PART 2. PROCESS MANAGEMENT

Chapter 5: Process Scheduling

WHAT'S AHEAD:

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

WE AIM:

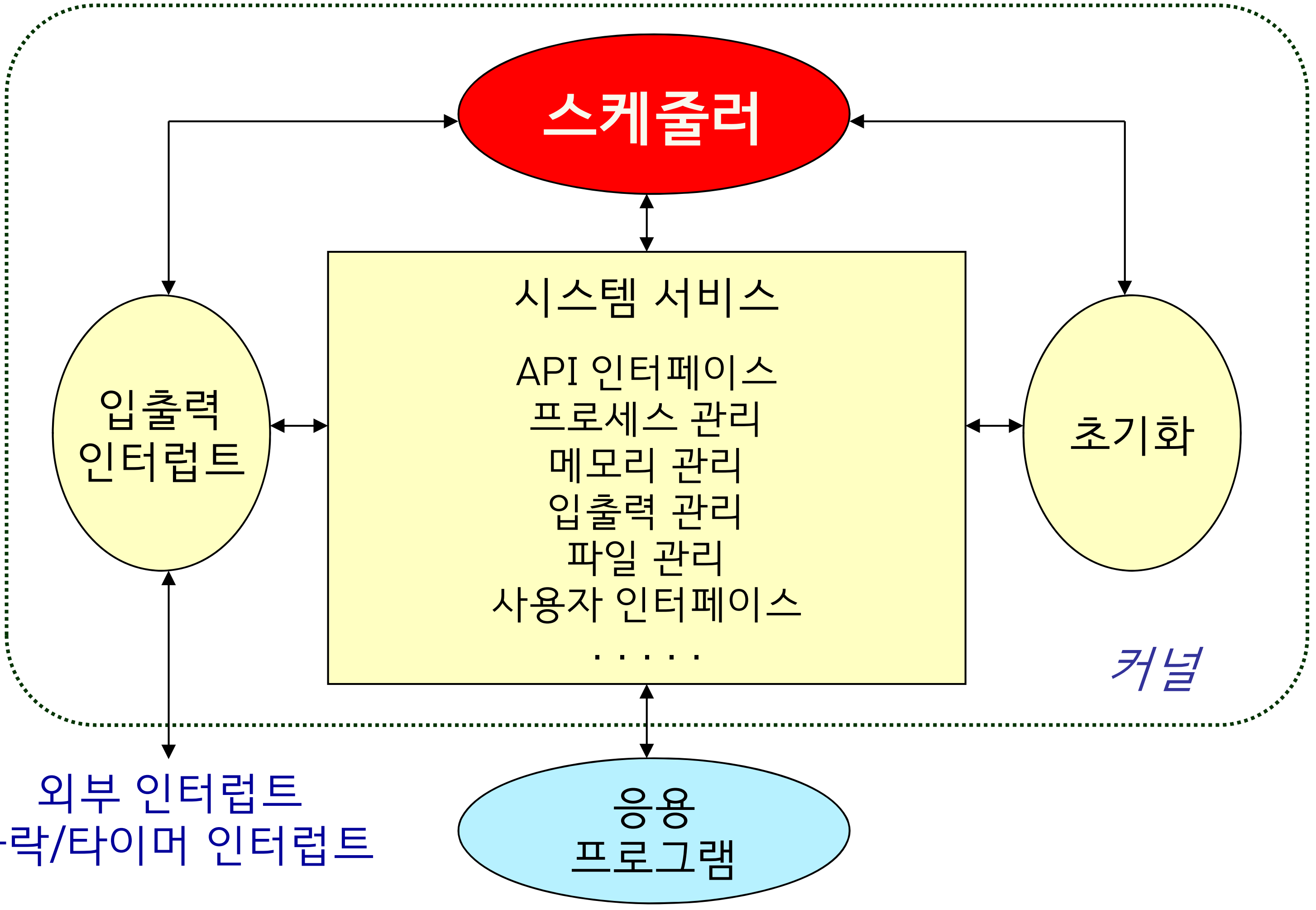
- To introduce CPU scheduling, for multi-programmed OSs
- To describe and evaluate various CPU-scheduling algorithms
- To examine the scheduling algorithms of several OSs

Note: These lecture materials are based on the lecture notes prepared by the authors of the book titled *Operating System Concepts*, 9e (Wiley)

핵심요점

모든 길은 스케줄러로 통한다

운영체제의 동적인 측면 –
스케줄러가 시스템을 움직인다

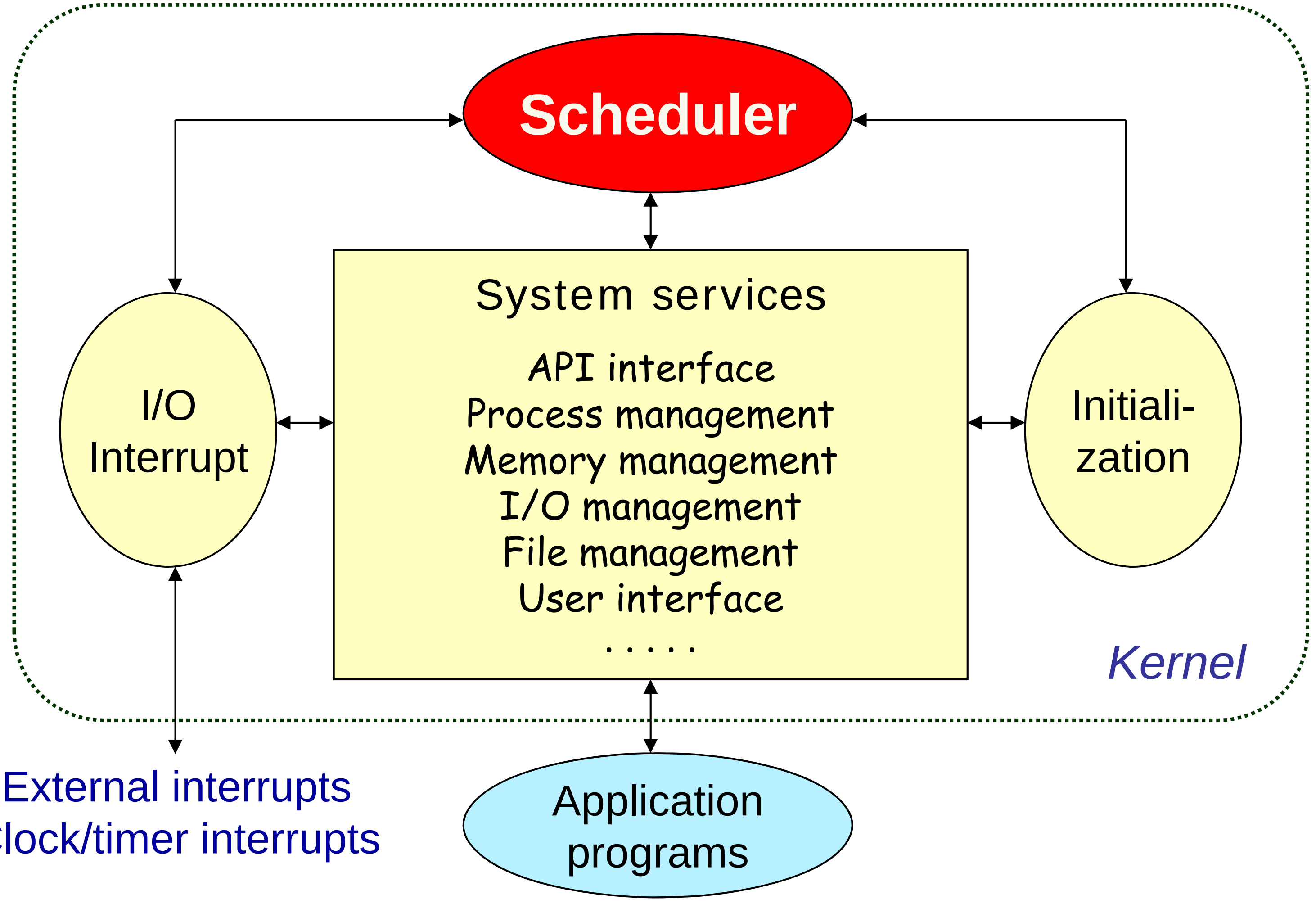


Core Ideas

All Roads Lead to Scheduler

A dynamic view of operating systems –

The **scheduler** gets the wheels turning



핵심•요점

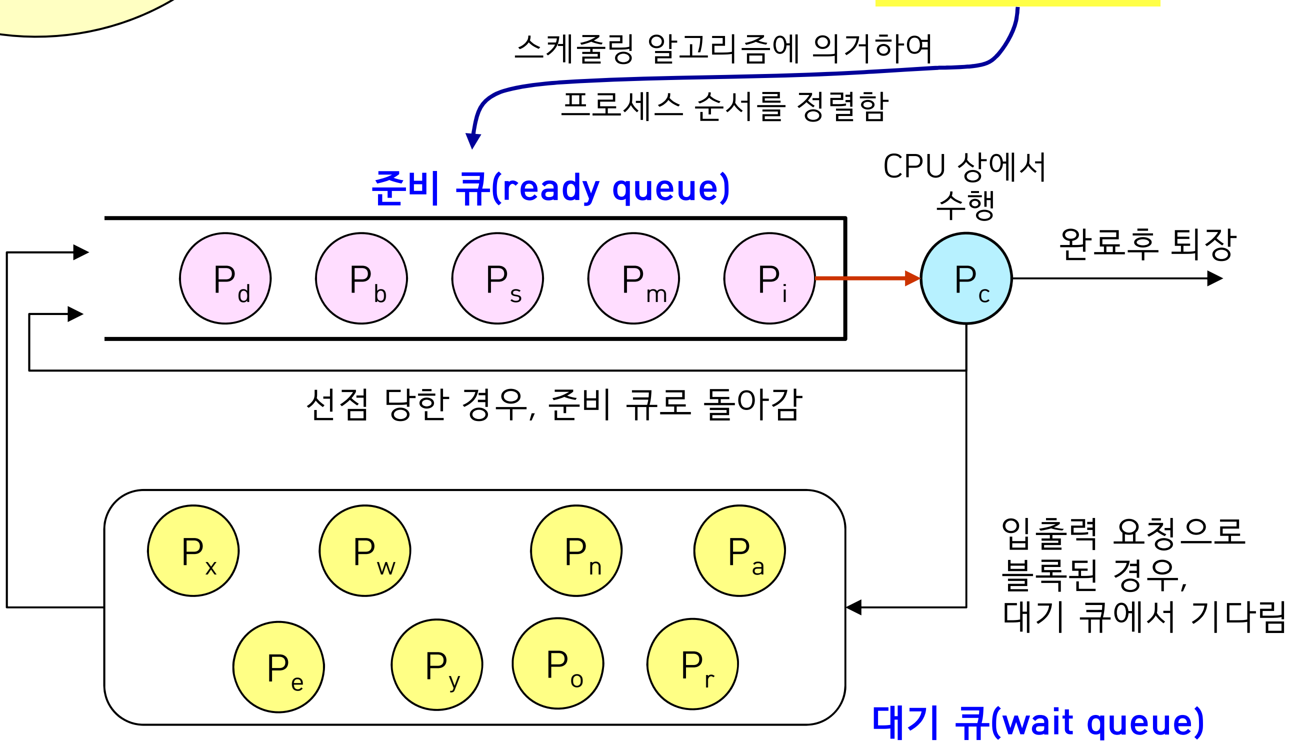
스케줄링 – 무엇을, 왜, 어떻게?

목표
- 처리율
- 효율
- 응답성
- 공정성
- 예측가능성

시스템이 요구하는 목표를 달성하라

Scheduler

모든 프로세스는 스케줄 되기 전, 준비큐 (*ready queue*)에 모임. 이 큐를 적절히 운용함으로서 시스템의 목적을 달성할 수 있음. 스케줄링 알고리즘 이용.



스케줄링 알고리즘에 의거하여
프로세스 순서를 정렬함
준비 큐(ready queue)
P_d P_b P_s P_m P_i
CPU 상에서 수행
P_c
완료후 퇴장
선점 당한 경우, 준비 큐로 돌아감
블록이 해제되면, 준비큐로 돌아감
P_x P_w P_n P_a
P_e P_y P_o P_r
입출력 요청으로 블록된 경우, 대기 큐에서 기다림
대기 큐(wait queue)

Core Ideas

Scheduling – What, Why and How?

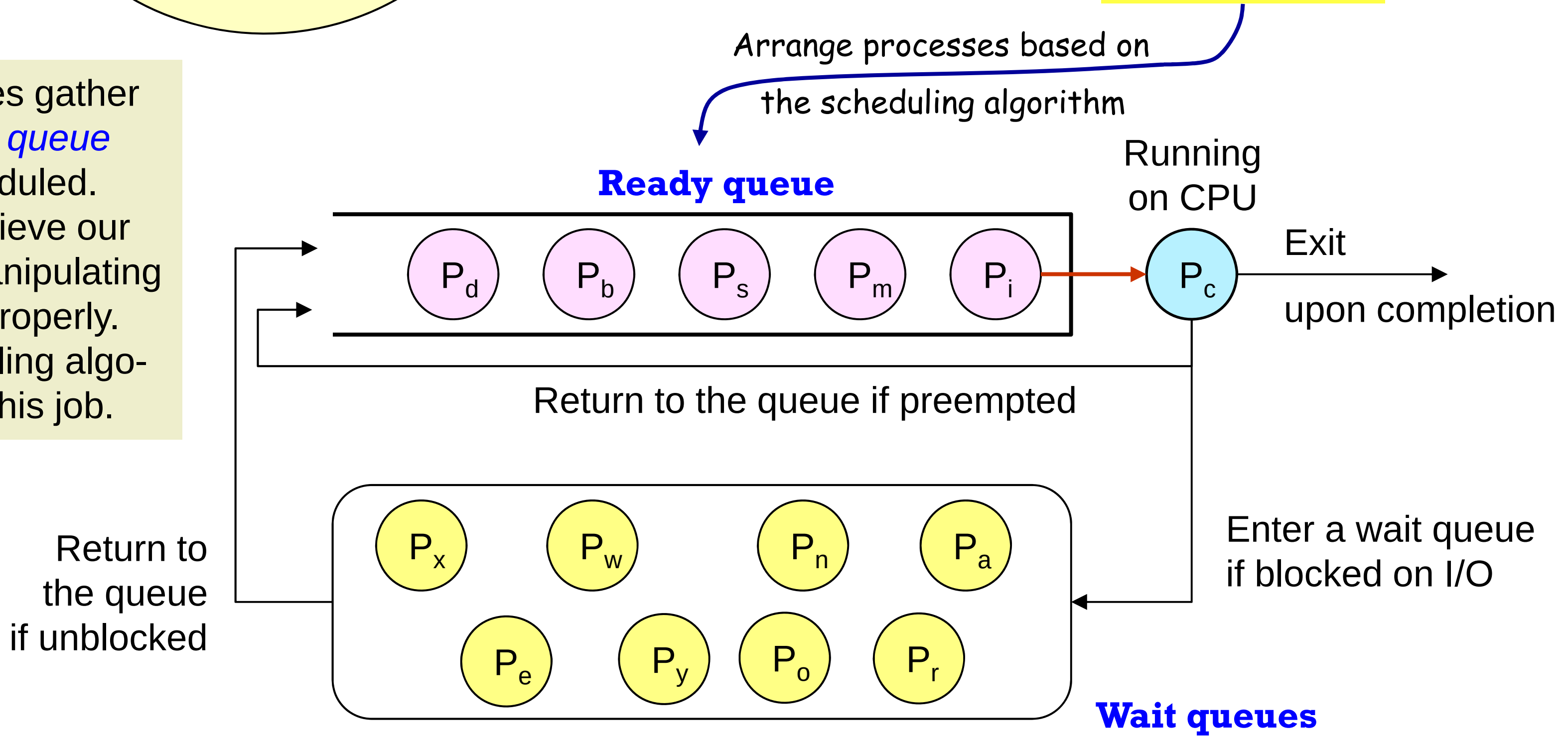
We aim at

- Throughput
- Efficiency
- Responsiveness
- Fairness
- Predictability

Meet one or more of these goals as necessary!

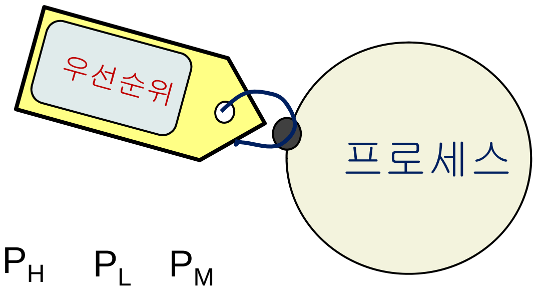
Scheduler

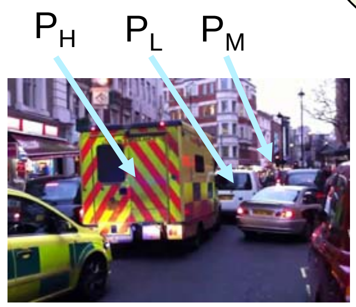
All processes gather in the *ready queue* before scheduled. We can achieve our goals by manipulating the queue properly. The scheduling algorithm does this job.


Arrange processes based on the scheduling algorithm
Ready queue
P_d
P_b
P_s
P_m
P_i
Running on CPU
P_c
Exit upon completion
Return to the queue if preempted
Enter a wait queue if blocked on I/O
Return to the queue if unblocked
P_x
P_w
P_n
P_a
P_e
P_y
P_o
P_r
Wait queues

핵심요점

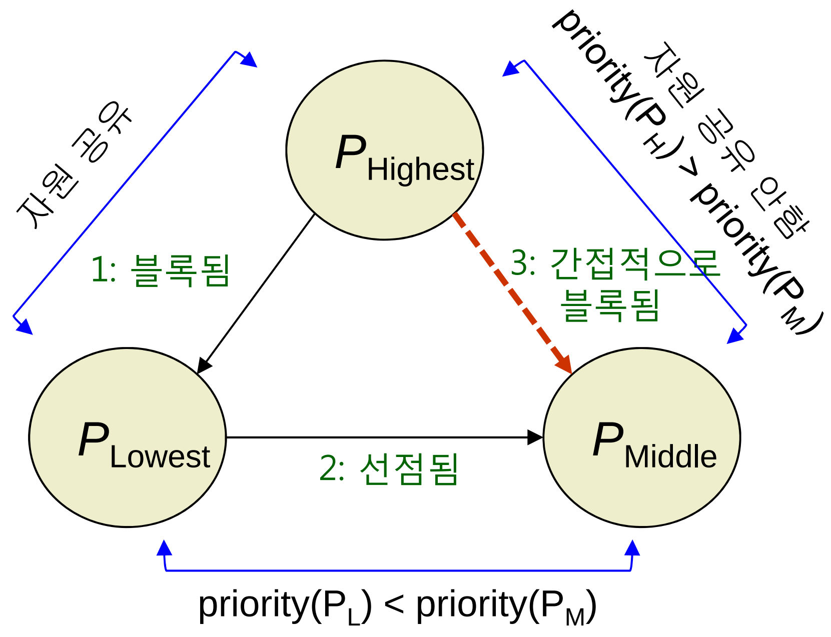
우선순위를 따르는 세상





선점(preemption)
우선순위가 높은 프로세스가 먼저 수행되고
우선순위가 낮은 프로세스는 수행 도중에
대기상태로 전환 → "선점됨 "

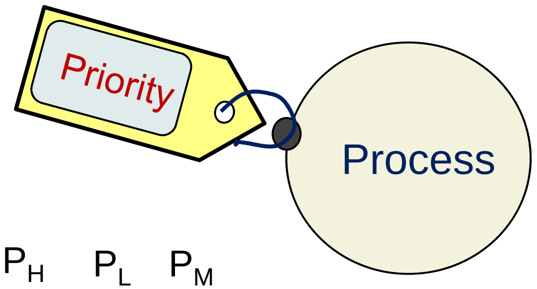
블로킹(Blocking)
우선순위가 낮은 프로세스가 먼저 수행되고
우선순위가 높은 프로세스는 대기상태로
전환 → "블록됨 "

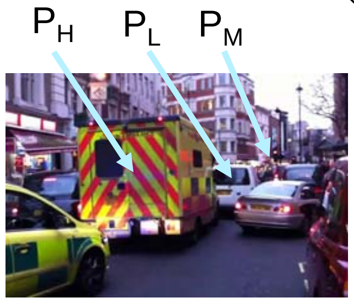


우선순위 전도(priority inversion)
$P_{Highest}$ 는 P_{Middle} 에 의해 블록되지 않아야 되는데 블록되었다면
→ "우선순위 전도"

프로세스의 총 수행시간 =
프로그램 "코드" 실행시간 +
블로킹된 시간 + 선점된 시간

Core Ideas Prioritized World





Preemption

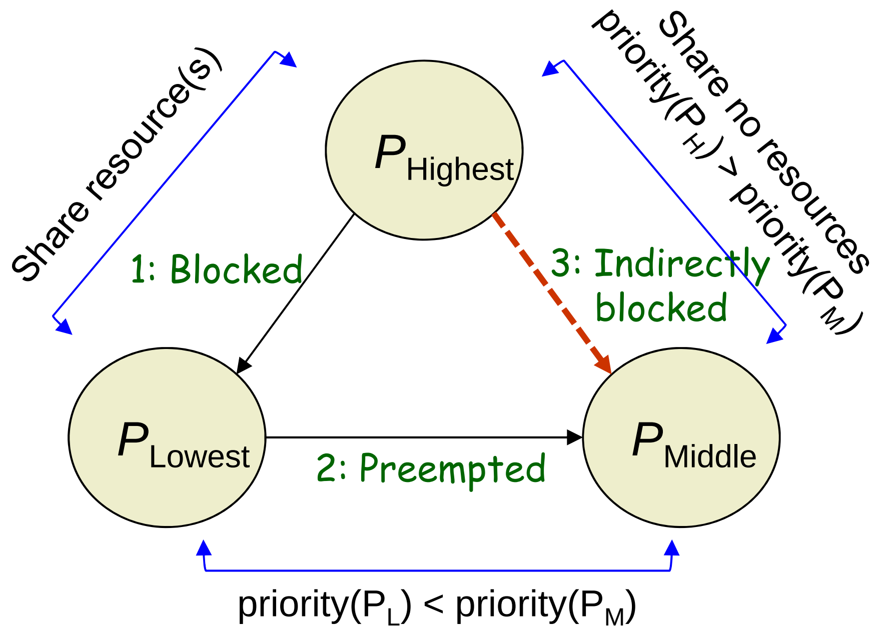
Higher priority process runs first.

Lower priority process is made to wait while running. → Preempted!

Blocking

Lower priority process runs first.

Higher priority process waits. → Blocked!



Priority inversion

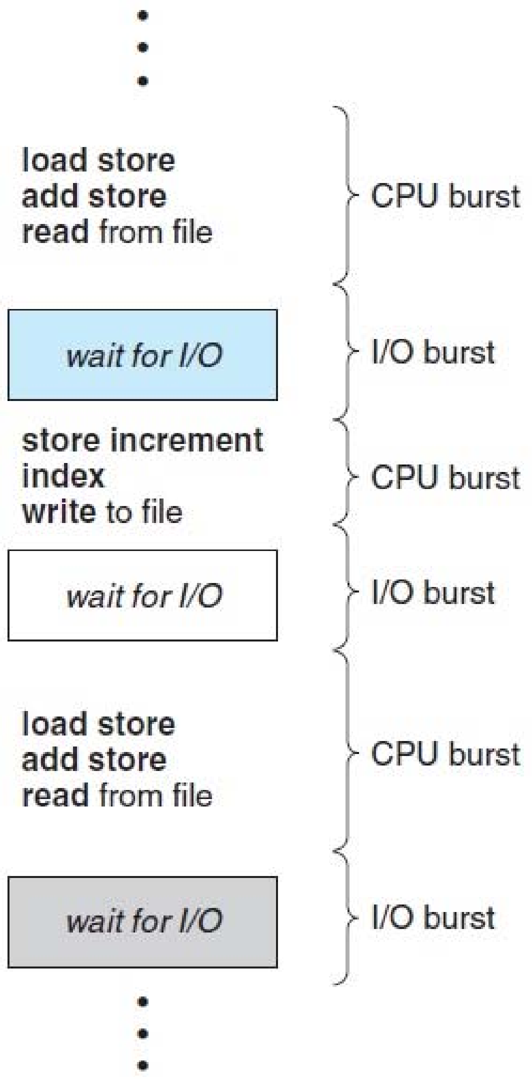
$P_{Highest}$ must not be blocked by P_{Middle}, but blocked → called *priority inversion*

Total execution time of a process =

Program "code" execution time + Blocking time + Preemption time

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
- CPU burst distribution is of main concern
- Real-world CPU execution
  - a large number of short CPU bursts (<8ms) and a small number of long CPU bursts.
  - An I/O-bound program typically has many short CPU bursts. A CPU-bound program might have a few long CPU bursts.



***Comment*:** Efficiency is the primary virtue of engineering practice. Nonetheless, computer engineers have been too much obsessed with CPU efficiency.

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
- 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
- Scheduling under 1 and 4 is **nonpreemptive**
  - Once the CPU has been allocated to a process, the process keeps the CPU until it releases the CPU (1, 4)
- All other scheduling (2, 3) is **preemptive**
  - A process may be forced to give way to another process while in execution
  - problem: race conditions
    - Consider access to shared data
    - Consider preemption while in kernel mode
    - Consider interrupts occurring during crucial OS activities

Dispatcher

- Dispatcher module gives control of the CPU to the process selected by the short-term scheduler; this involves:
  - switching context, i.e., load the context of a user process from a predetermined memory location
  - 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 and Goals

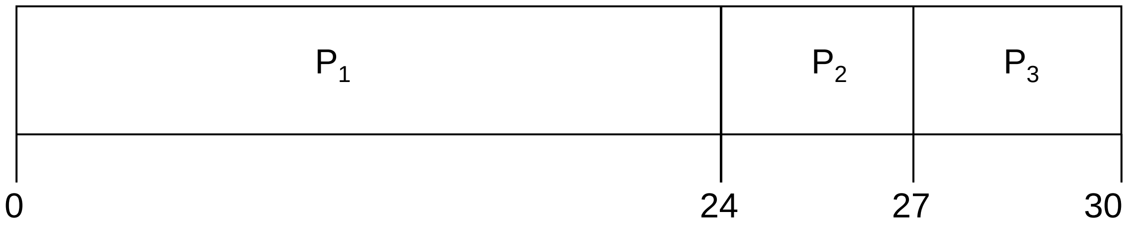
- **CPU utilization** – keep the CPU as busy as possible
  - Max CPU utilization
- **Throughput** – # of processes that complete their execution per time unit
  - Max throughput
- **Turnaround time** – amount of time to execute a particular process
  - Min turnaround time
- **Waiting time** – amount of time a process has been waiting in the ready queue
  - 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)
  - Min response time

Scheduling Algorithms

First-Come, First-Served (FCFS) Scheduling

Process	Burst Time
P1	24
P2	3
P3	3

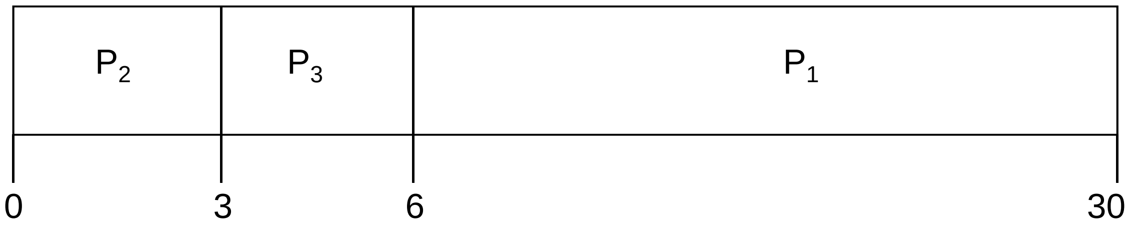
- Suppose that the processes arrive in the order: P1 , P2 , P3
- The Gantt Chart for the schedule is:



- Waiting time for P1 = 0; P2 = 24; P3 = 27
- Average waiting time: (0 + 24 + 27)/3 = 17

FCFS Scheduling (Cont.)

- Suppose that the processes arrive in the order:
  P2 , P3 , P1
- The Gantt chart for the schedule is:



- Waiting time for P1 = 6; P2 = 0; P3 = 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

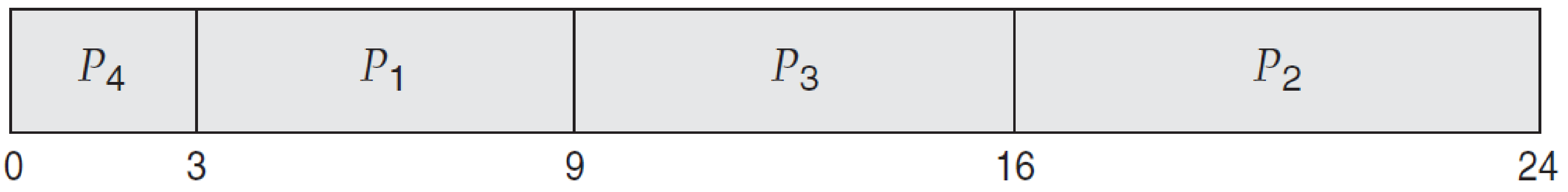
- Associate with each process the length of its next CPU burst
  - Use these lengths to schedule the process with the shortest time

- 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

Example of SJF

Process	Burst Time
P1	6
P2	8
P3	7
P4	3

- SJF scheduling chart



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

Determining Length of Next CPU Burst

- Can only estimate the length – should be similar to the 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. $\tau_{n+1} =$ predicted value for the next CPU burst
  3. $\alpha, 0 \le \alpha \le 1$
  4. Define: $\tau_{n+1} = \alpha\, t_n + (1-\alpha)\tau_n.$

  where the parameter α controls the relative weight of recent and past history in our prediction

- Commonly, α set to ½
- Preemptive version called **shortest-remaining-time-first**

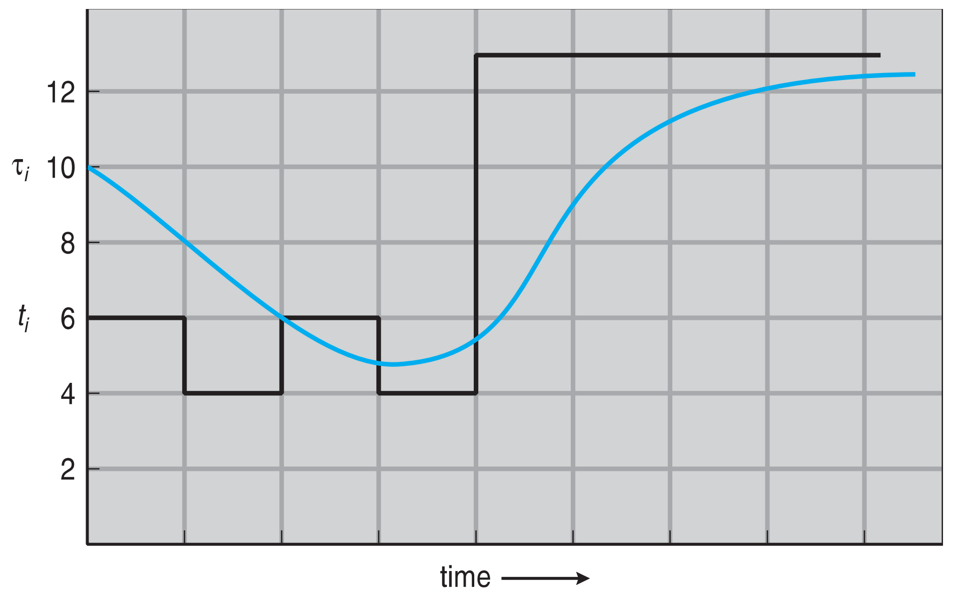
Examples of Exponential Averaging

- $\alpha = 0$
  - $\tau_{n+1} = \tau_n$
  - Recent history does not count
- $\alpha = 1$
  - $\tau_{n+1} = \alpha\ t_n$
  - Only the actual last CPU burst counts
- If we expand the formula, we get:

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

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

- Example: Prediction of the length of the next CPU burst
  - Based on exponential averaging
  - $\alpha = \frac{1}{2}$, $\tau_0 = 10$



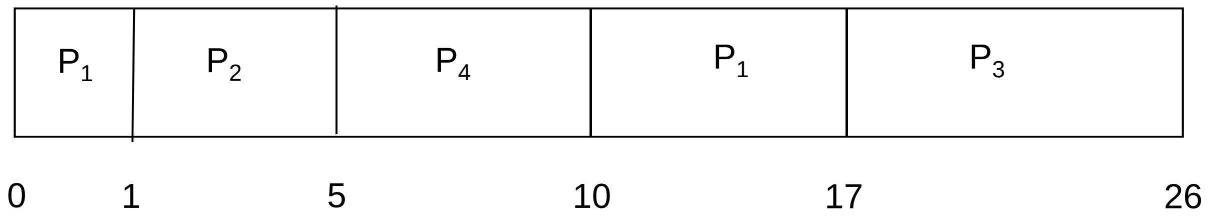
CPU burst (t_i)		6	4	6	4	13	13	13	…
"guess" (τ_i)	10	8	6	6	5	9	11	12	…

Example of Shortest-remaining-time-first

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

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

- *Preemptive* SJF Gantt Chart



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

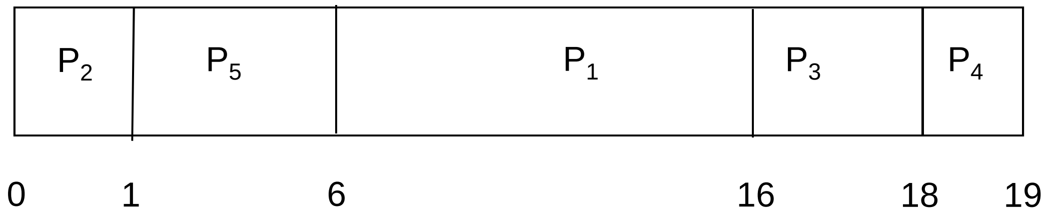
Priority Scheduling

- A priority number (integer) is associated with each process
- The CPU is allocated to the process with the highest priority (smallest integer ≡ highest priority)
  - Preemptive
  - Nonpreemptive
- 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

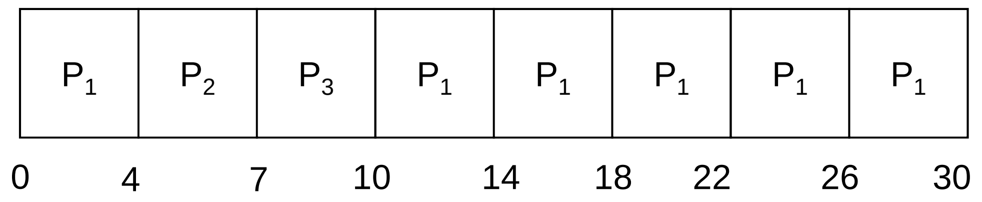
Round Robin (RR)

- 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 is q, then each process gets $1/n$ of the CPU time in chunks of at most q time units at once. No process waits more than $(n-1)q$ time units.
- 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

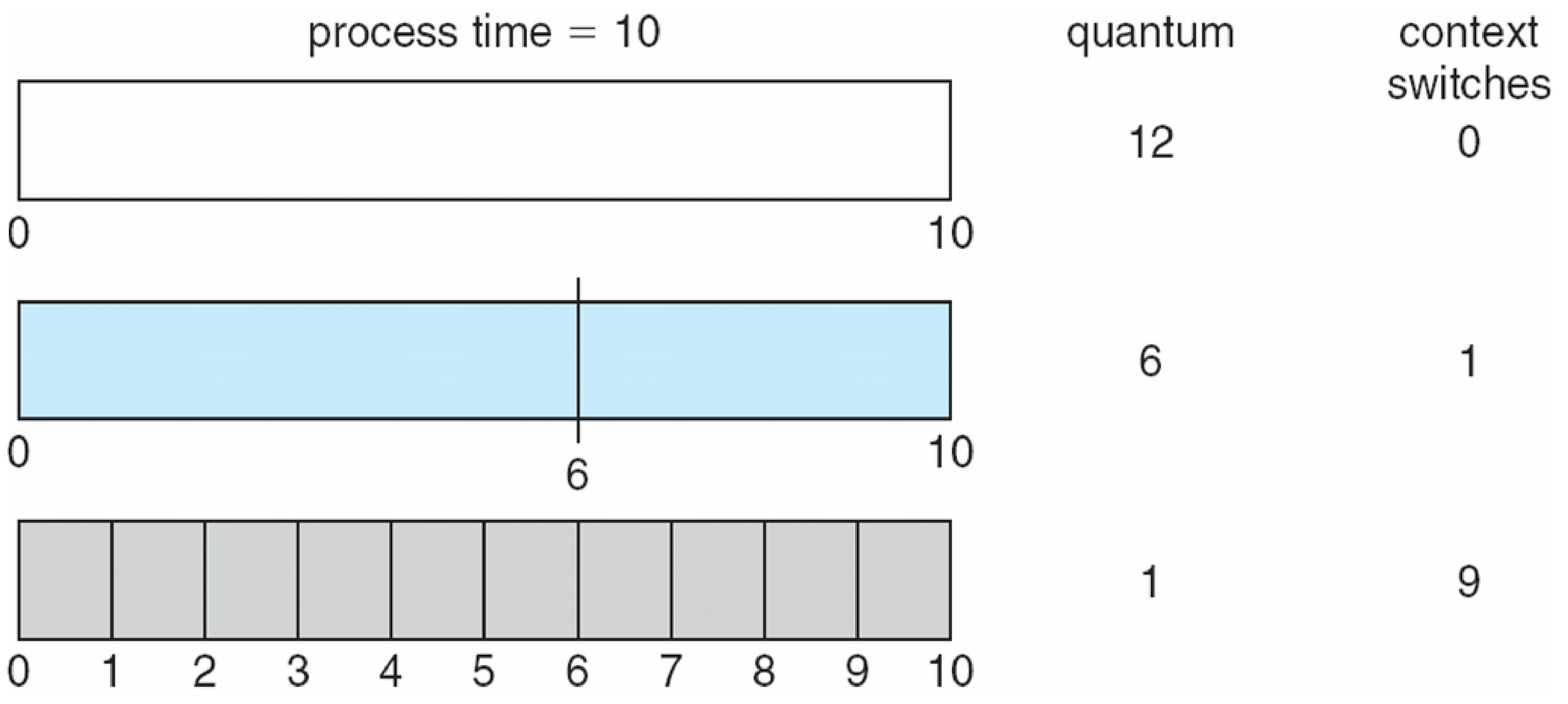
Process	Burst Time
P_1	24
P_2	3
P_3	3

- The Gantt chart is:

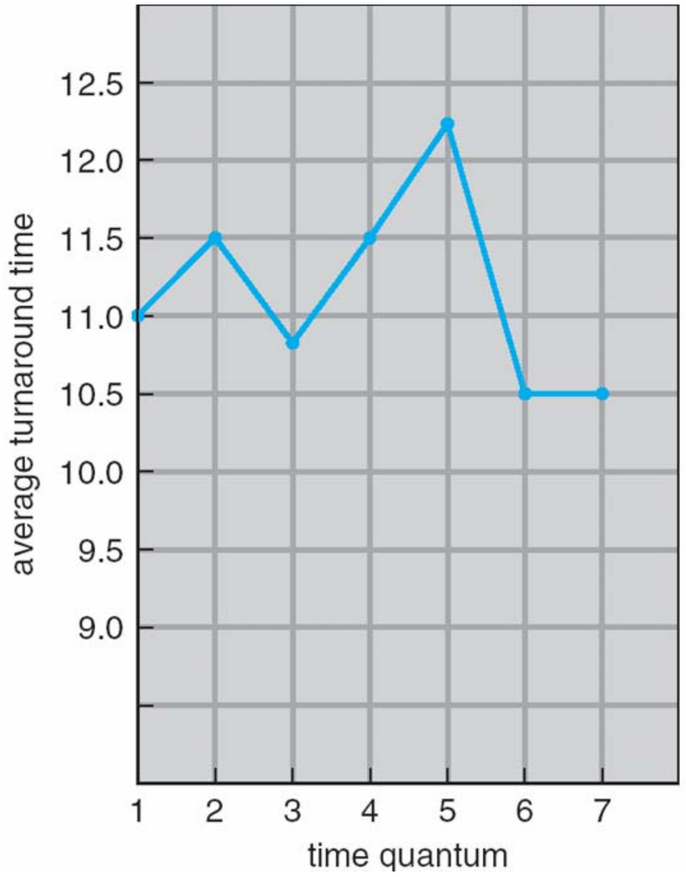


- Typically, higher average turnaround than SJF, but better ***response***
- q should be large compared to context switch time
- q usually 10ms to 100ms, context switch < 10 usec

Time Quantum and Context Switch Time



Turnaround Time Varies With 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

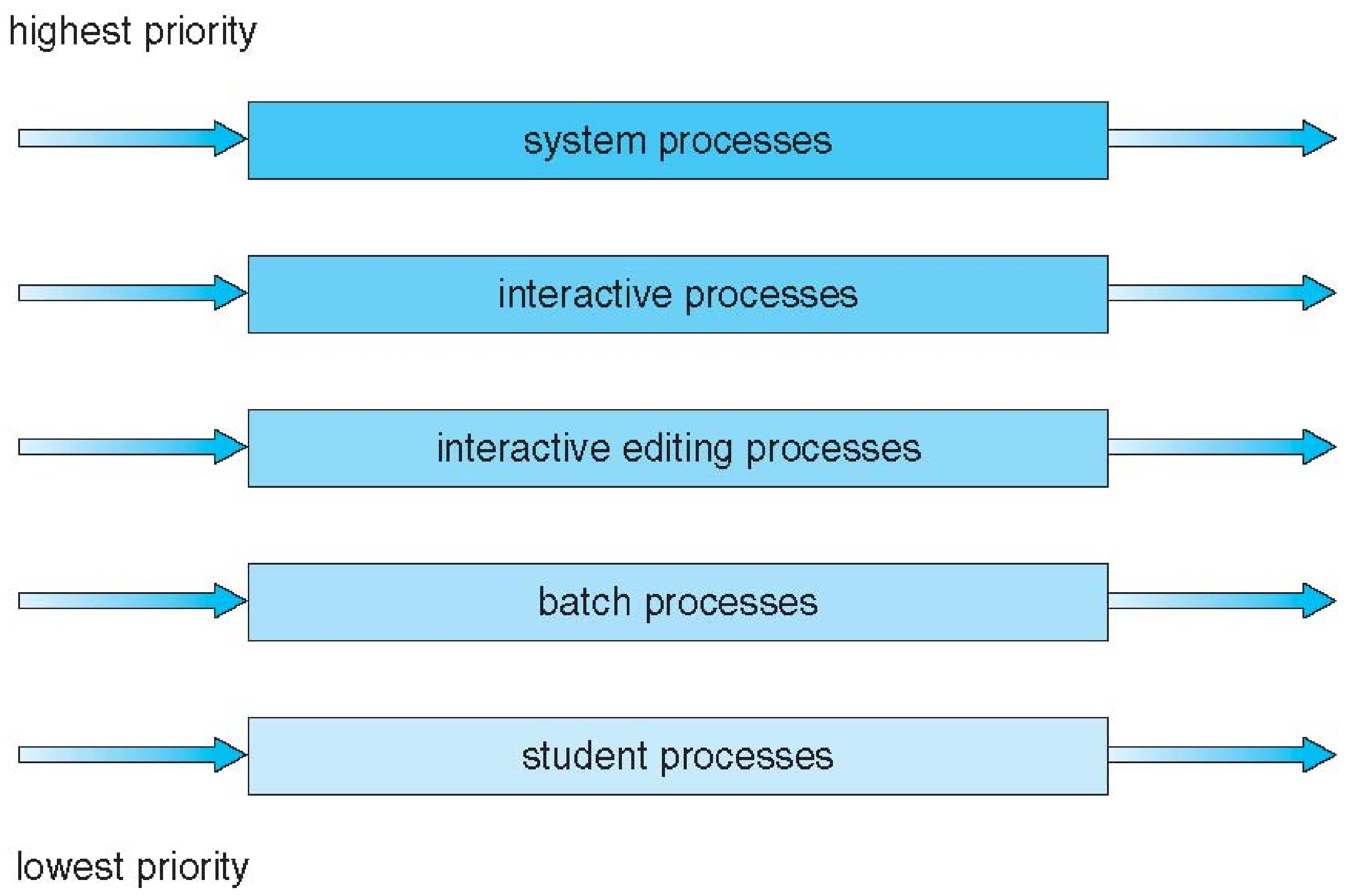
Multilevel Queue

- Ready queue is partitioned into separate queues, eg:
  - foreground (interactive)
  - background (batch)
- Process permanently in a given queue

- 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 amongst its processes; i.e., 80% to foreground in RR, 20% to background in FCFS

Multilevel Queue Scheduling



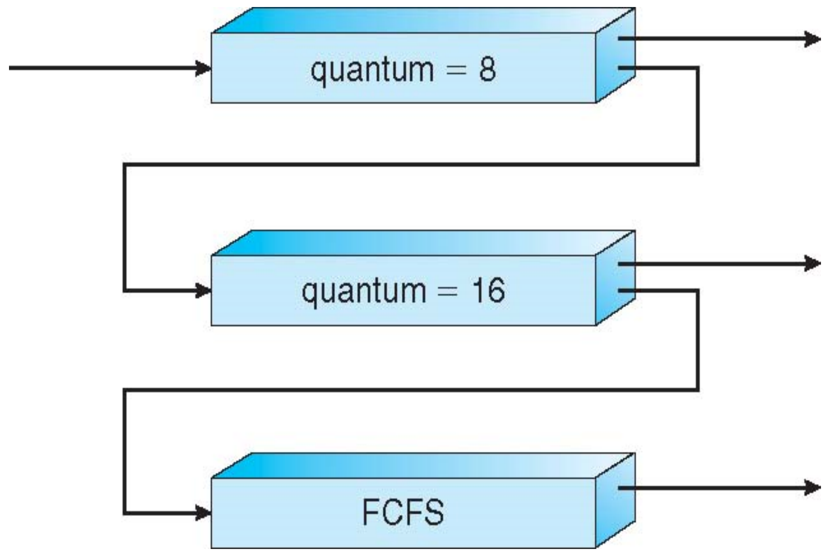
Multilevel Feedback Queue

- A process can move between the various queues; aging can be implemented this way

- Multilevel-feedback-queue scheduler defined by the following parameters:
  - number of queues
  - scheduling algorithms for each queue
  - method used to determine when to upgrade a process
  - method used to determine when to demote a process
  - method used to determine which queue a process will enter when that process needs service

Example of Multilevel Feedback Queue

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



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

Thread Scheduling

- Distinction between user-level and kernel-level threads
- When threads are supported by the kernel, threads are scheduled (instead of processes)
- Many-to-many models: thread library schedules user-level threads to run on LWP (light-weight process)
  - Known as **process-contention scope** (**PCS**) since scheduling competition is within the process
  - Typically done via priority set by programmer
- Kernel thread scheduled onto available CPU is **system-contention scope** (**SCS**) – competition among all threads in system

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

- The Pthread IPC provides two functions for getting and setting the contention scope policy:
  - **`pthread_attr_setscope(pthread_attr_t *attr, int scope)`**
  - **`pthread_attr_getscope(pthread_attr_t *attr, int *scope)`**

- 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");
   }

/* 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

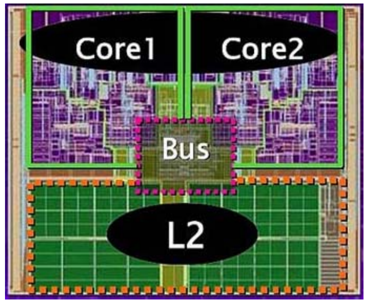
- CPU scheduling more complex when multiple CPUs are available
- Homogeneous processors within a multiprocessor
- **Asymmetric multiprocessing** – only one processor accesses the system data structures, alleviating the need for data sharing
- **Symmetric multiprocessing** (**SMP**) – each processor is 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

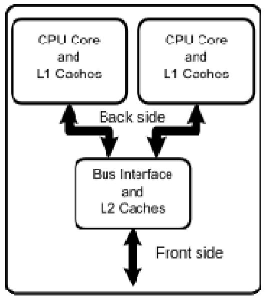
Multiple-Processor Scheduling – Load Balancing

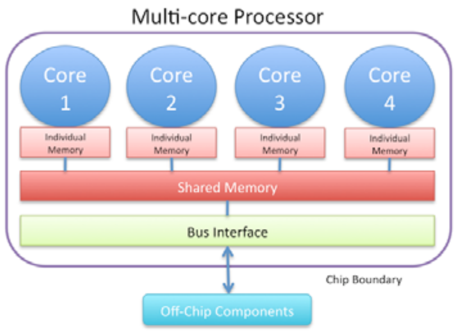
- If SMP, need to keep all CPUs loaded for efficiency
- **Load balancing** attempts to keep workload evenly distributed
- **Push migration** – periodic task checks load on each processor, and if found pushes task from overloaded CPU to other CPUs
- **Pull migration** – idle processors pulls waiting task 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
- Hyper-Threading (HT) technology - Intel
  - Hardware support for multithreading on each core – e.g., extra register set, PC, interrupt controller, etc.
  - Core components switch between two threads, either automatically or in case of memory stall
  - Allows each core to execute two software threads at the same time
  - A core appear to the OS as two virtual or logical processors, thus the OS can schedule two processes at once







Multithreaded Multicore System

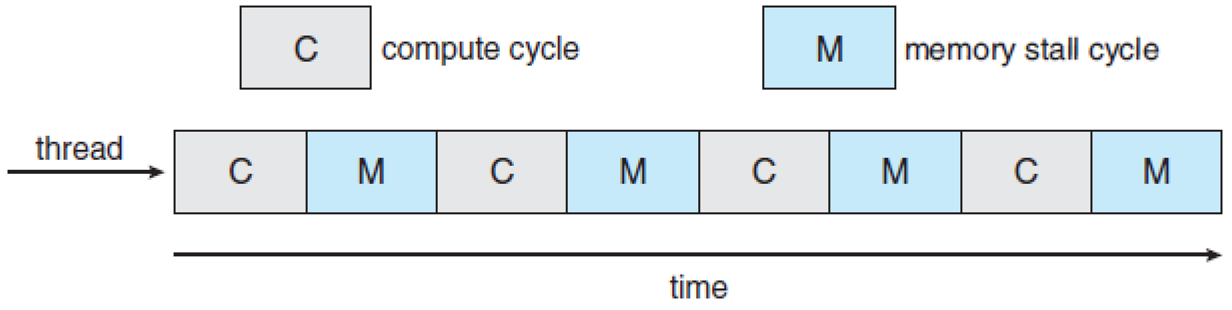
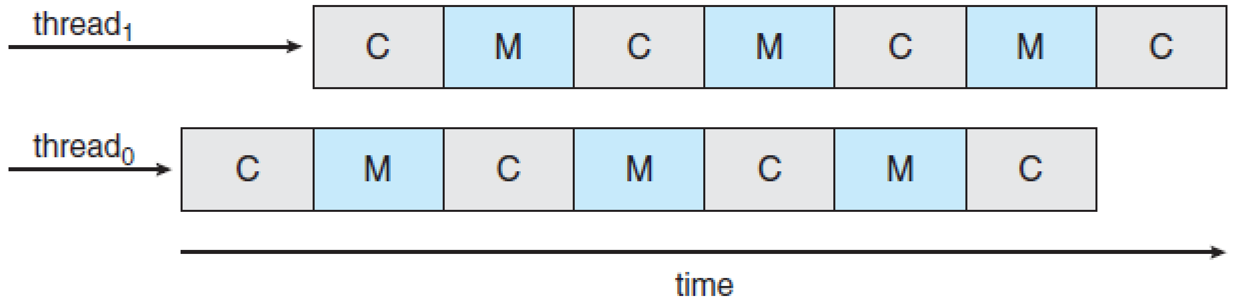


Figure 6.10 Memory stall.

While accessing memory, the processor is forced to wait for the data to become available. This situation, known as a memory stall, may occur for various reasons, such as a cache miss.



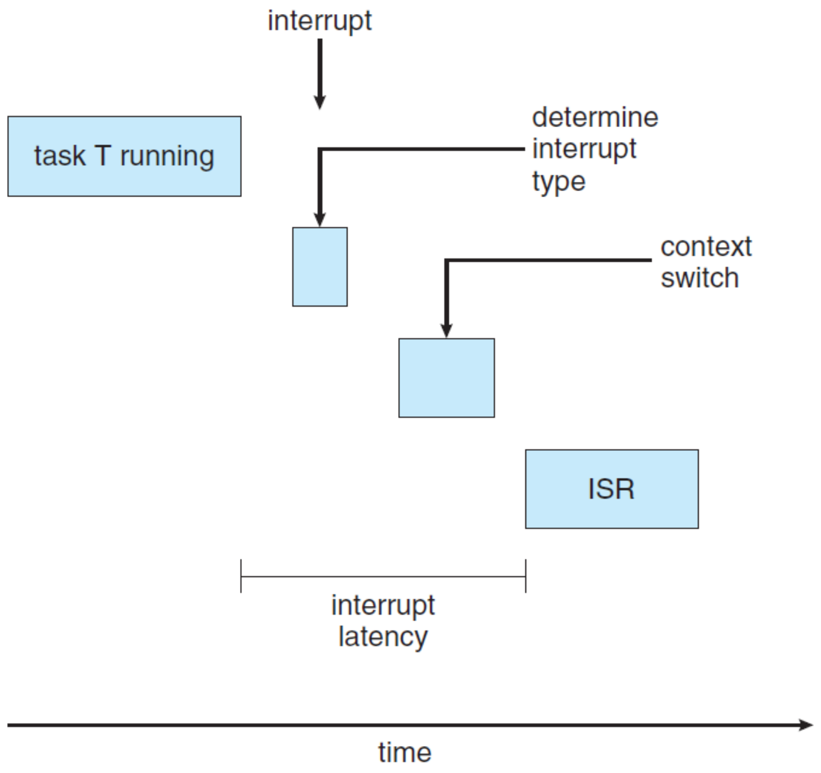
case of a dual-threaded core

Figure 6.11 Multithreaded multicore system.

Multithreaded processor cores: two (or more) hardware threads are assigned to each core. If one thread stalls while waiting for memory, the core can switch to another thread. Refer to Hyper-Threading technology.

Real-Time CPU Scheduling

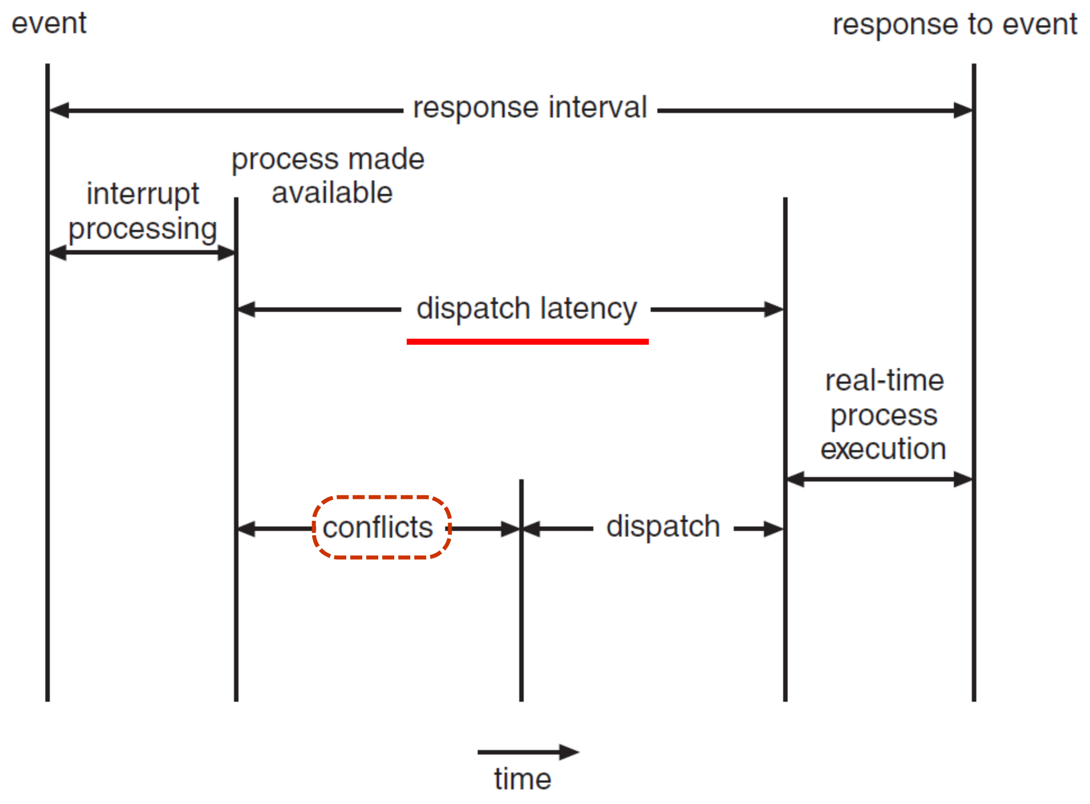
- Can present obvious challenges
- **Soft real-time systems** – no guarantee as to when critical real-time process will be scheduled
- **Hard real-time systems** – task must be serviced by its deadline
- Two types of latencies affect performance
  1. Interrupt latency
     - time from arrival of interrupt 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.)

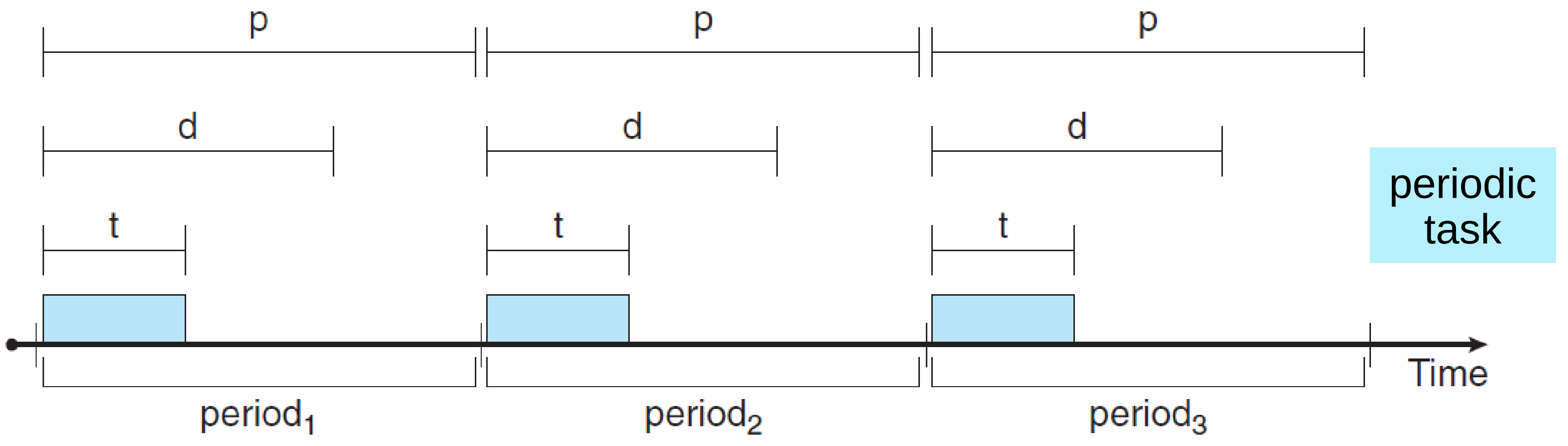
- Conflict phase of dispatch latency:
  1. Preemption of any process running in kernel mode
  2. Release by low-priority process of resources needed by high-priority processes

- The POSIX.1b standard
  - Specifies API that provides functions for managing real-time threads



Priority-based Scheduling

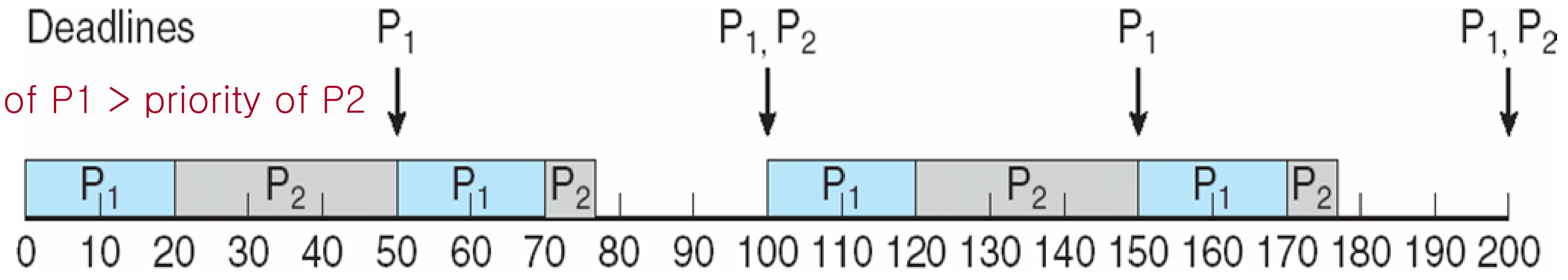
- For real-time scheduling, scheduler must support preemptive, priority-based scheduling
  - But only guarantees soft real-time
- For hard real-time must also provide ability to meet deadlines
- Processes have new characteristics: **periodic** ones require CPU at constant intervals
  - Has processing time *t*, deadline *d*, period *p*
  - $0 \leq t \leq d \leq p$
  - **Rate** of periodic task is 1/*p*



Rate Monotonic Scheduling

- Assume that
  - all tasks are periodic and never interdependent
  - deadline = end of current period
- A priority is assigned statically based on the inverse of its period, i.e., higher rate → higher priority
  - Shorter periods = higher priority
  - Longer periods = lower priority
- The **rate-monotonic (RM)** scheduling algorithm
  - schedules periodic tasks using the aforementioned static priority policy with preemption.

priority of P1 > priority of P2



period of P_1 = 50, period of P_2 = 100

Rate Monotonic Scheduling (Cont.)

- Sufficient condition
  - CPU utilization rate u_i = processing time (t_i) / period (p_i) for task *i*
  - If the total utilization rate has least upper bound $U(n) = n(2^{1/n} - 1)$ where n = #tasks, there exists a feasible rate monotonic schedule. That is,

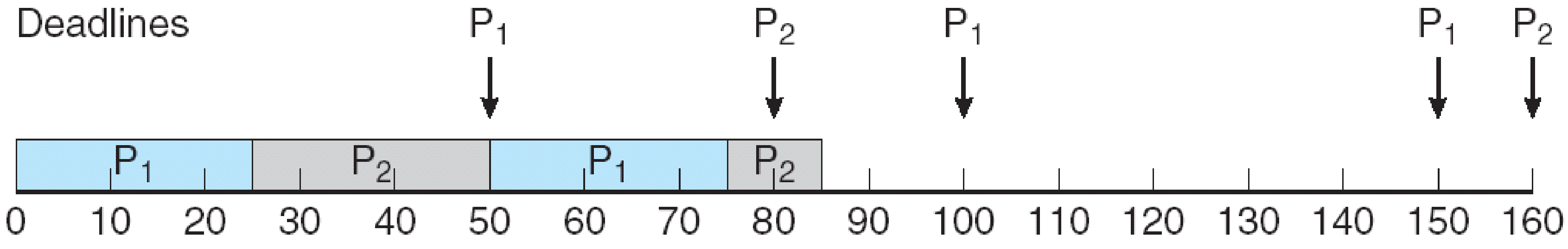
$$\sum_{i=1}^{n} u_i \le n(2^{\frac{1}{n}} - 1) = U(n) \;\; where \;\; u_i = \frac{t_i}{p_i}$$

    - e.g., U(1) = 1.0, U(2) = 0.828, U(3) = 0.779, U(∞) = ln2

- Observations
  - Although the above equation is not satisfied, all the tasks may still be scheduled.
  - If the above equation is satisfied, all the tasks must be schedulable

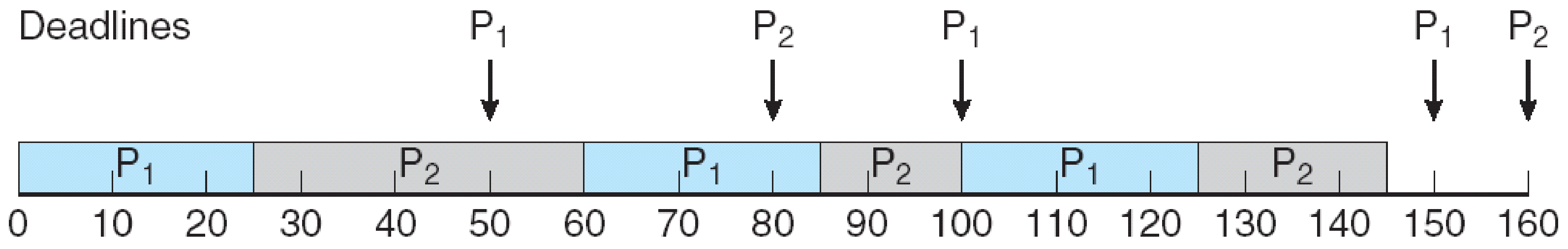
RM Scheduling - Examples

- Given two processes, P_1 and P_2
- Case 1: Schedulable
  - $p_1 = 50, p_2 = 100.\ t_1 = 20, t_2 = 35$
  - To check to see if they meet their deadlines, we use the sufficient condition based on CPU utilization
  - CPU utilization of P_1 and P_2: $u_1 = 20/50 = 0.40, u_2 = 35/100 = 0.35$ ➔ total CPU utilization of $0.75 < U(2)\ (= 0.82)$
  - Therefore, P_1 and P_2 are schedulable (shown in a preceding slide)
- Case 2: Missed deadlines with RM scheduling
  - $p_1 = 50, p_2 = 80.\ t_1 = 25, t_2 = 35$
  - $u_1 = 25/50 = 0.50, u_2 = 35/80 = 0.44$ ➔ total CPU utilization of $0.94 > U(2)\ (= 0.82)$
  - Therefore, P_1 and P_2 may be unschedulable



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
- Earliest-deadline-first (EDF) scheduling *dynamically* assigns priorities according to deadline.
- Observations
  - For every scheduling event, perform EDF scheduling by checking the deadline of each task in the ready queue
  - Optimal in the sense of CPU utilization
  - In reality, hard to estimate the deadline at runtime



Operating System Examples - Linux

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

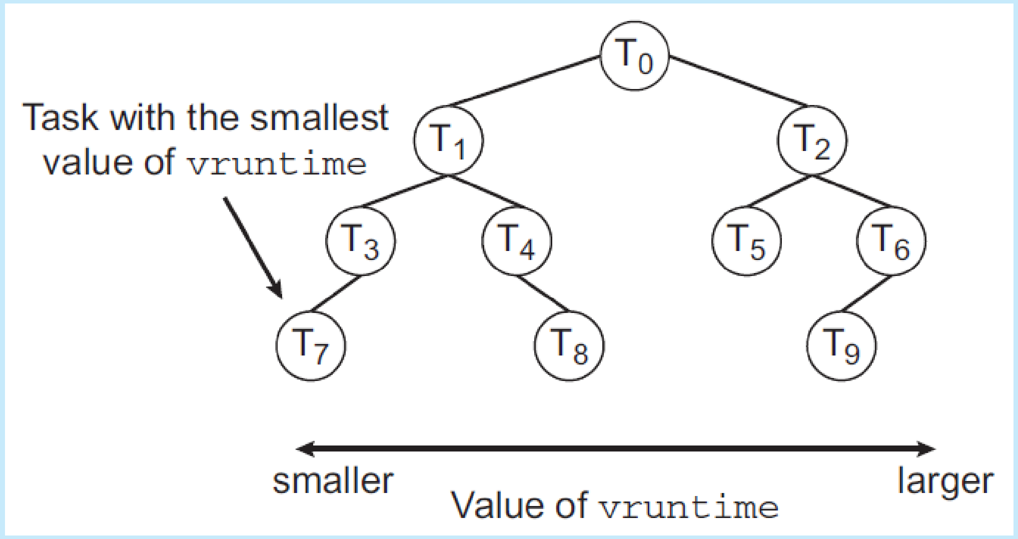
Linux Scheduling in Version 2.6.23 +

- *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 the 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

Linux CFS Performance

CFS PERFORMANCE

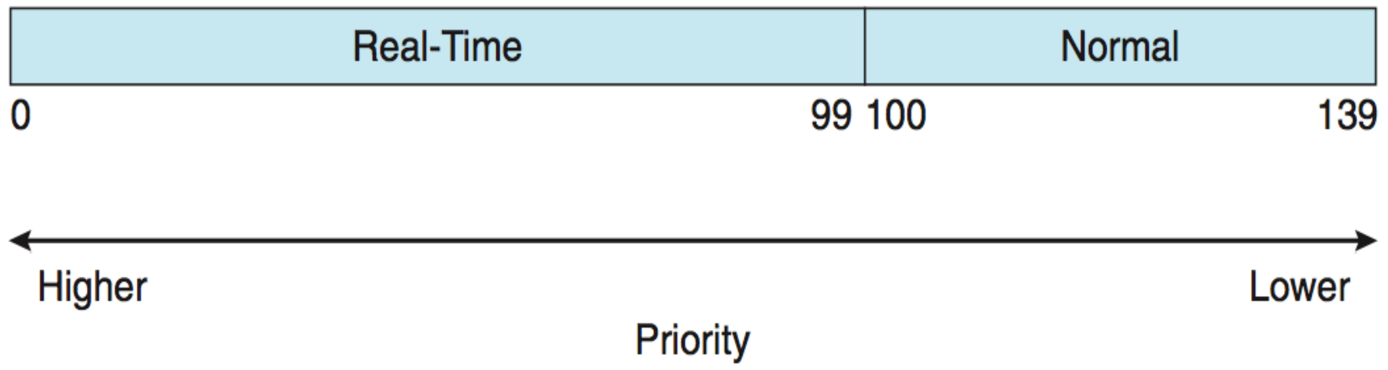
The Linux CFS scheduler provides an efficient algorithm for selecting which task to run next. Each runnable task is placed in a red-black tree—a balanced binary search tree whose key is based on the value of `vruntime`. This tree is shown below:



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**

Summary

- CPU 스케줄러
  - 선점 vs. 비선점(nonpreemptive)
- 디스팻쳐
- 스케줄러의 평가기준(성능척도)
- 스케줄링 알고리즘
  - FCFS, SJF, SRTF, 우선순위에 의한 스케줄링, round robin, multilevel queue, multilevel feedback queue
- 쓰레드 스케줄링
  - 커널 쓰레드는 system contention scope, pthread scheduling
- 멀티프로세서 스케줄링
  - 프로세서 밀착성(affinity), 부하균배(load balancing), 멀티코어 스케줄링
- 실시간 스케줄링
  - 실시간시스템 특성: 시간 제약조건
  - 스케줄링: 우선순위 기반, rate monotonic priority-based, EDF
- 사례: Linux, Windows

- CPU scheduler
  - Preemptive, nonpreemptive
- Dispatcher
- Scheduling criteria (performance metrics)
- Scheduling algorithms
  - FCFS, SJF, SRTF, priority scheduling, round robin, multilevel queue, multilevel feedback queue
- Thread scheduling
  - Kernel-level thread - system contention scope, pthread scheduling
- Multiprocessor scheduling
  - Processor affinity, load balancing, multicore scheduling
- Real-time systems
  - Characteristics: time constraints
  - Scheduling: priority based, rate monotonic priority-based, EDF
- Examples: Linux, Windows