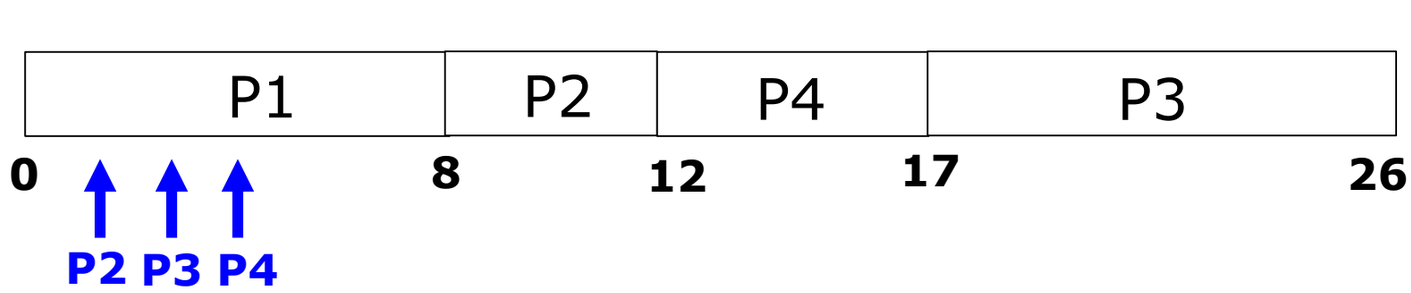
0.5x6 + 0.5x10 = 8

6 = 0.5x4 + 0.5x8

Shortest-Job-First (SJF) Scheduling

- Add the concepts of **varying arrival** to the analysis

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



- **Average waiting time** P2 P4 P3

  = [(0-0)+(8-1)+(12-3)+(17-2)]/4 = 31/4 = 7.75 msec

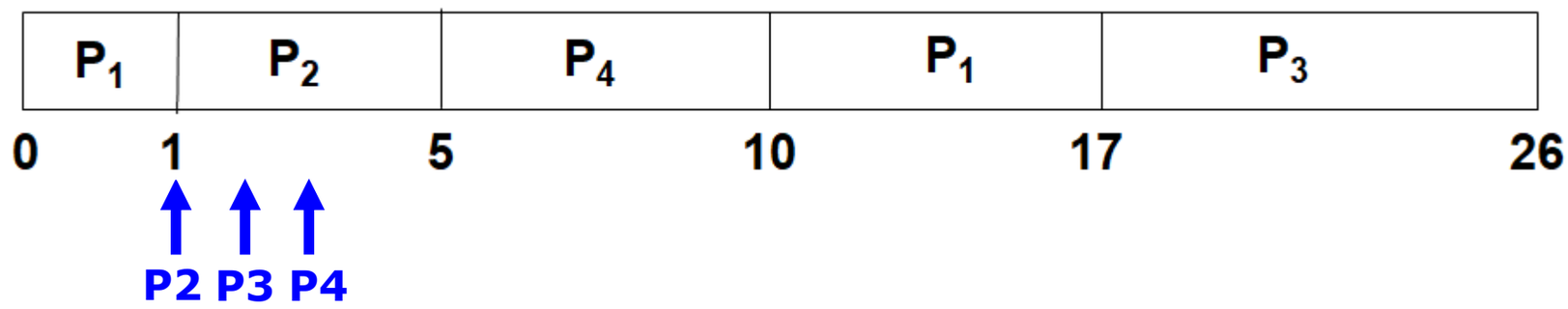
- **Can it be improved?** ➔ **Preemptive SJF**

Example of Shortest-remaining-time-first

- **Shortest-Remaining-Time-First (SRTF) : *Preemptive*** version of **SJF**
  - Currently executing process might be *preempted* by a newly arrived process
- 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

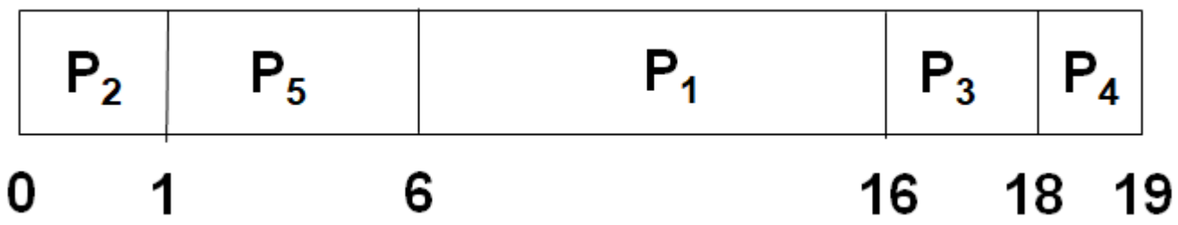
Priority Scheduling

- A priority number (integer) is associated with each process
- The CPU is allocated to the process with the highest priority (smallest integer ≡ highest priority)
  - See next slide
- Can be ***preemptive*** or ***non-preemptive***
- SJF is priority scheduling where priority is the inverse of predicted next CPU burst time
- Problem ≡ **Starvation** – low priority processes may never execute
- Solution ≡ **Aging** – as time progresses increase the priority of the process

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

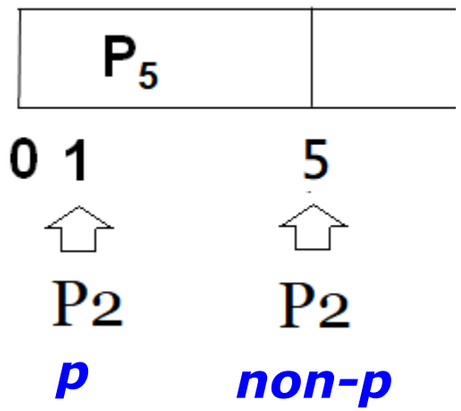


- Average waiting time = 8.2 msec

What's the difference between *preemptive* and *non-preemptive* priority scheduling?

- Consider the arrival time below

Process	Arrival
P_1	0
P_2	1
P_3	0
P_4	0
P_5	0



Round Robin (RR)

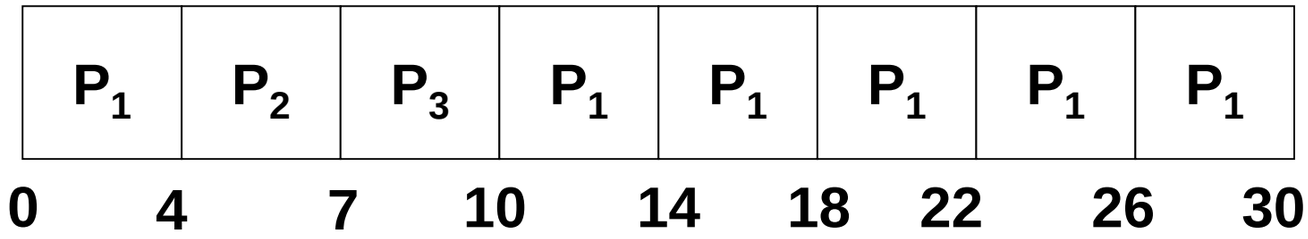
- Each process gets a small unit of CPU time (**time quantum** *q*), usually 10-100 milliseconds. After this time has elapsed, the process is preempted and added to the end of the ready queue.
- If there are *n* processes in the ready queue and the time quantum for each process is *q*, no process waits more than (*n*-1)*q* time units.
- Timer interrupts every quantum to schedule next process
- Performance
  - *q* large $\Rightarrow$ FIFO
  - *q* small $\Rightarrow$ *q* must be large with respect to context switch, otherwise overhead is too high

Example of RR with Time Quantum = 4

Process	Burst Time
P_1	24
P_2	3
P_3	3

(Time Quantum: 4)

- The Gantt chart is:

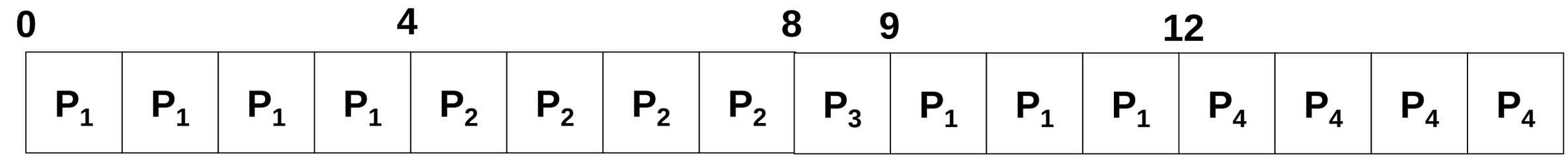


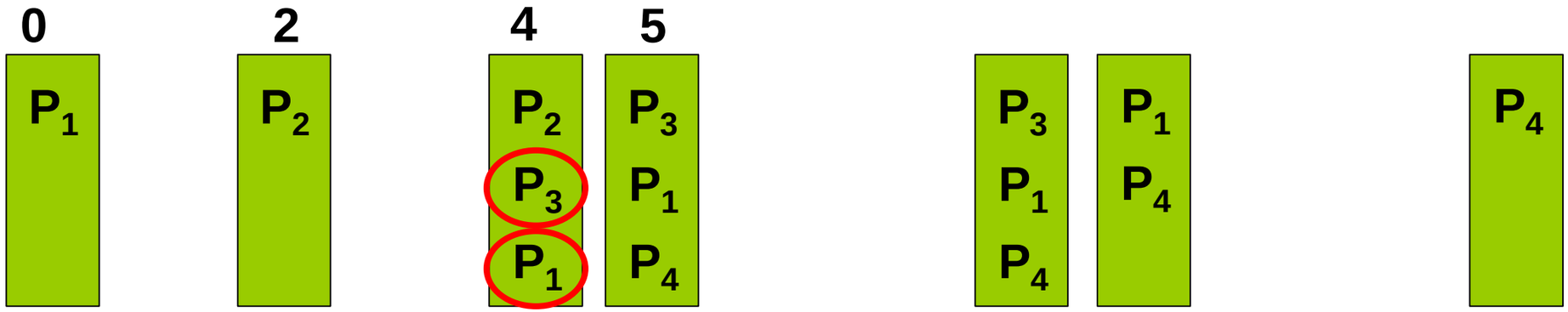
- Typically, higher average waiting time and average turnaround time than SJF, but better ***response time***
  - ***Average waiting time:*** **[(10-4)+(4-0)+(7-0)]/3**
  - ***Average response time:*** **[(0-0)+[4-0]+[7-0)] /3**
  - ***Average turnaround time*****: [(30-0) + (7-0) + (10-0)]/3**
- q should be large compared to context switch time
- q usually 10ms to 100ms, context switch < 10 usec

RR with different arrival times

Process	Arrival	CPU burst
P1	0	7
P2	2	4
P3	4	1
P4	5	4

Time quantum (time slice) = 4



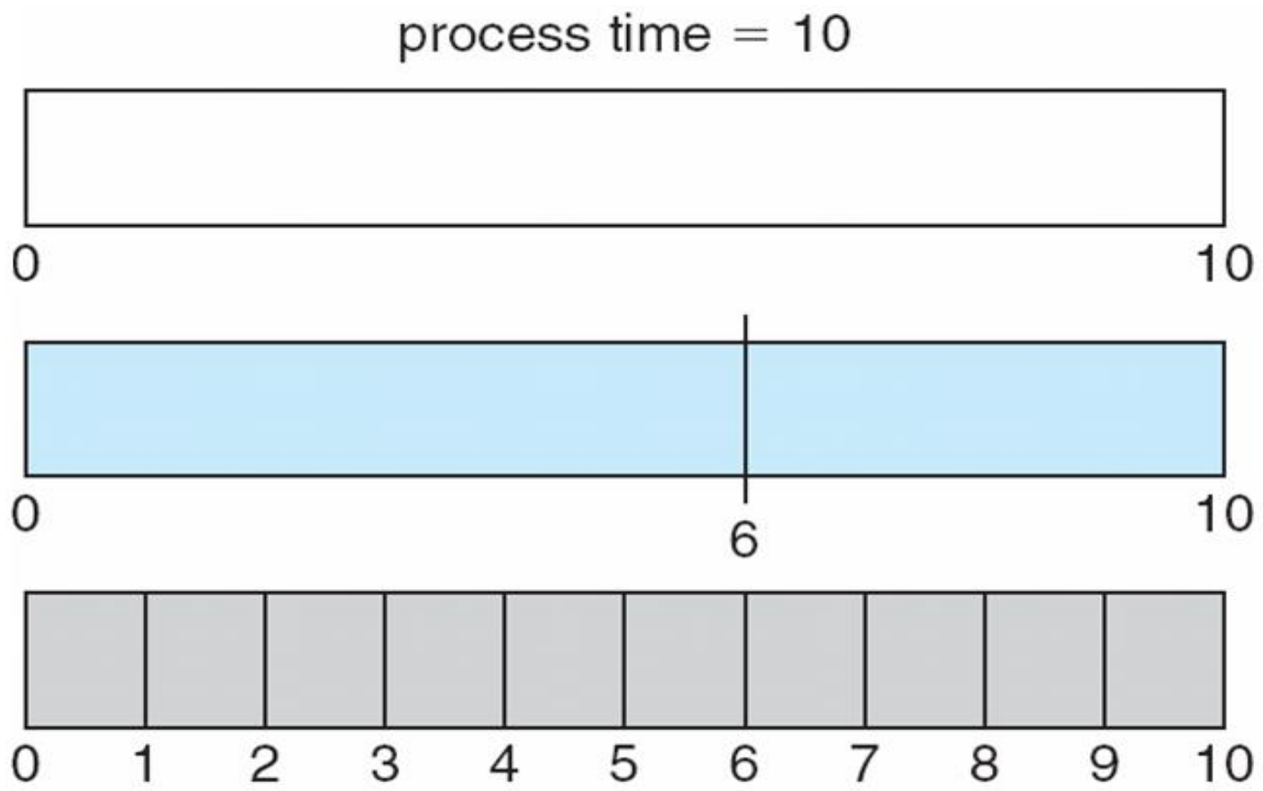


Average waiting time: [5 +(4-2)+(8-4)+(12-5)]/4 =

Average Response time : [0 +(4-2)+(8-4)+(12-5)]/4 =

Average turnaround time : [(12-0)+(8-2)+(9-4)+(16-5)]/4 =

Time Quantum and Context Switch Time



quantum	context switches
12	0
6	1
1	9

Turnaround Time Varies With The Time Quantum

- Given three processes, each requires 10 time units to finish their jobs.

- If time quantum Q = 1
  - Turn around time could be 28, 29, 30, respectively.

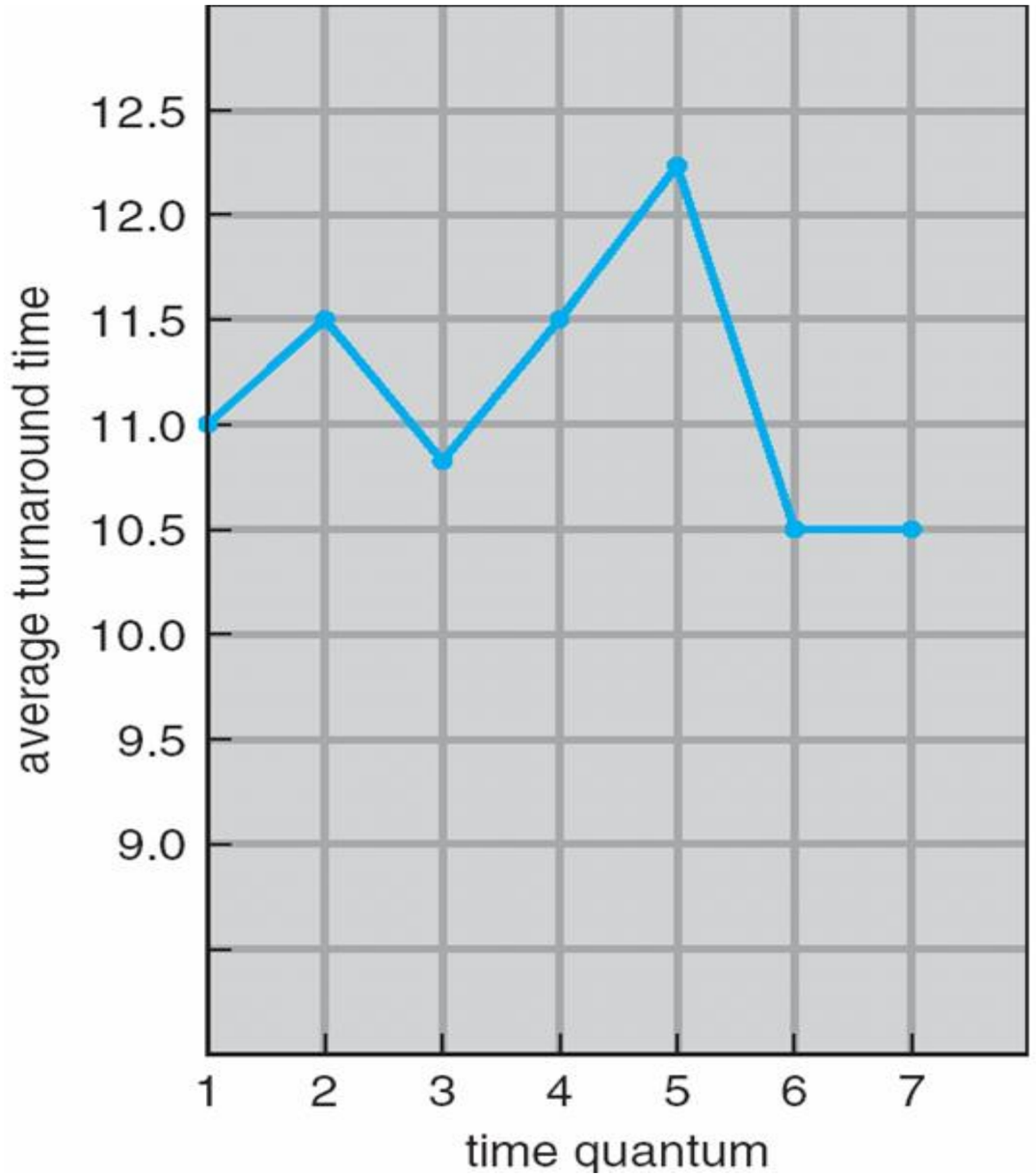
  ➔ average turnaround time = 29

- If time quantum Q = 10
  - Turn around time could be 10, 20, 30, respectively

  ➔ average turnaround time = 20

➔ Turnaround time varies with the time quantum

Turnaround Time Varies With The Time Quantum



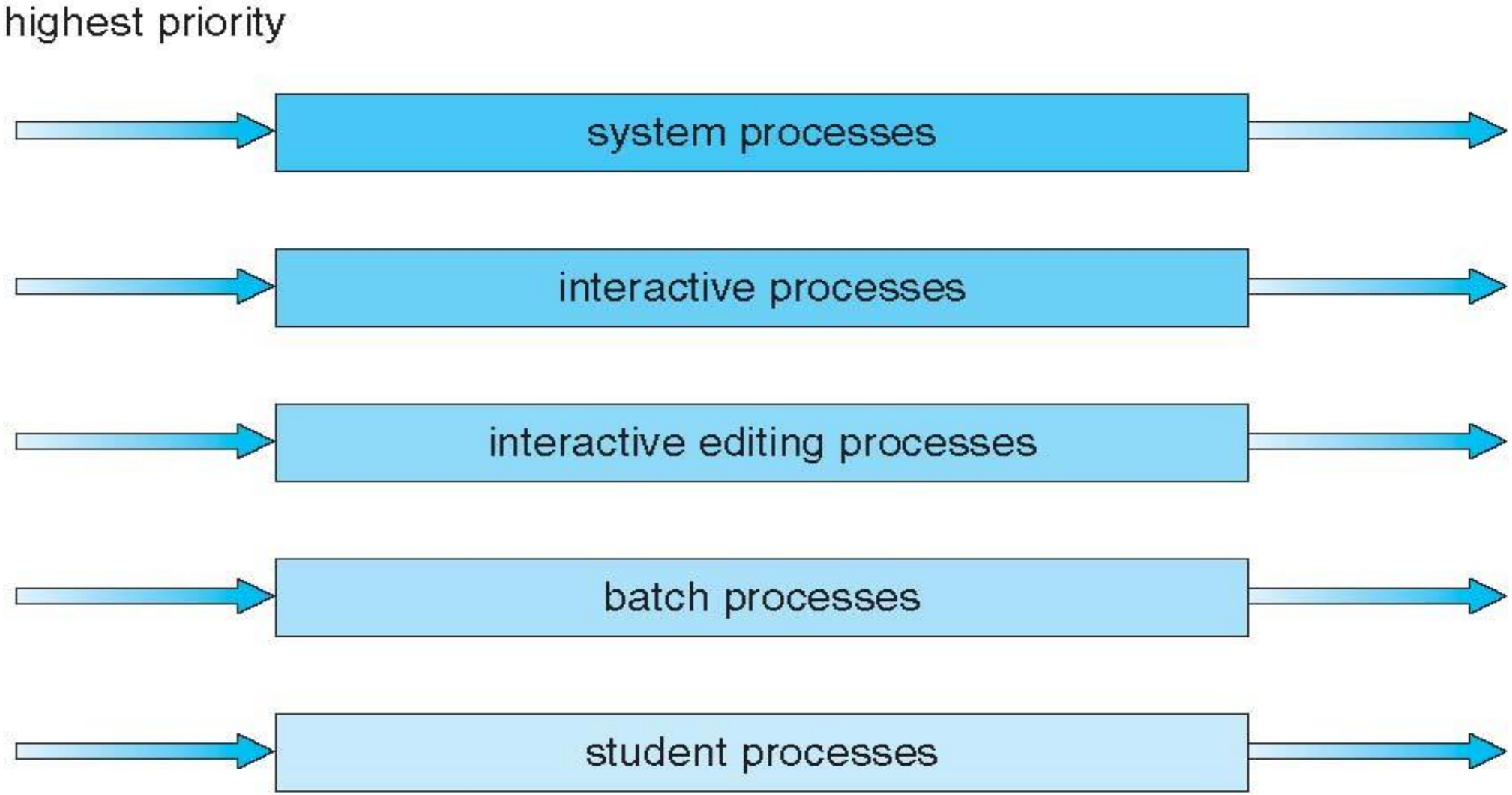
process	time
P_1	6
P_2	3
P_3	1
P_4	7

80% of CPU bursts should be shorter than q

Multilevel Queue

- Ready queue is partitioned into separate queues, eg:
  - **foreground** (interactive), **background** (batch)
- Process permanently in a given queue (i.e., no movement between different queues)
- Each queue has its own scheduling algorithm:
  - foreground – RR
  - background – FCFS
- Scheduling must be done *between the 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 among its processes; i.e., 80% to foreground in RR, 20% to background in FCFS

Multilevel Queue Scheduling

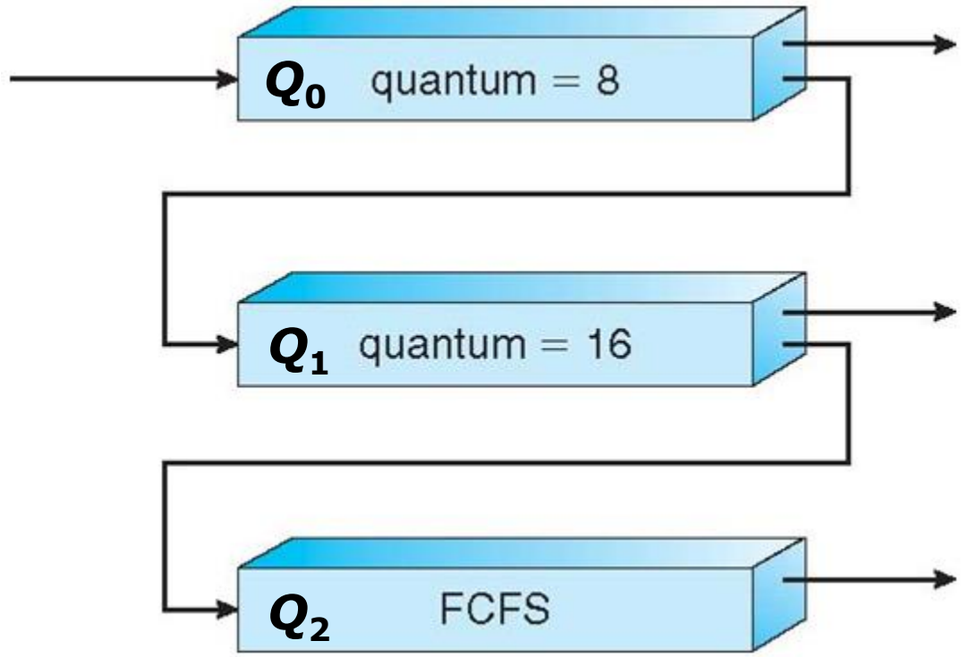


Multilevel Feedback Queue

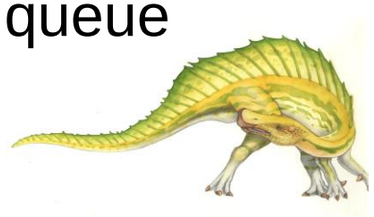
- A process **can move between the various queues**; aging can be implemented this way
- Multilevel-feedback-queue scheduler defined by the following parameters:
  - number of queues
  - scheduling algorithms for each queue
  - method used to determine when to upgrade a process
  - method used to determine 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

- e.g., **Three queues**:
  - Q_0 – RR with time quantum 8 milliseconds
  - Q_1 – RR time quantum 16 milliseconds
  - Q_2 – FCFS
- **Scheduling**
  - The jobs in Q_1 will be served only when Q_0 is empty; and the jobs in Q_2 will be served only when both Q_0 and Q_1 are empty.



- A new job enters queue Q_0 which is served *RR*
  - When it gains CPU, job receives 8 milliseconds
  - If it does not finish in 8 milliseconds, job is moved to the tail of queue Q_1
- At Q_1 job is again served RR and receives 16 additional milliseconds
  - If it still does not complete, it is preempted and moved to the tail of queue Q_2



Chapter 5: Process Scheduling

- Basic Concepts
- Scheduling Criteria
- Scheduling Algorithms
- Thread Scheduling
- Multiple-Processor Scheduling
- Real-Time CPU Scheduling
- Operating Systems Examples
- Algorithm Evaluation
- **Objectives**
  - To introduce CPU scheduling, which is the basis for multiprogrammed operating systems
  - 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

Thread Scheduling

- Distinction between user-level and kernel-level threads
- When threads supported, threads scheduled, not processes
- **process-contention scope (PCS)** – Thread library schedules user-level threads ➔ scheduling competition is within the process
- **system-contention scope (SCS)** – Kernel thread scheduled onto available CPU ➔ 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
    - PTHREAD_SCOPE_SYSTEM schedules threads using SCS scheduling
  - Linux and Mac OS X only allow PTHREAD_SCOPE_SYSTEM

Pthread Scheduling API

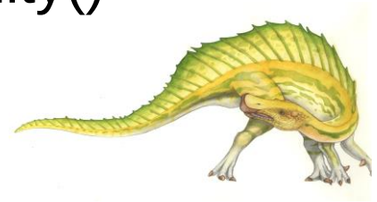
```
#include <pthread.h>
#include <stdio.h>
#define NUM THREADS 5
int main(int argc, char *argv[]) {
   int i, scope;
   pthread_t tid[NUM THREADS];
   pthread_attr_t attr;
   pthread_attr_init(&attr); /* get the default attributes */
   /* first inquire on the current scope */
   if (pthread_attr_getscope(&attr, &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

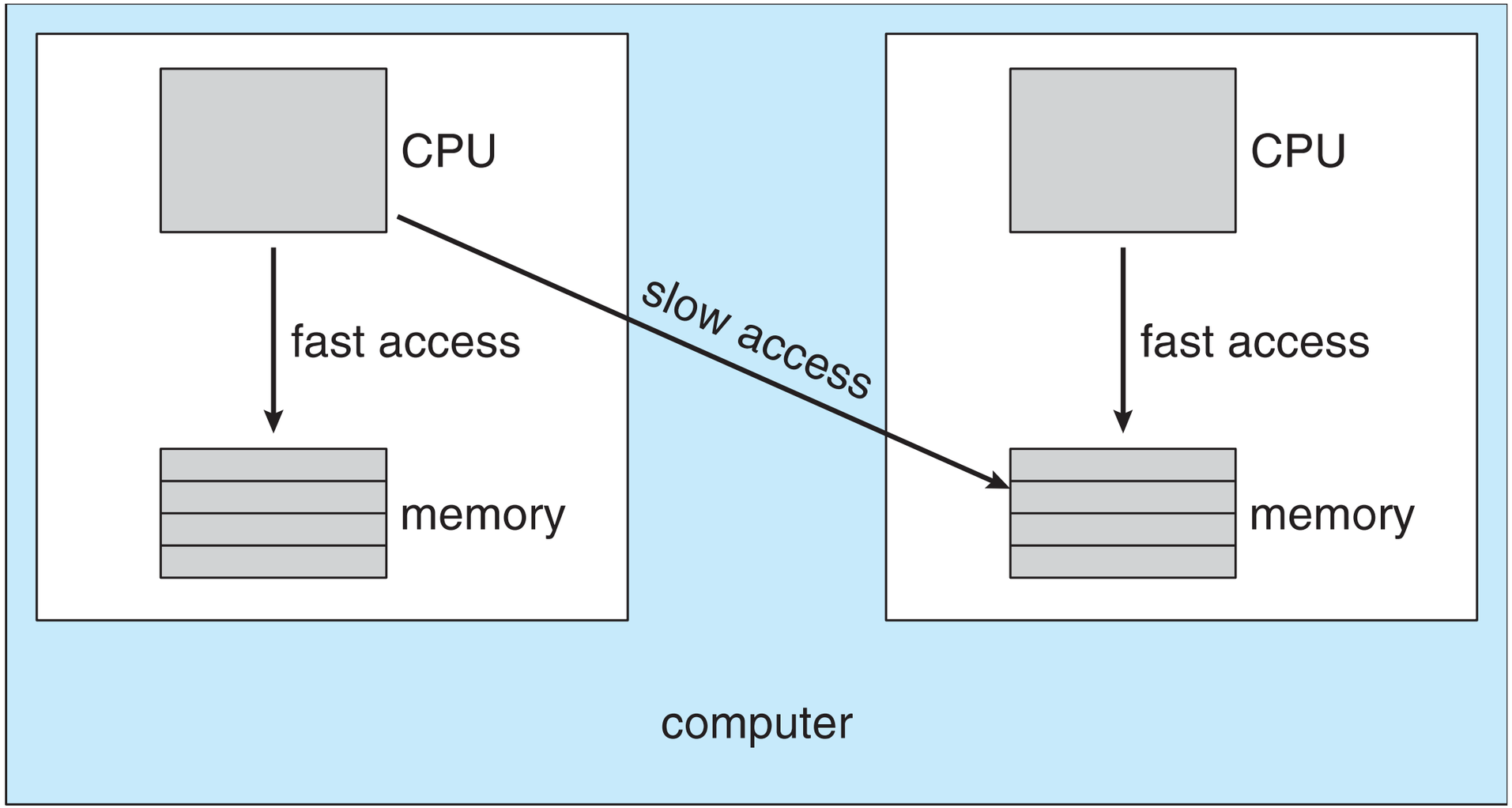
```
/* set the scheduling algorithm to PCS or SCS */
pthread_attr_setscope(&attr, PTHREAD_SCOPE_SYSTEM);
/* create the threads */
for (i = 0; i < NUM_THREADS; i++)
   pthread_create(&tid[i],&attr,runner,NULL);
/* now join on each thread */
for (i = 0; i < NUM_THREADS; i++)
   pthread_join(tid[i], NULL);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

Multiple-Processor Scheduling

- **Asymmetric multiprocessing** – only one processor accesses the system data, alleviating the need for data sharing
- **Symmetric multiprocessing (SMP)** – each processor is self-scheduling (i.e., each can access system data). Two possible ways for queues:
  1. each has its own private queue of ready processes
  2. all processes in common ready queue
- **Processor affinity** – process has affinity for processor on which it is currently running. Most SMP try to avoid migration of processes from one processor to another.
  - **soft affinity:** OS has a policy of attempting to keep a process running on the same processor – but not guarantee
  - **hard affinity:** systems provide system calls that allows a process to specify a subset of processors on which it can run.
  - Linux implements soft affinity but it also provides the sched_setaffinity() system call, which support hard affinity.



NUMA and CPU Scheduling



NUMA: Non-Uniform Memory Access

Note: Memory-placement algorithms can also consider affinity

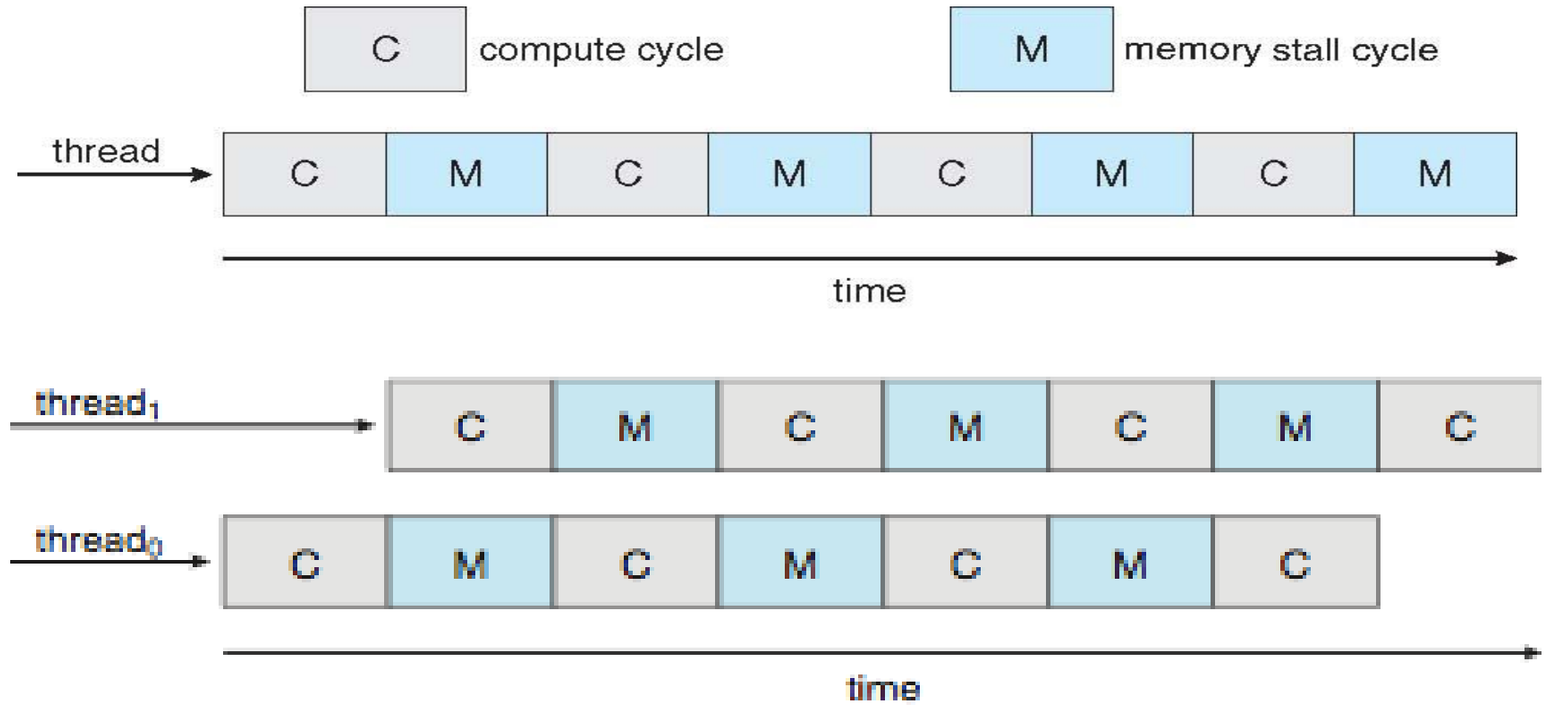
Multiple-Processor Scheduling – Load Balancing

- Keep all CPUs loaded in SMP
  - On Systems with a common ready queue, load balancing is unnecessary.
  - However, in most contemporary OS supporting SMP, each processor does have a private queue of eligible processes.
- **Load balancing** attempts to keep workload evenly distributed
  - **Push migration** – a specific task periodically checks load on each processor, and - if found an imbalance - pushes task from overloaded CPU to other CPUs
  - **Pull migration** – idle processors pulls waiting task from busy processor
- Some systems implement both push and pull migration techniques. For example, the Linux scheduler and ULE scheduler available for FreeBSD.
- Note: load balancing often counteracts the benefits of processor affinity.

Multicore Processors

- place multiple processor cores on same physical chip ➔ multicore processor
  - Faster and consumes less power
- Multiple threads per core also growing ➔ multithreaded multicore system
  - Takes advantage of memory stall to make progress on another thread while memory retrieve happens
- From OS perspective, each hardware thread appears as a logical processor
  - On a dual-threaded, dual-core system, four logical processors are presented to the OS.

dual threaded processor

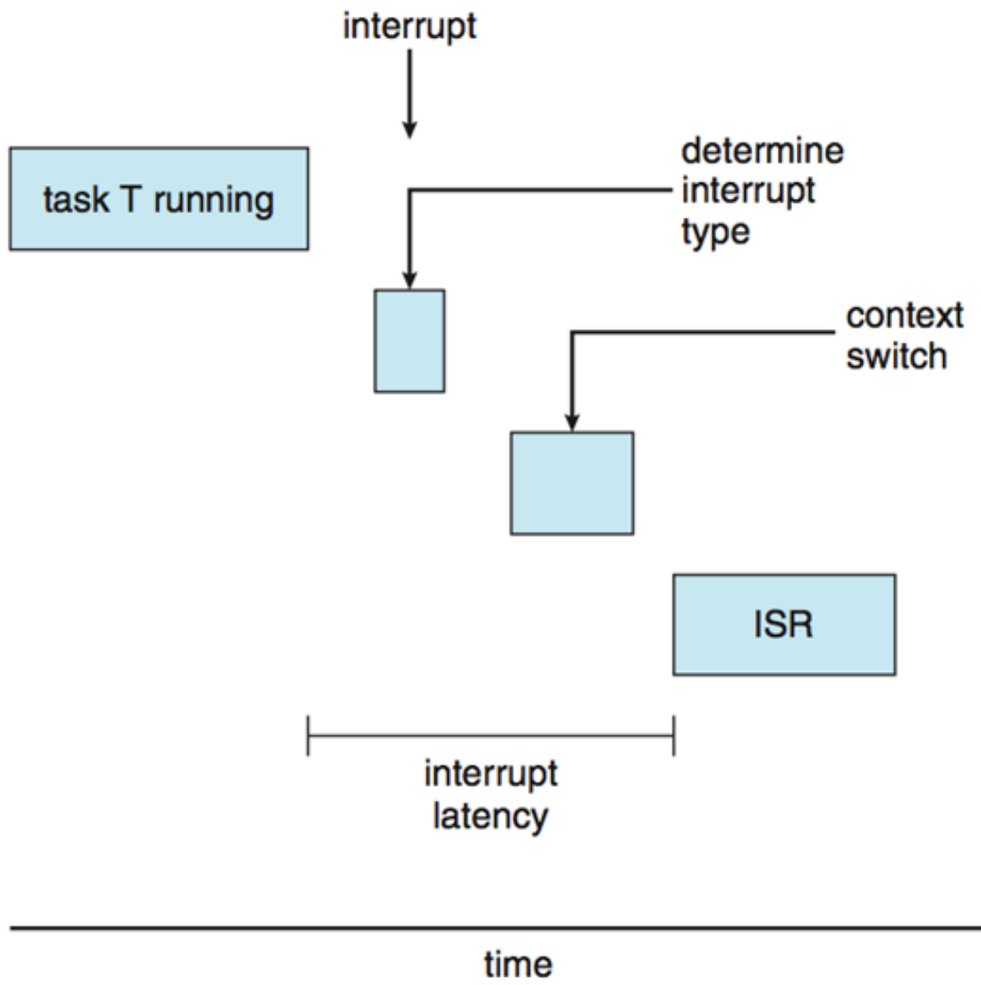


Chapter 5: Process Scheduling

- Basic Concepts
- Scheduling Criteria
- Scheduling Algorithms
- Thread Scheduling
- Multiple-Processor Scheduling
- Real-Time CPU Scheduling
- Operating Systems Examples
- Algorithm Evaluation
- **Objectives**
  - To introduce CPU scheduling, which is the basis for multiprogrammed operating systems
  - 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

Real-Time CPU Scheduling

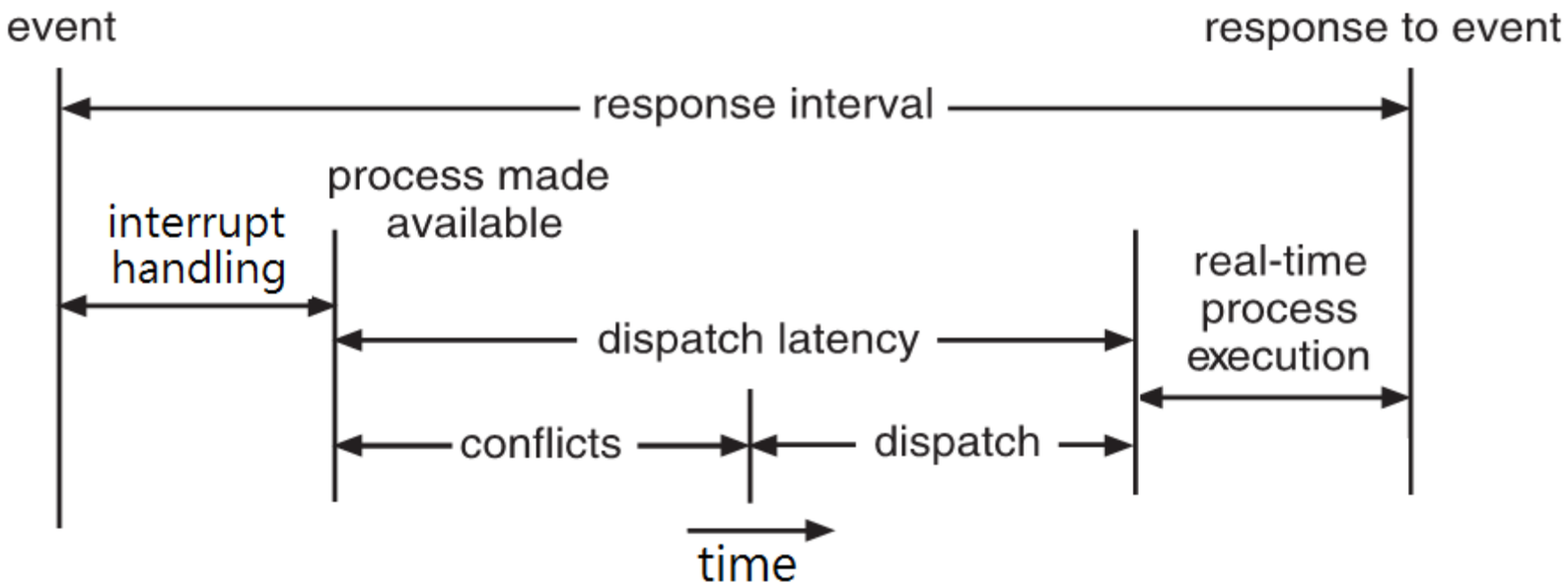
- Can present obvious challenges
- **Soft real-time systems** – no guarantee as to when critical real-time process will be scheduled
- **Hard real-time systems** – task must be serviced by its deadline
- Two types of latencies affect performance of real-time systems
  1. Interrupt latency – time from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

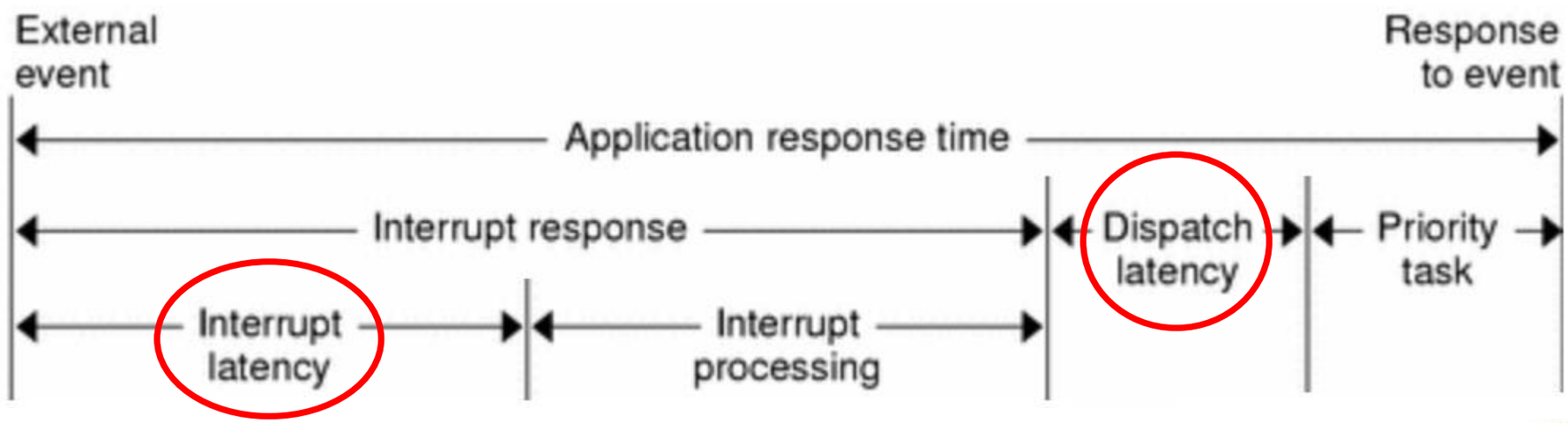


Real-Time CPU Scheduling (Cont.)

Dispatch latency

- Conflict phase
  1. Preemption of any process running in kernel mode
  2. Low-priority process releases resources needed by high-priority processes
- Dispatch phase
  1. Restore the context for the new process to run





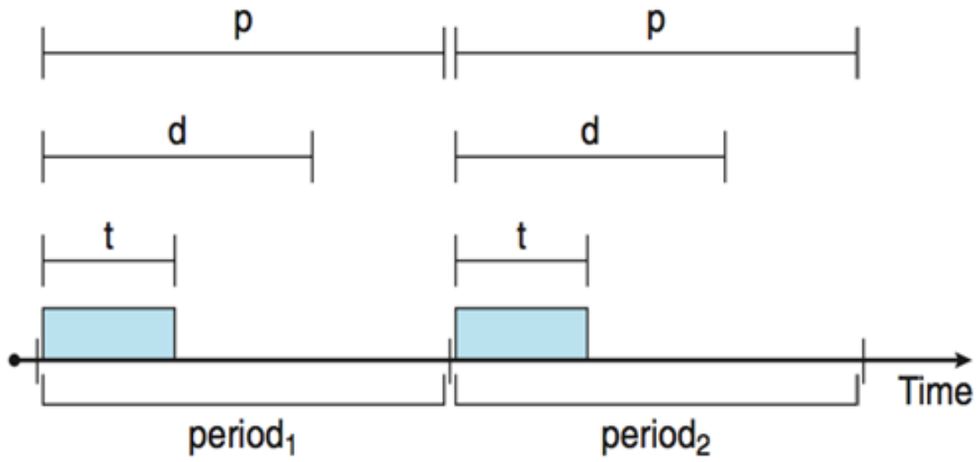
Real-time CPU Scheduling: Priority-based

- For real-time scheduling, scheduler must support preemptive, priority-based scheduling
  - Some systems assigns real-time processes the highest scheduling priority.
    - For example, windows has 32 different priority levels. The highest levels – priority values 16 to 31 – are reserved for real-time processes.
  - But only guarantees soft real-time

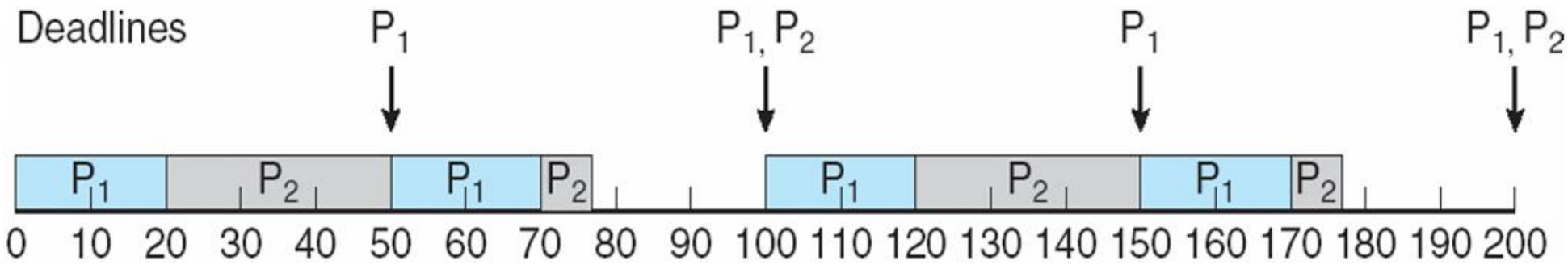
- For hard real-time must also provide ability to meet deadlines
  - Appropriate scheduling algorithms
    - **Rate-monotonic scheduling (RMS)**
    - **Earliest-deadline-first scheduling (EDF)**

Rate Montonic Scheduling

- **Periodic process**: the one requires CPU at constant intervals
  - processing time *t*, deadline *d,* period *p, where* $0 \le t \le d \le p$
- RMS requires that processes be periodic and constant CPU time per burst.
- A priority is assigned based on its period
  - Shorter periods = higher priority;
- Example: two processes:
  - P_1 : period *p* = **50**, processing time *t* = **20**
  - P_2 : period *p* = **100**, processing time *t* = **35**



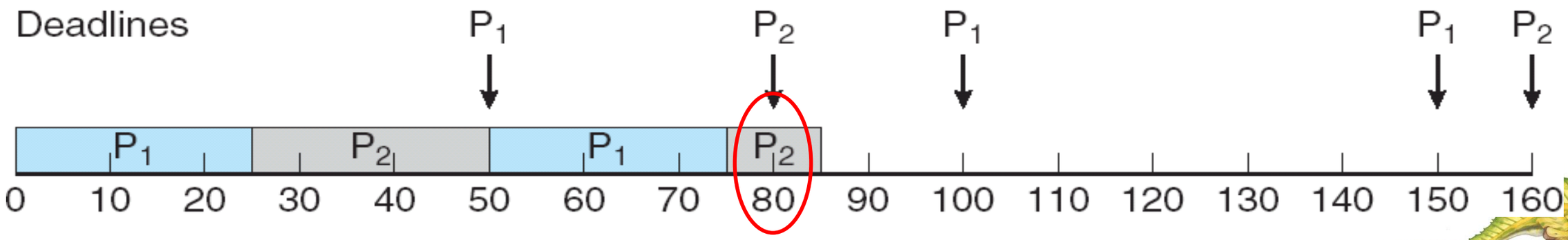
➔ P_1 is assigned a higher priority than P_2 (because 50<100)

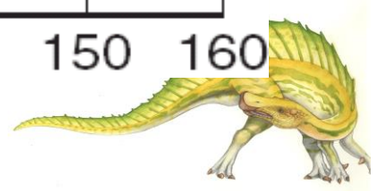


Missed Deadlines with Rate Monotonic Scheduling

- RMS is considered optimal in that if a set of processes cannot be scheduled by this algorithm, it cannot be scheduled by any other algorithm that uses **static priorities**.
- The case that cannot be scheduled by RMS
  - P_1 : period p = **50**, processing time t = **25**
  - P_2 : period p = **80**, processing time t = **35**
  - ➔ P_1 is assigned a higher priority than P_2 (because 50<80)
- P_2 will miss its deadline although total CPU utilization is (25/50)+(35/80)=0.94 < 1
  - The worst-case CPU utilization for scheduling N processes is: $N(2^{1/N} - 1)$
  - 1 for N=1, 0.828 for N=2, ~69% for N= infinity

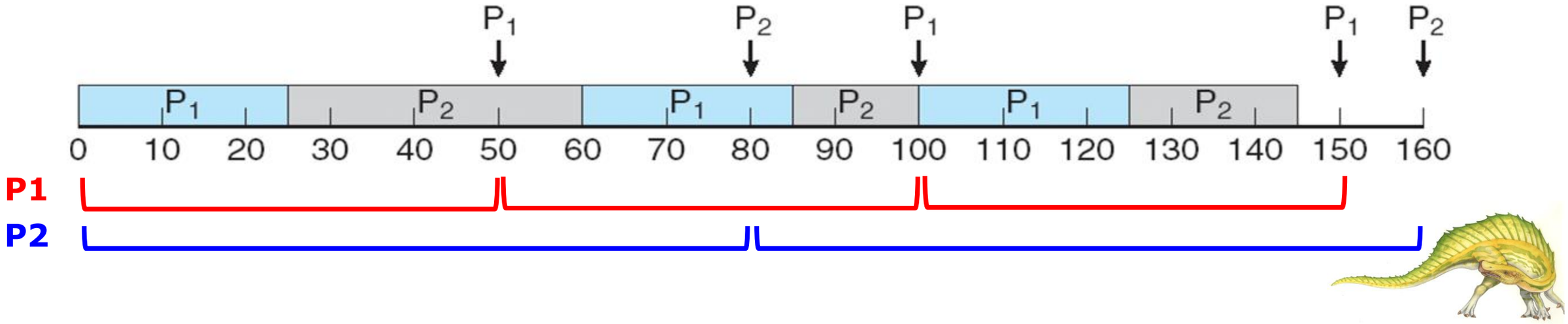
A feasible scheduling exists if CPU utilization is below the bound





Earliest Deadline First Scheduling (EDF)

- Priorities are assigned according to deadlines:
  - the earlier the deadline, the higher the priority;
  - the later the deadline, the lower the priority
- The case that cannot be scheduled by RMS can be scheduled by EDF
  - P_1 : period p = **50**, processing time t = **25**
  - P_2 : period p = **80**, processing time t = **35**
  - EDF allows P_2 to continue running at time 50 because P_2 has next deadline (at time 80) earlier than P_1 (at time 100). Thus, both meet their first deadline.
- EDF does not require the processes be periodic, nor constant CPU time per burst
- The **priority** in EDF is ***dynamic***, while in RMS it is ***static***.



Proportional Share Scheduling

- *T* shares are allocated among all processes in the system
- An application receives *N* shares where *N* < *T*
- This ensures each application will receive ***N*** / *T* of the total processor time
  - Assume a total of T=100 shares is for processes A, B, C. A is assigned 50 shares, B is 15, and C is 20.
  - This scheduling ensures A will have 50 percent of total CPU time, and B will have 15 percent, and C will have 20 percent.
- This scheduling must work with an admission-control policy which will admit a process requesting a particular number of shares only if sufficient shares are available.

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;
    pthread_attr_init( &attr );   /* get the default attributes */
/* 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);
}
```

Chapter 5: Process Scheduling

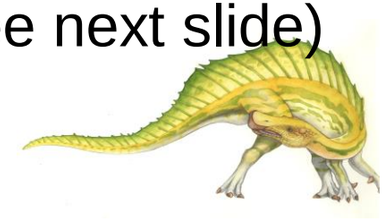
- Basic Concepts
- Scheduling Criteria
- Scheduling Algorithms
- Thread Scheduling
- Multiple-Processor Scheduling
- Real-Time CPU Scheduling
- Operating Systems Examples
  - Linux scheduling
  - Windows scheduling
  - Solaris scheduling
- Algorithm Evaluation

Linux Scheduling Through Version 2.5

- Prior to kernel version 2.5, ran variation of standard UNIX scheduling algorithm

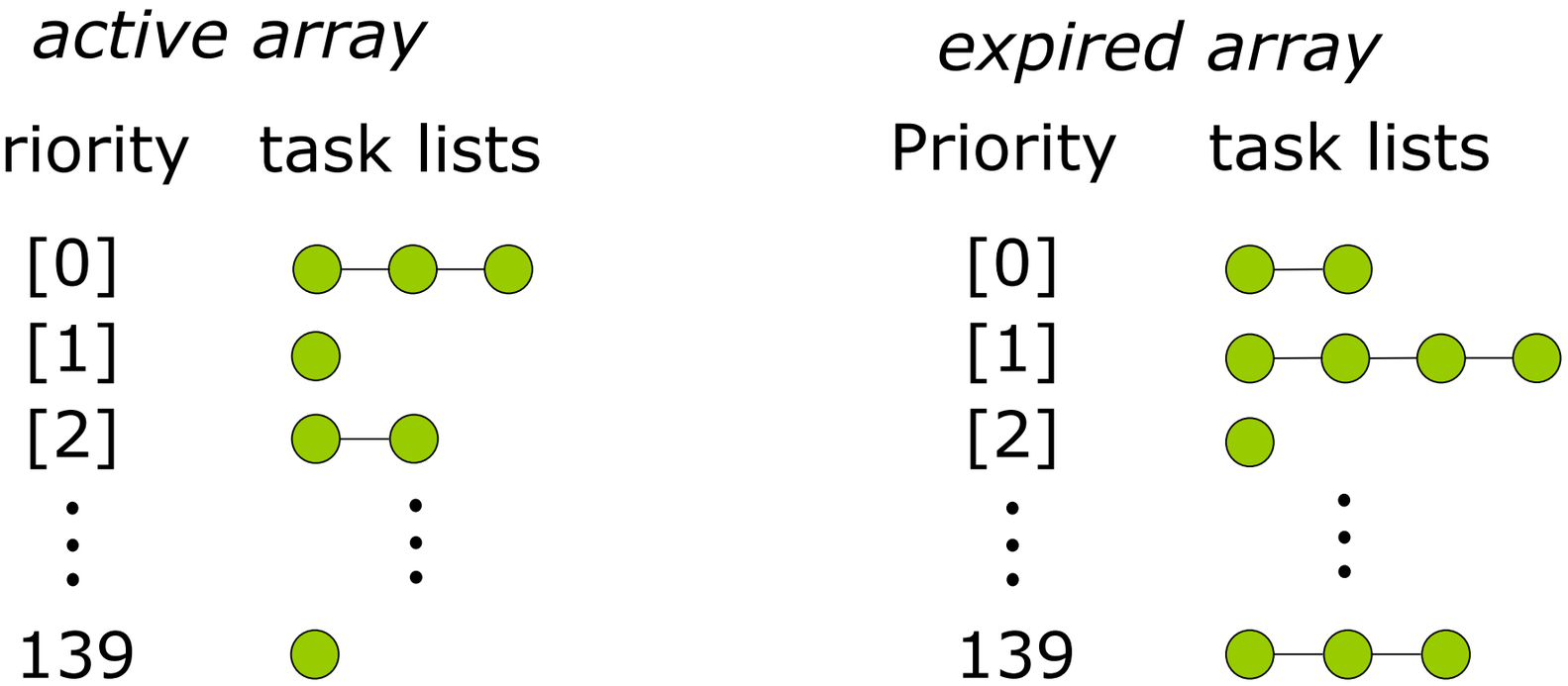
O(1) Scheduler

- Version 2.5 moved to constant order *O*(1) scheduling time ➔ **O(1) scheduler**
  - Preemptive, priority based
  - Two priority ranges: **real-time** (0 – 99) and **time-sharing** (100 – 140)
  - Real-time tasks have **static** priority
  - Time-sharing tasks have **dynamic** priority depending on their interactively
    - Priority is recalculated when time slice expired
    - According to its past behavior, the task priority may be increased or decreased
  - Maintains two priority arrays for each processor: **active** and **expired**
    - The active array contains tasks that have timeslice left (**active**)
    - The expired array contains tasks that have exhausted their timeslice (**expired**)
    - In each array, lists of tasks are indexed according to priorities (see next slide)



Linux Scheduling Through Version 2.5

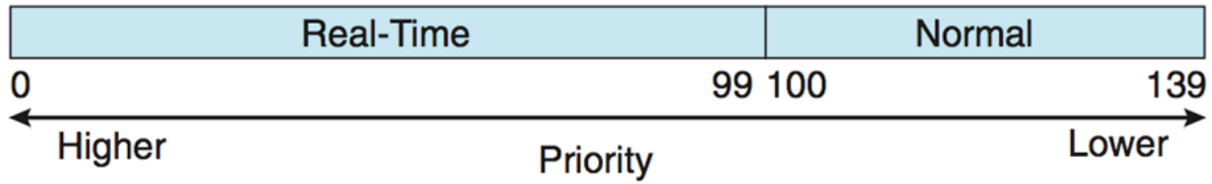
- **O(1) scheduler**



- Earlier Linux versions put all runnable processes (active) in the same list. The scheduler scan the whole list to select the “best” runnable process.
- The trick used in O(1) to achieve the speedup is to split the runqueue in many lists of processes, one list per process priority (140 lists)

Linux Scheduling in Version 2.6.23 +

Completely Fair Scheduler (CFS)

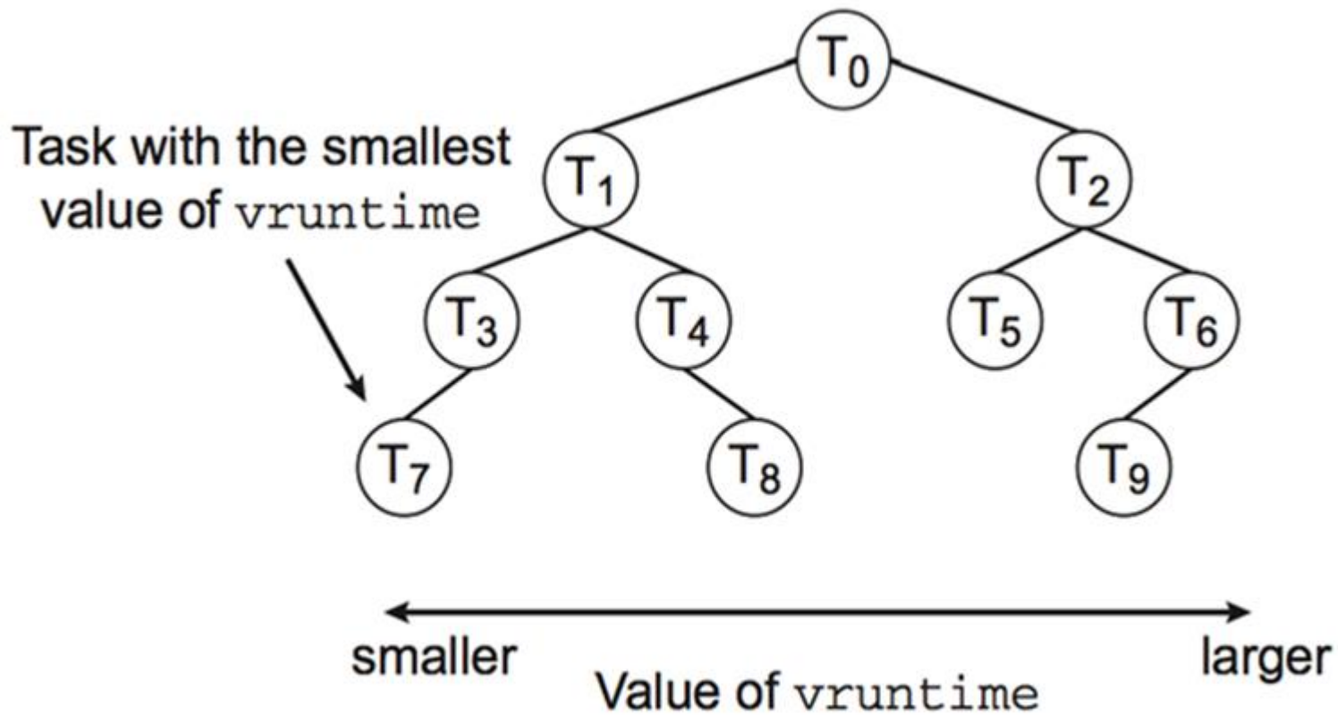


- **Scheduling classes**
  - Scheduler picks highest priority task in highest scheduling class
  - 2 scheduling classes included: 1. real-time (0-99), 2. normal (100-139)
  - Rather than based on fixed time, the quantum is based on proportion of CPU time
    - Quantum calculated based on **nice value** from -20 to +19
    - Lower nice value is higher priority
    - Calculate **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`**
  - If a lower-priority task (nice>0) runs for 200ms, its vruntime will be > 200ms.
  - If a higher-priority task (nice<0) runs for 200ms, its vruntime will be < 200ms.
  - Normal priority (nice value=0) yields virtual run time = actual run time
- To decide next task to run, scheduler picks task with **lowest** virtual run time

CFS Performance

- Red-black tree: used by the CFS scheduler for efficiently selecting which task to run next.
  - a balanced binary search tree based on the value of vruntime.
  - When a task becomes runnable, it is added to the tree.
  - Non-runnable tasks will be removed from the tree.
  - Tasks with smaller vruntime are toward the left side of the tree.

    ➔ leftmost node has the highest priority.
  - Navigating the tree to discover the leftmost node will require O(lgN) operations.



Windows Scheduling

- Windows uses priority-based preemptive scheduling
- Highest-priority thread runs next
- **Dispatcher** is scheduler
- Thread runs until (1) blocks, (2) uses up 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 six priority classes
  - REALTIME_PRIORITY_CLASS,
  - HIGH_PRIORITY_CLASS,
  - ABOVE_NORMAL_PRIORITY_CLASS,
  - NORMAL_PRIORITY_CLASS,
  - BELOW_NORMAL_PRIORITY_CLASS,
  - IDLE_PRIORITY_CLASS
- 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 (*see next slide*)
- Base priority is NORMAL within the class
- If quantum expires, priority lowered, but never below base
- 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

Windows Priorities

Priority class and relative priority combine to give numeric priority

	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 (TS) is multi-level feedback queue
  - It dynamically alters priorities and assigns time slices of different lengths
    - The higher the priority, the smaller the time slice
  - Loadable table configurable by sysadmin

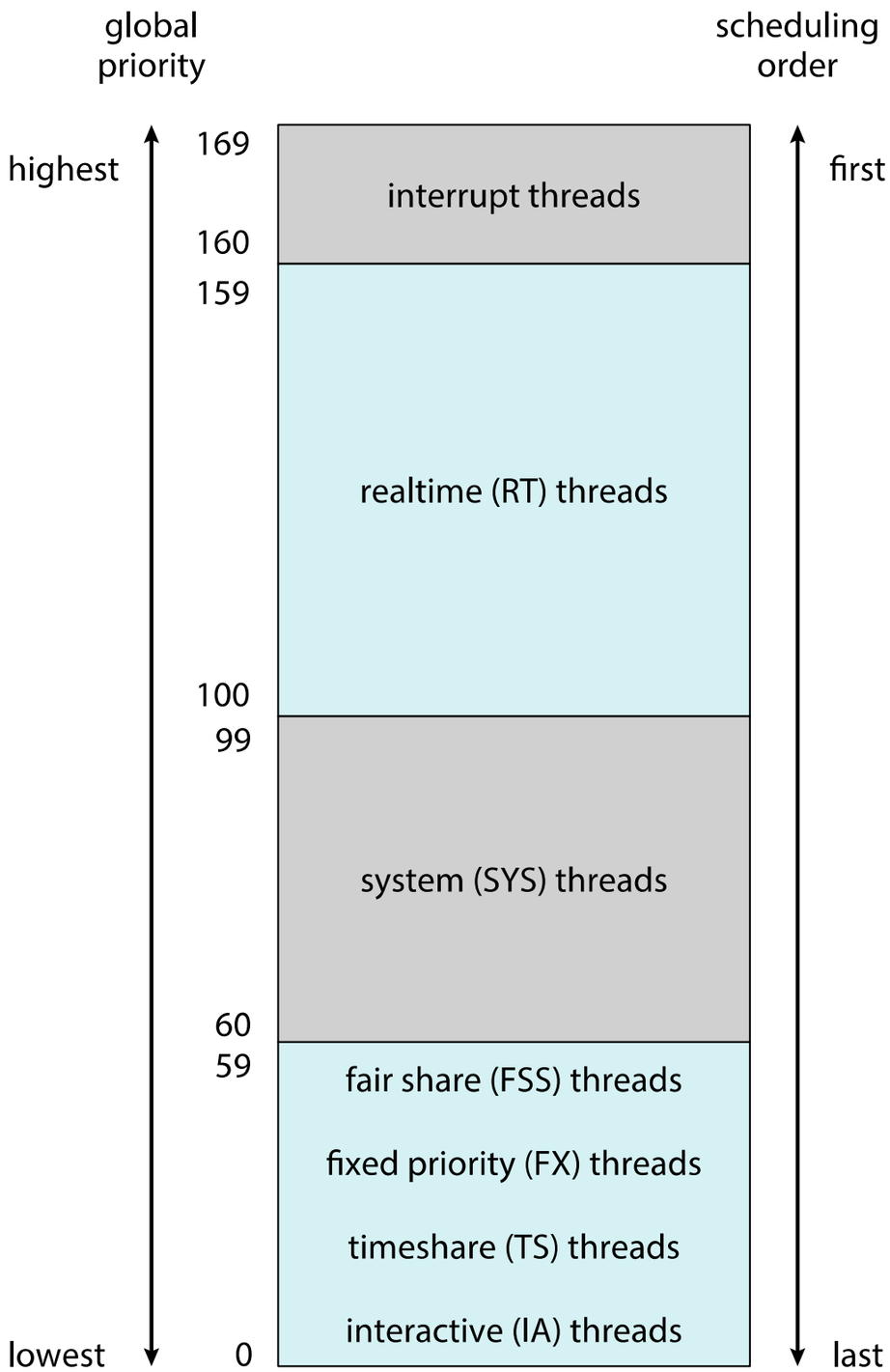
Solaris Dispatch Table

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

low ↓ ***high***

- **Priority**: the class-dependent priority for time-sharing and *interactive* classes. A higher number indicates a higher priority.
- **Time quantum**: time quantum for the associated priority.
- **Time quantum expired**: The new priority of a thread that has used its entire time quantum without blocking.
- **Return from sleep**: The priority of a thread that is returning from sleeping (such as from waiting for I/O).

Solaris Scheduling



- System class is used to run Kernel threads, such as scheduler and paging daemon.
- The fixed-priority and fair-share classes were introduced with Solaris 9.
- Scheduler converts class-specific priorities into a per-thread global priority
  - Thread with highest priority runs next
  - Runs until (1) blocks, (2) uses up time slice, (3) preempted by higher-priority thread
  - Multiple threads at same priority selected via RR

Chapter 5: Process Scheduling

- Basic Concepts
- Scheduling Criteria
- Scheduling Algorithms
- Thread Scheduling
- Multiple-Processor Scheduling
- Real-Time CPU Scheduling
- Operating Systems Examples
- Algorithm Evaluation
- **Objectives**
  - To introduce CPU scheduling, which is the basis for multiprogrammed operating systems
  - 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

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:

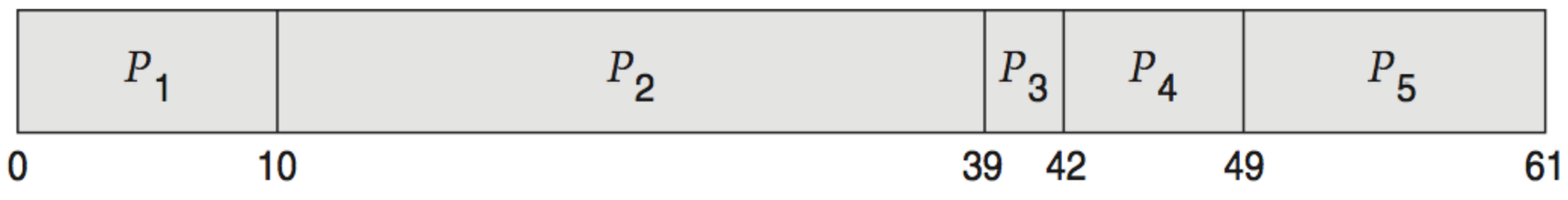
Process	Burst Time
P_1	10
P_2	29
P_3	3
P_4	7
P_5	12

Deterministic Evaluation

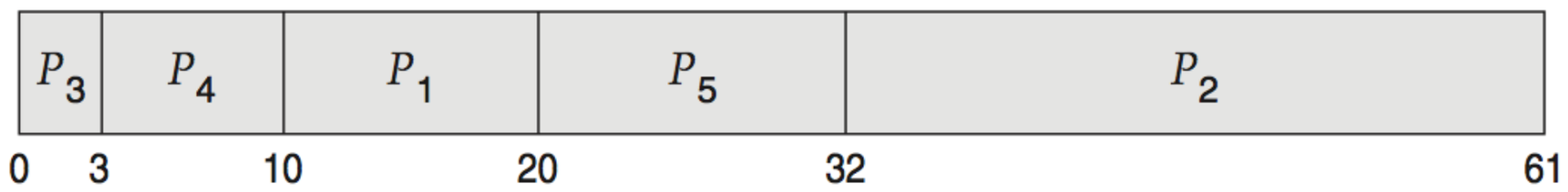
- For each algorithm, calculate minimum average waiting time

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

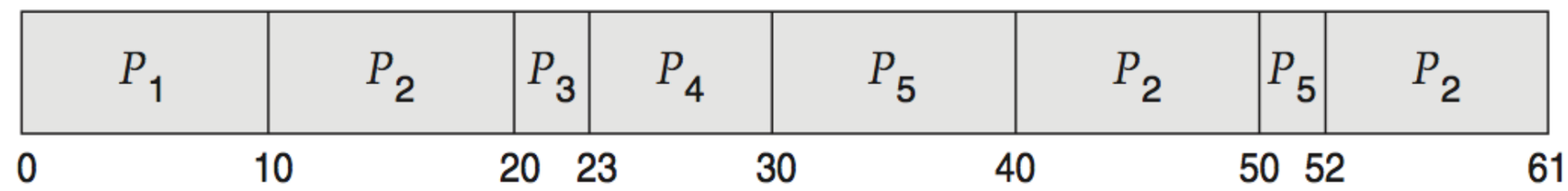
  - FCFS is 28ms:



  - Non-preemptive SFJ is 13ms:



  - RR is 23ms: (time quantum = 10)



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

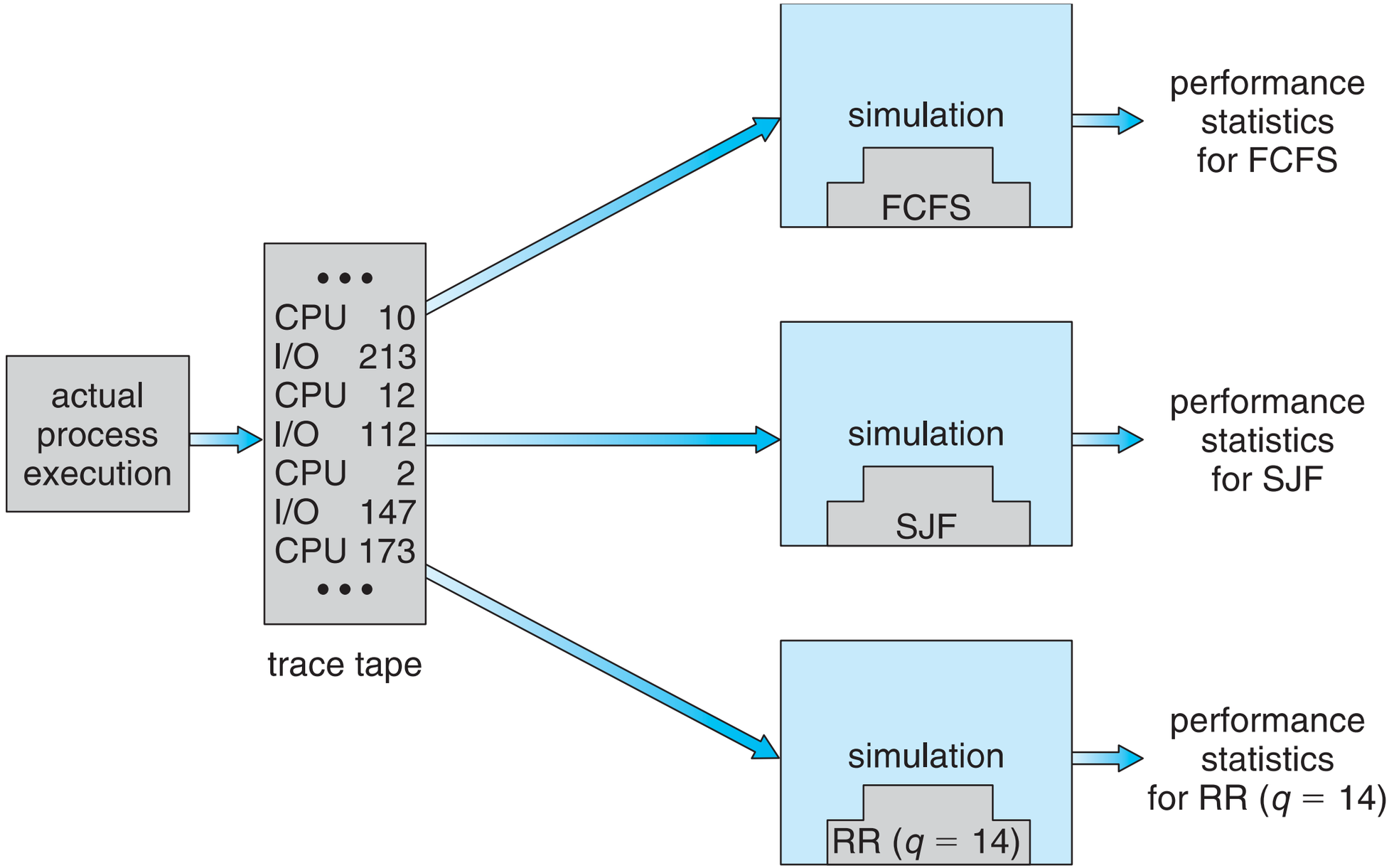
- n = average queue length
- W = average waiting time in queue
- λ = average arrival rate into queue
- Little's law – in steady state, processes leaving queue must equal processes arriving, thus $n = \lambda \times W$
  - Valid for any scheduling algorithm and arrival distribution

- 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
  - The classes of algorithm and distributions that can be handled are fairly 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