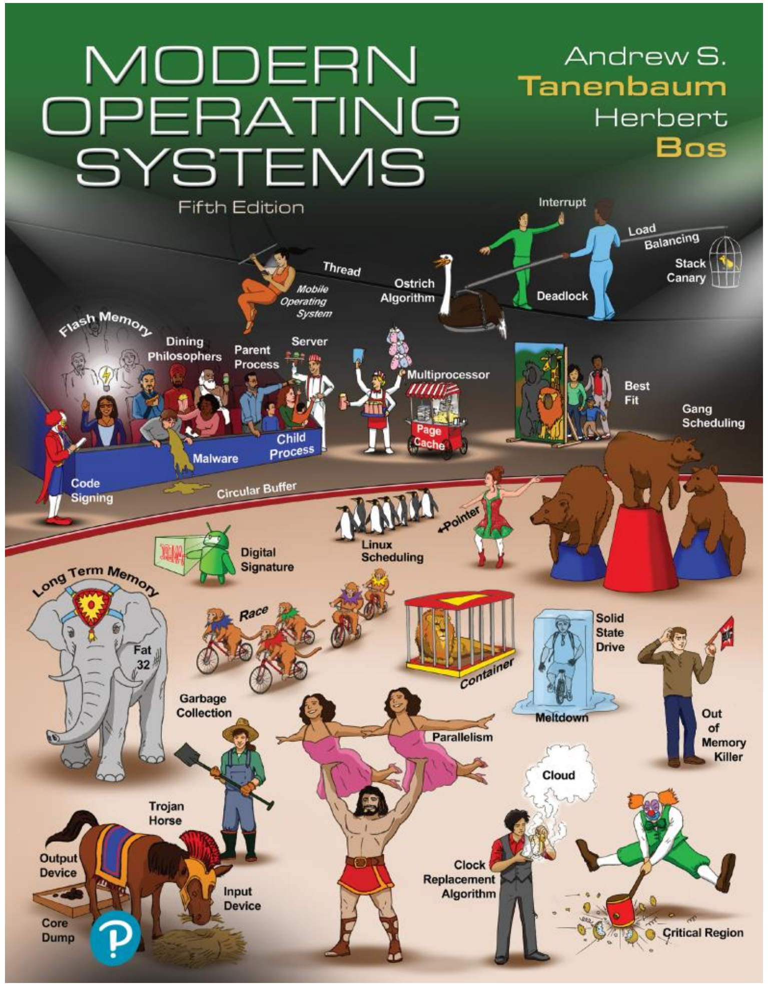
Modern Operating Systems

Fifth Edition



Chapter 2

Scheduling

Pearson

Introduction to Scheduling Process Behavior

CPU bound vs. I/O bound processes

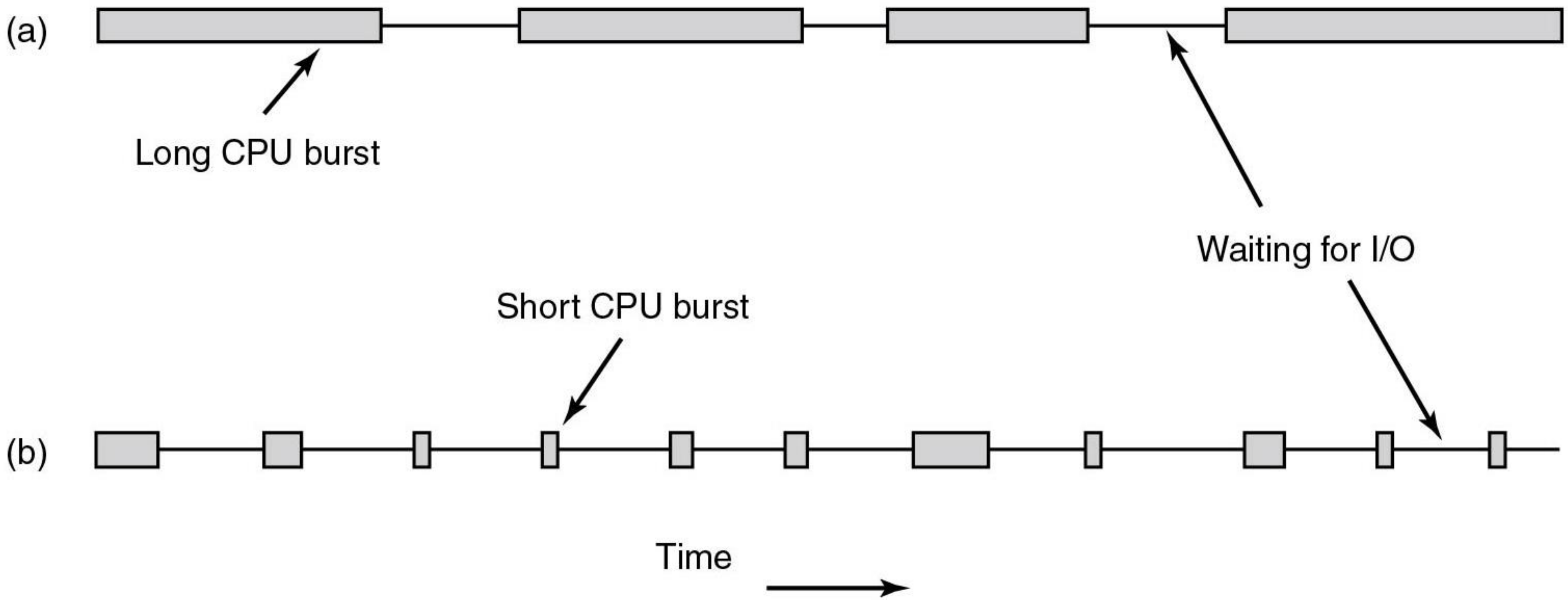
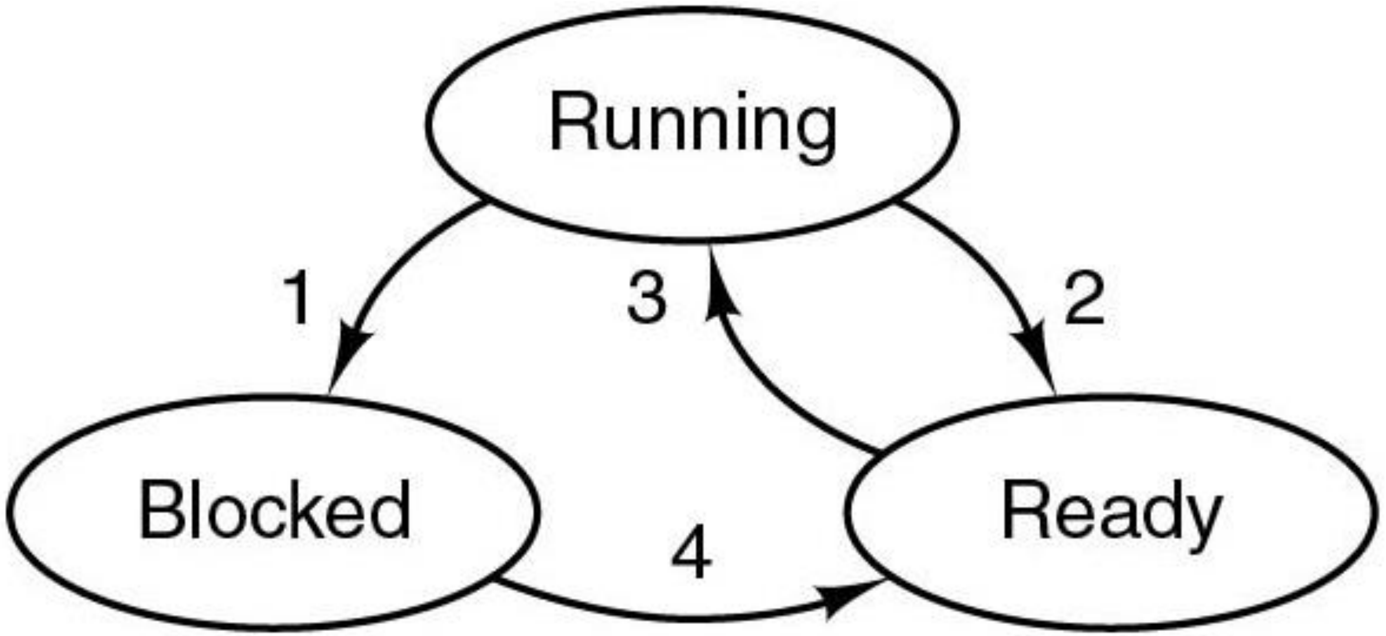


Figure 2.40 Bursts of CPU usage alternate with periods of waiting for I/O.
(a) A CPU-bound process. (b) An I/O-bound process.

Process State Revisited



If more processes ready than CPUs available:

- **Scheduler** decides which process to run next
- Algorithm used by scheduler is called **scheduling algorithm**

When to Schedule?

- ***Process exits***
- ***Process blocks on I/O, Semaphore, etc.***
- *When a new process is created*
- *When an interrupt occurs:*
  - *I/O, clock, syscall, etc.*

Preemptive vs **non-preemptive** scheduling?

Categories of Scheduling Algorithms

- Batch
- Interactive
- Real time

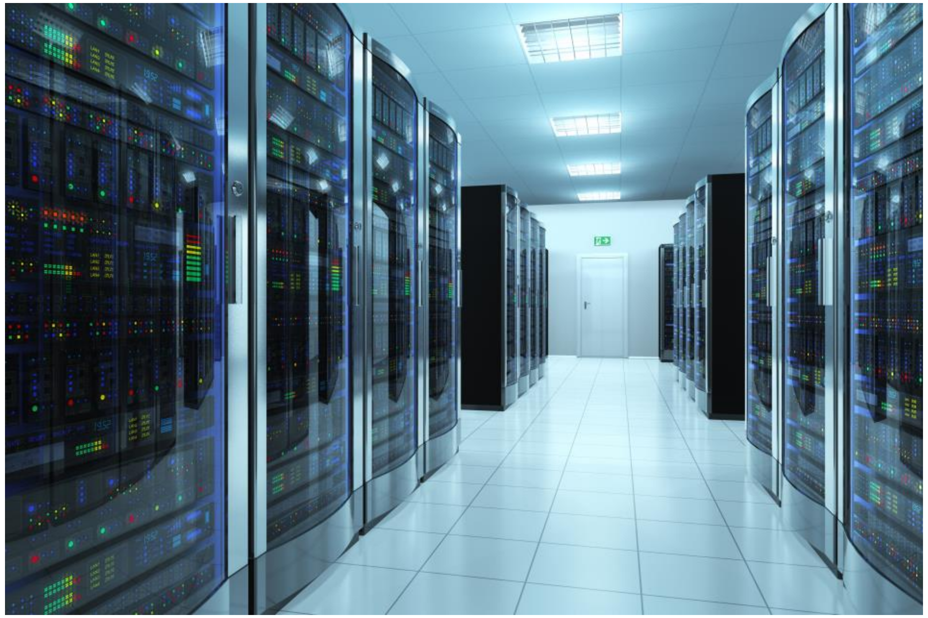
Scheduling Algorithm Goals

- Different goals for different systems
  - Batch
  - Interactive
  - Real time
- All systems:
  - Fairness - giving each process a fair share of the CPU
  - Policy enforcement - seeing that stated policy is carried out
  - Balance - keeping all parts of the system busy

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
- **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)

Batch Systems



Scanrail/123RF

- Throughput : maximize jobs per hour
- Turnaround time : minimize time between submission and termination
- CPU utilization : keeping the CPU busy all the time

First-Come First-Served

- Process jobs in order of their arrival
- Non-preemptive
- Single Process Queue
  - New jobs or blocking processes are added to the end of the queue
- “Convoy Effect” if only few CPU bound and many I/O bound processes

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

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
P_1	6
P_2	8
P_3	7
P_4	3

SJF scheduling chart

P_4	P_1	P_3	P_2
0–3	3–9	9–16	16–24

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

Interactive Systems

- **Response time:** respond to requests quickly
- **Proportionality:** meet users' expectations

McClatchy-Tribune/Tribune Content Agency LLC/Alamy Stock Photo

Apple MacIntosh (1984)

Round Robin Scheduling

- Preemptive scheduling algorithm
- Each process gets a **time slice** or **quantum**
- If process is still running at end of quantum it gets preempted end goes to end of ready queue
- Question: How big should the quantum be?

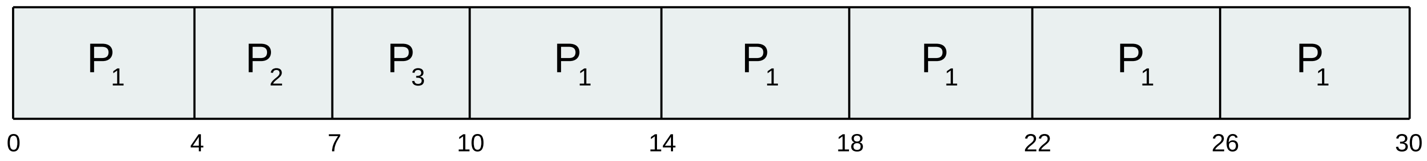
CPU utilization vs. response time

(a) (b)

Example of RR with Time Quantum = 4

Process	Burst Time
P_1	24
P_2	3
P_3	3

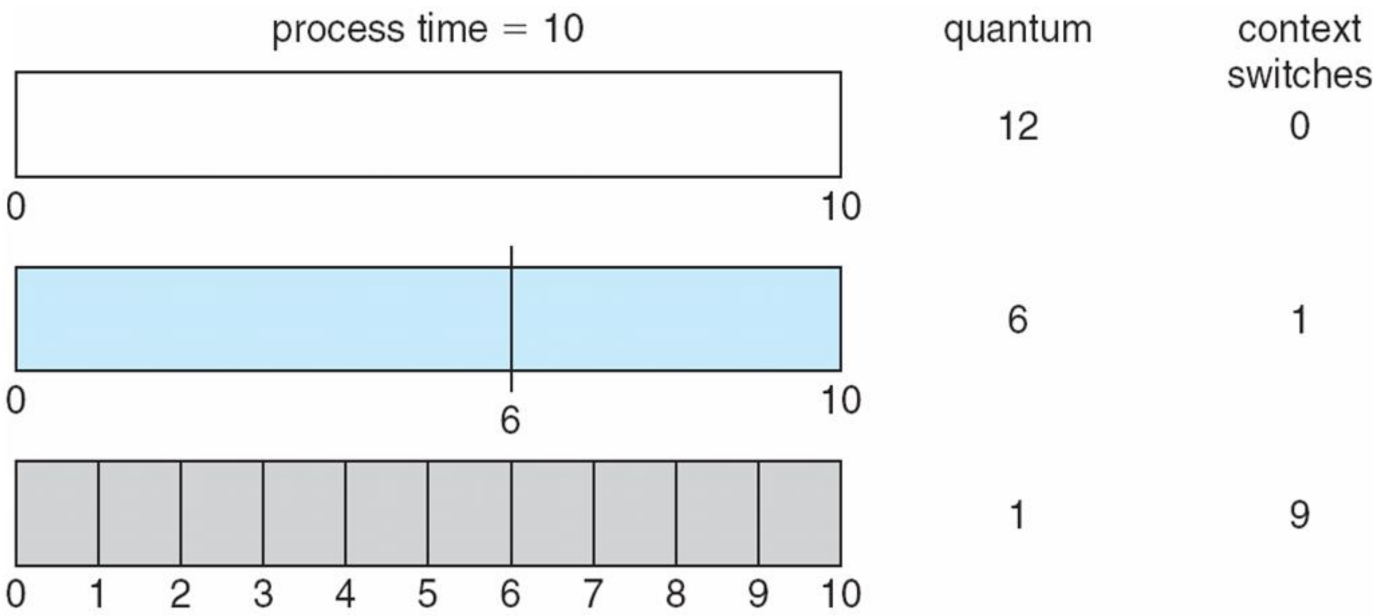
The Gantt chart is:



Typically, higher average turnaround than SJF, but better ***response***

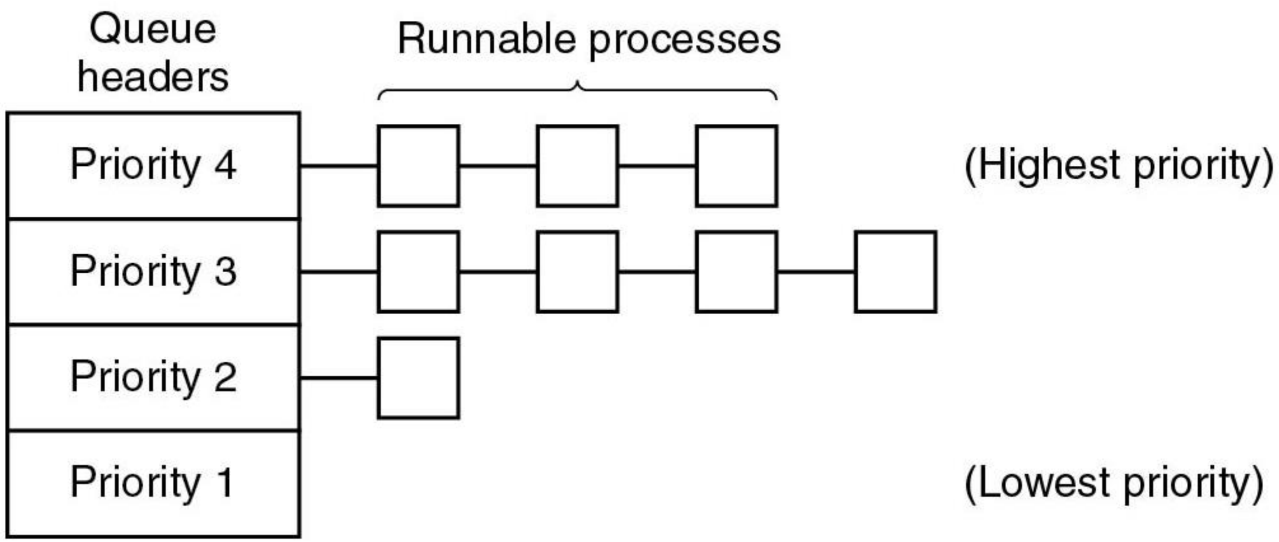
- q should be large compared to context switch time
- q usually 10ms to 100ms, context switch < 10 usec

Time Quantum and Context Switch Time



Priority Scheduling (1 of 2)

Simplest, multiple queues

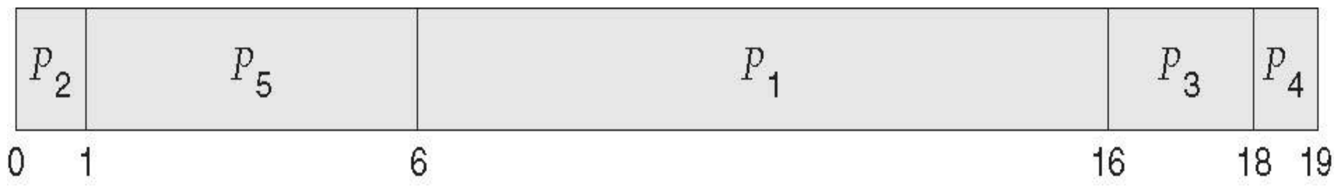


- Similar to round robin but several ready queues
- Next process is picked from queue with highest priority
- Static vs. dynamic priorities
- What processes should have high priorities?

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

Shortest Process Next

Problem: How to minimize response time for each priority queue?

Idea: Use shortest "job" first and try to best predict next running time

whatever runs between the waits

Solution: Form weighted average of previous running times of process $\longrightarrow$ **Aging**

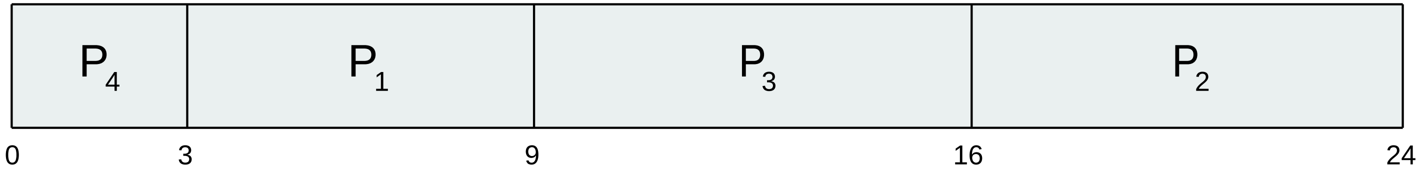
$$T_{k+1} = a * T_{k-1} + (1-a) * T_k$$

Easy to implement when $a = 1/2$

Example of SJF

Process	Burst Time
P_1	6
P_2	8
P_3	7
P_4	3

SJF scheduling chart



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

Determining Length of Next CPU Burst

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

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

Guaranteed Scheduling

Idea: N processes running → each process gets 1/Nth of CPU time
(also known as fair-share)

Solution:

- Calculate how much CPU time it might have gotten:
  - Time since process creation divided by N
- Measure actual consumed CPU time and form ratio

- 0.5 → process running half the time it was entitled to
- 2.0 → process running twice as much as it was entitled to
- Pick process with the smallest ratio to run next
- How to incorporate priorities (See Linux' CFS)?
- Note: fair-share scheduling can be per-user/-process

Lottery Scheduling

- Processes get lottery tickets
- Whenever a scheduling decision is made OS chooses a winning ticket randomly
- Processes can possess multiple tickets $\longrightarrow$ Priorities
- Tickets can be traded between processes.
- Tickets are immediately available to newly created processes.

Policy vs. Mechanism

Important principle

Here: we may have a scheduling algorithm, but parameters to be filled in by user (process)

For instance, to give some child processes higher priority than others

Real Time Systems (1 of 3)

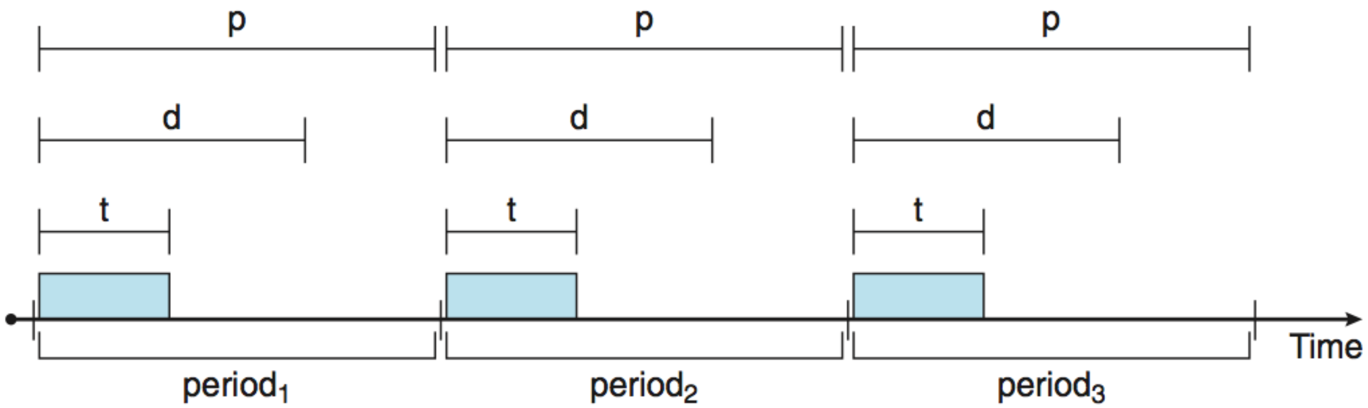
Yauhen_D/Shutterstock

NASA

- Meeting deadlines: avoid losing data
- Predictability: avoid quality degradation in multimedia systems

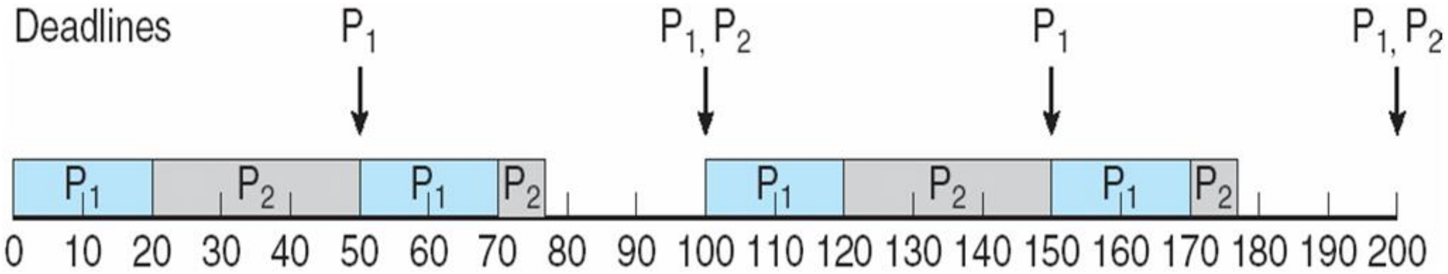
Priority-based Scheduling

- 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 $1/p$

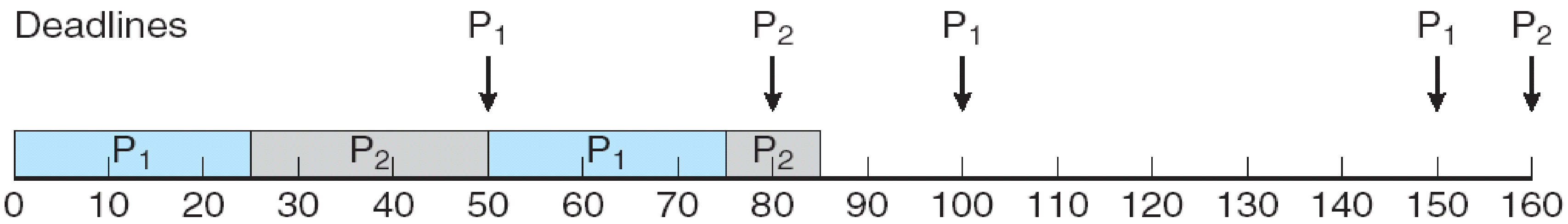


Rate Montonic Scheduling

- *P*1 and *P*2 are 50 and 100, respectively—that is, *p*1 = 50 and *p*2 = 100. The
  - processing times are *t*1 = 20 for *P*1 and *t*2 = 35 for *P*2. The deadline for each
  - process requires that it complete its CPU burst by the start of its next period.



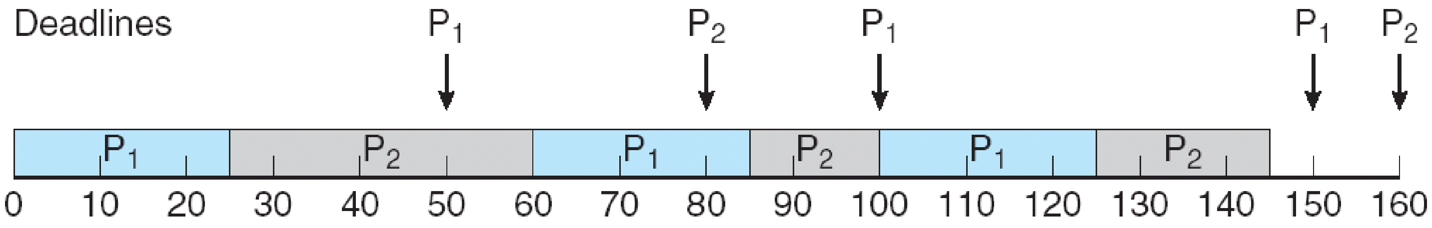
Missed Deadlines with Rate Monotonic Scheduling



Assume that process P_1 has a period of $p_1 = 50$ and a CPU burst of $t_1 = 25$.
For P_2, the corresponding values are $p_2 = 80$ and $t_2 = 35$.

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



Recall that P_1 has values of $p_1 = 50$ and $t_1 = 25$ and that P_2 has values of $p_2 = 80$ and $t_2 = 35$.

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);
}
```

Multiple-Processor Scheduling

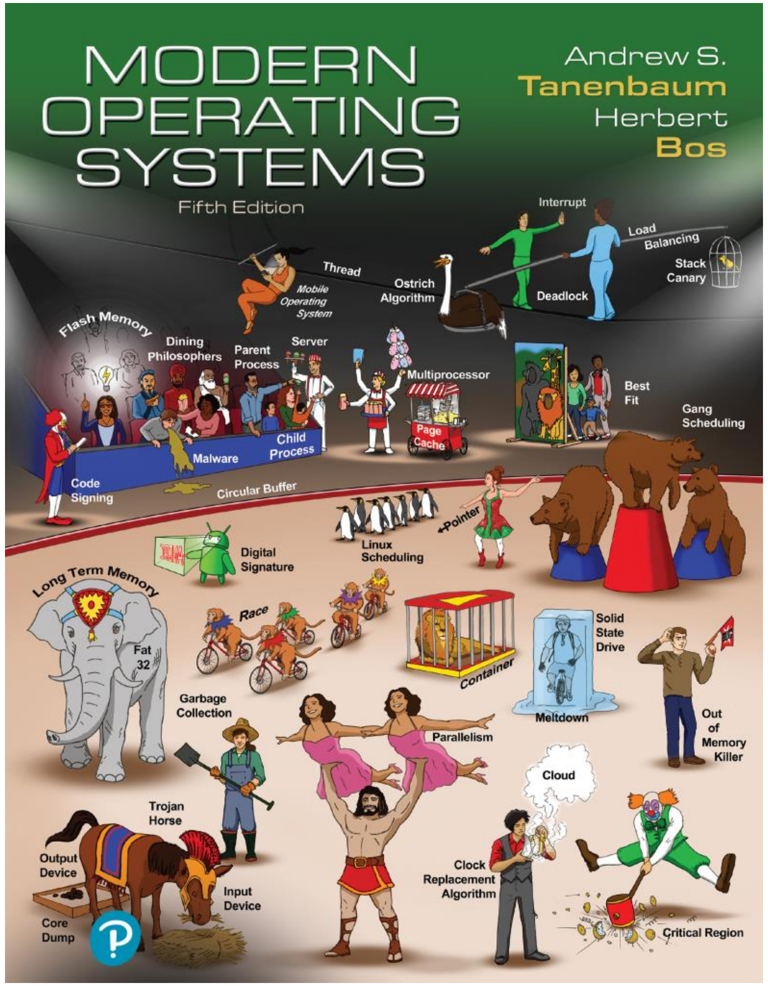
- CPU scheduling more complex when multiple CPUs are available
- **Homogeneous processors** within a multiprocessor
- **Asymmetric multiprocessing** – only one processor accesses the system data structures, alleviating the need for data sharing
- **Symmetric multiprocessing** (**SMP**) – each processor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

Modern Operating Systems

Fifth Edition



Chapter 2

Scheduling - End

Pearson