12. Process Scheduling

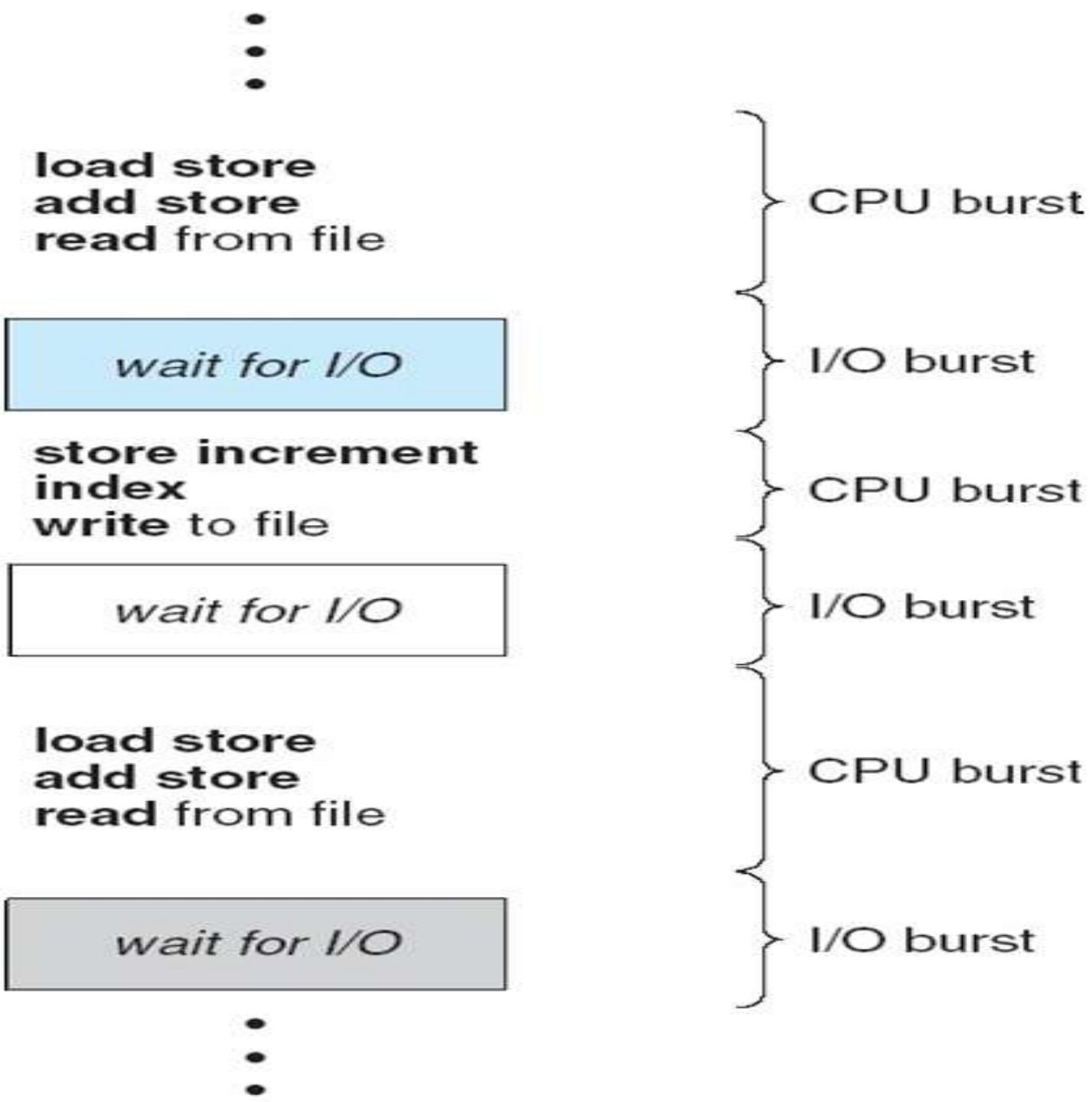
- Basic Concepts
- Scheduling Criteria
- Scheduling Algorithms
- Thread Scheduling
- Multiple-Processor Scheduling
- Operating Systems Examples
- Algorithm Evaluation

A. Scheduling Objectives

- To introduce process scheduling, which is the basis for multi-programmed operating systems
- To describe various process-scheduling algorithms
- To discuss evaluation criteria for selecting a process-scheduling algorithm for a particular system

B. Basic Concepts

- Maximum CPU utilization obtained with multiprogramming
- Process execution time consists of CPU execution time and I/O wait time
  - **CPU burst** distribution
  - CPU – I/O Burst Cycle


load store
add store
read from file
wait for I/O
store increment
index
write to file
wait for I/O
load store
add store
read from file
wait for I/O
CPU burst
I/O burst
CPU burst
I/O burst
CPU burst
I/O burst

C. CPU Scheduler

- Selects from among the processes in memory that are ready to execute, and allocates the CPU to one of them
- 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 under 1 and 4 is **non-preemptive**
- All other scheduling is **preemptive**

Dispatching

- Dispatcher module gives control of the CPU to the process selected
  - 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 execution of another

D. 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)

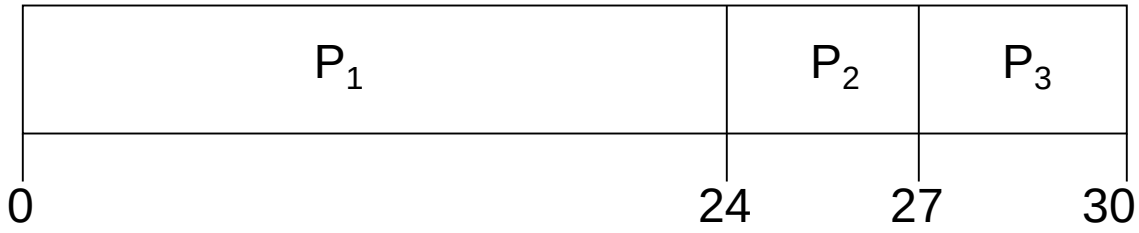
E. Scheduling Algorithm Optimization Criteria

- Max CPU utilization
- Max throughput
- Min turnaround time
- Min waiting time
- Min response time

1. 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:



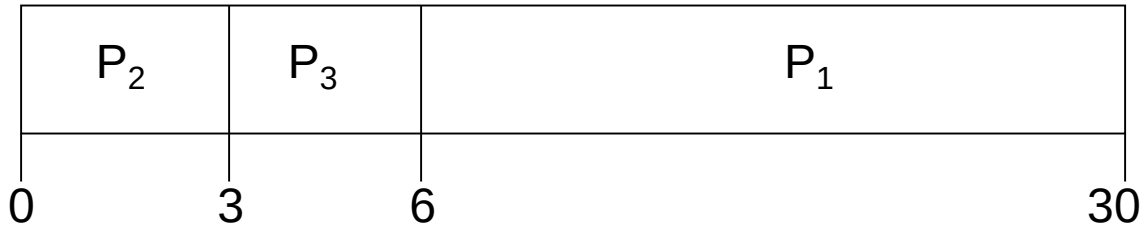
- Waiting time for P_1 = 0; P_2 = 24; P_3 = 27
- Average waiting time: (0 + 24 + 27)/3 = 17

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

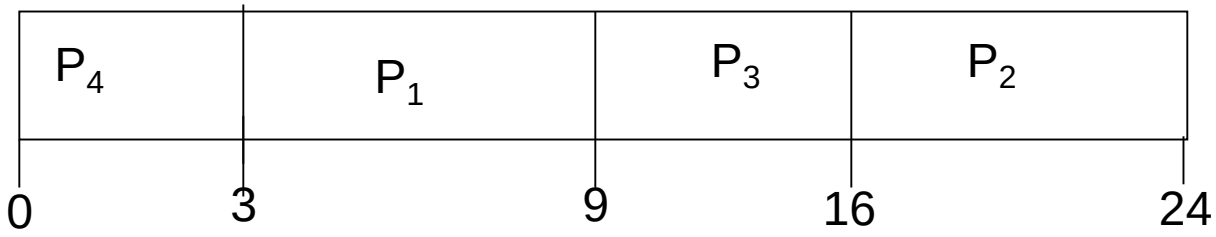
2. Shortest-Job-First (SJF) Scheduling

- 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

Example of SJF

Process	Arrival time	Burst time
P_1	0.0	6
P_2	2.0	8
P_3	4.0	7
P_4	5.0	3

- SJF scheduling chart



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

3. 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
  - Non-preemptive
- SJF is a priority scheduling where priority is the predicted next CPU burst time
- Problem ≡ **Starvation** – low priority processes may never execute
- Solution ≡ **Aging** – as time progresses increase the priority of the process

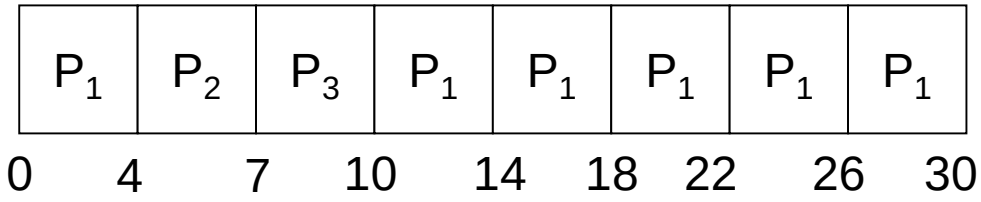
4. Round Robin (RR)

- Each process gets a small unit of CPU time (*time quantum*), 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.
- 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

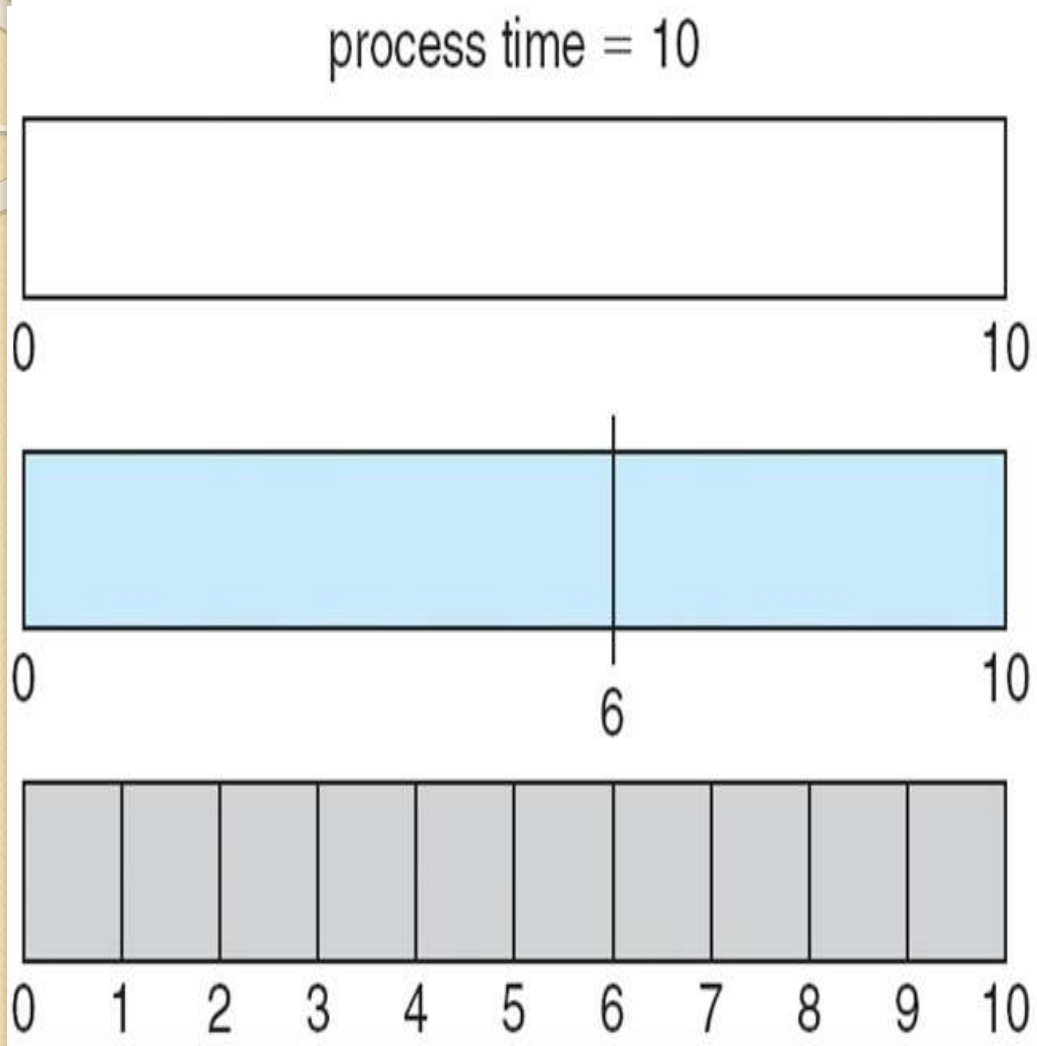
Process	Burst Time
P_1	24
P_2	3
P_3	3

- The Gantt chart is:

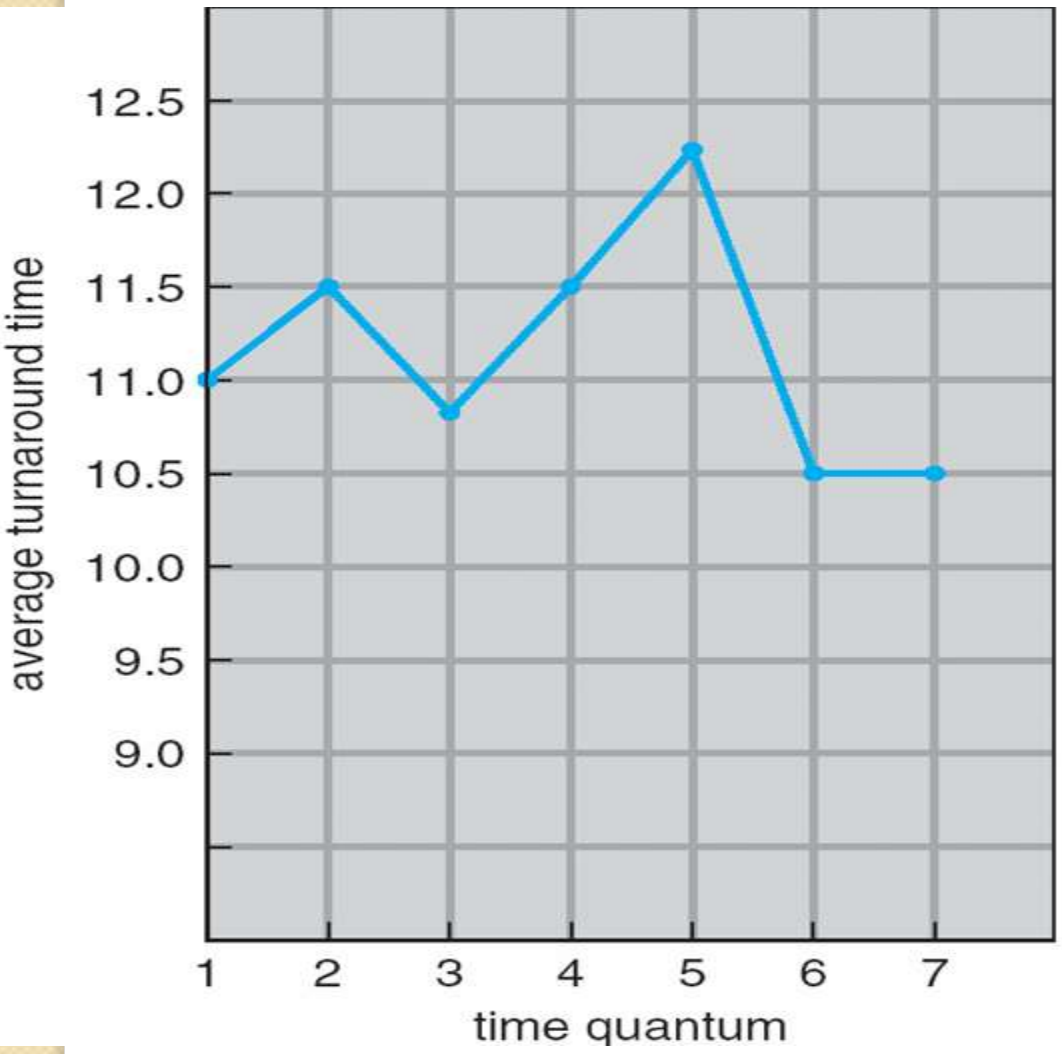


- Typically, higher average turnaround than SJF, but better *response*

Time Quantum and Context Switch Time



Turnaround Time Varies With The Time Quantum

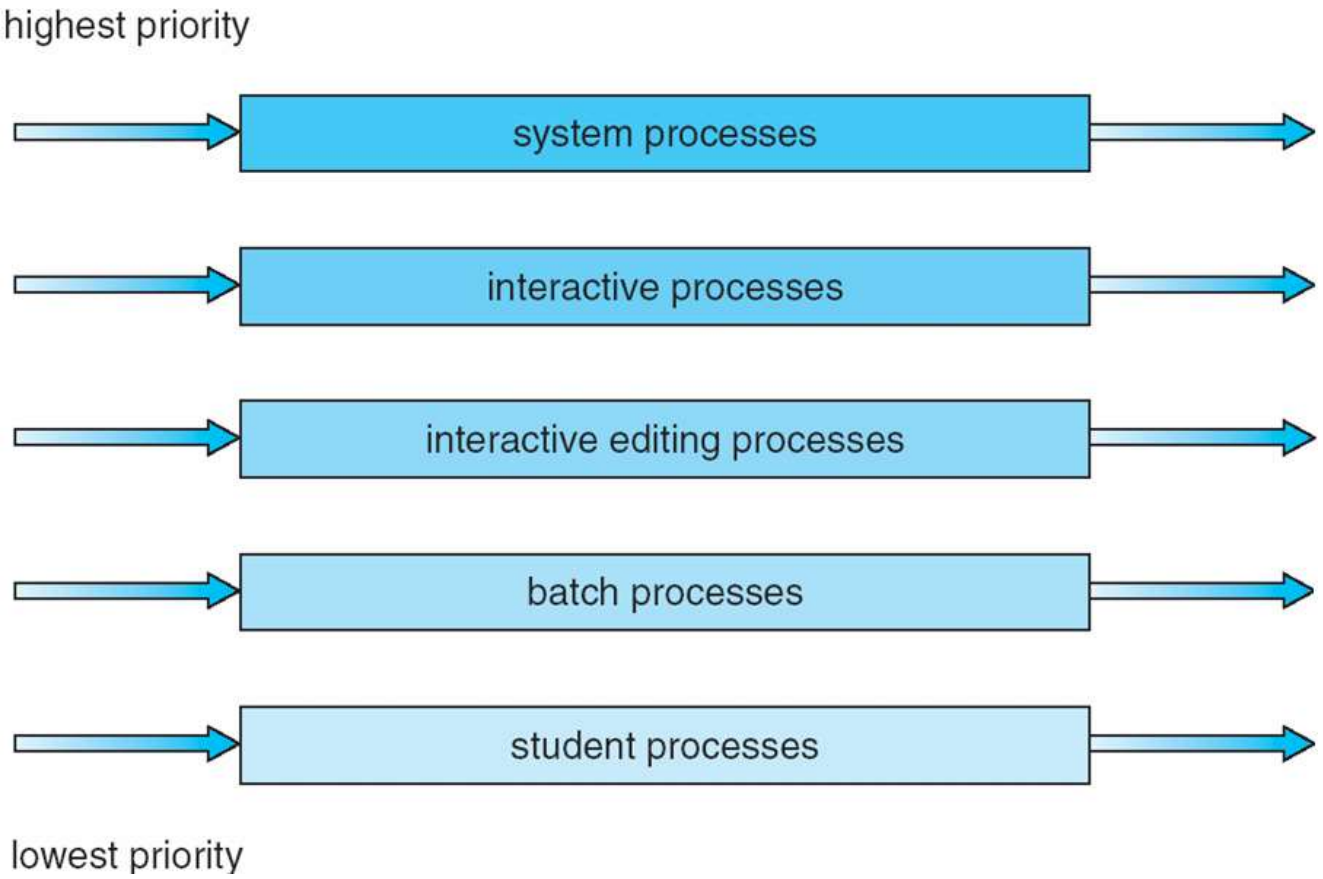


process	time
P_1	6
P_2	3
P_3	1
P_4	7

5. Multilevel Queue

- Ready queue is partitioned into separate queues:
  foreground (interactive)
  background (batch)
- 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



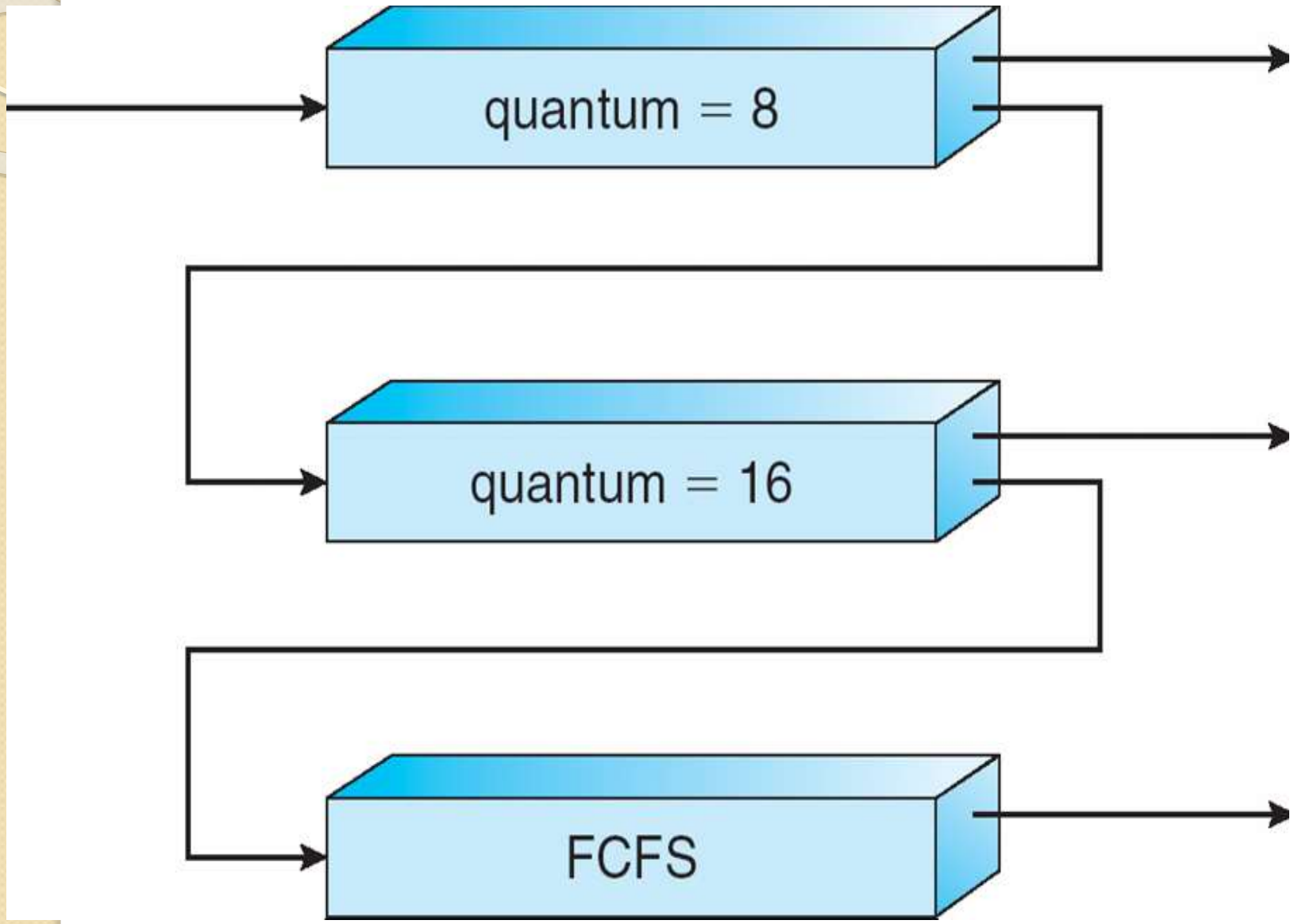
6. Multilevel Feedback Queue

- 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:

- 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.

Multilevel Feedback Queues



Continued……

Unit -2

Threads and Processes

Objectives

- To introduce the notion of a thread — a smallest dispatchable unit of CPU utilization that forms the basis of multithreaded computer systems
- To examine issues related to multithreaded programming

I. Process

- Resource ownership - process is allocated a virtual address space to hold the process image
- Scheduling/execution - follows an execution path that may be interleaved with other processes
- These two characteristics are treated independently by the operating system

Process

- Dispatching is referred to as a thread
- Resource of ownership is referred to as a process or task
- Protected access to processors, other processes, files, and I/O resources

2. Multithreading

- Operating system supports multiple threads of execution within a single process
- MS-DOS supports a single thread
- UNIX supports multiple user processes but only supports one thread per process
- Windows 2000, Solaris, Linux, Mach, and OS/2 support multiple threads

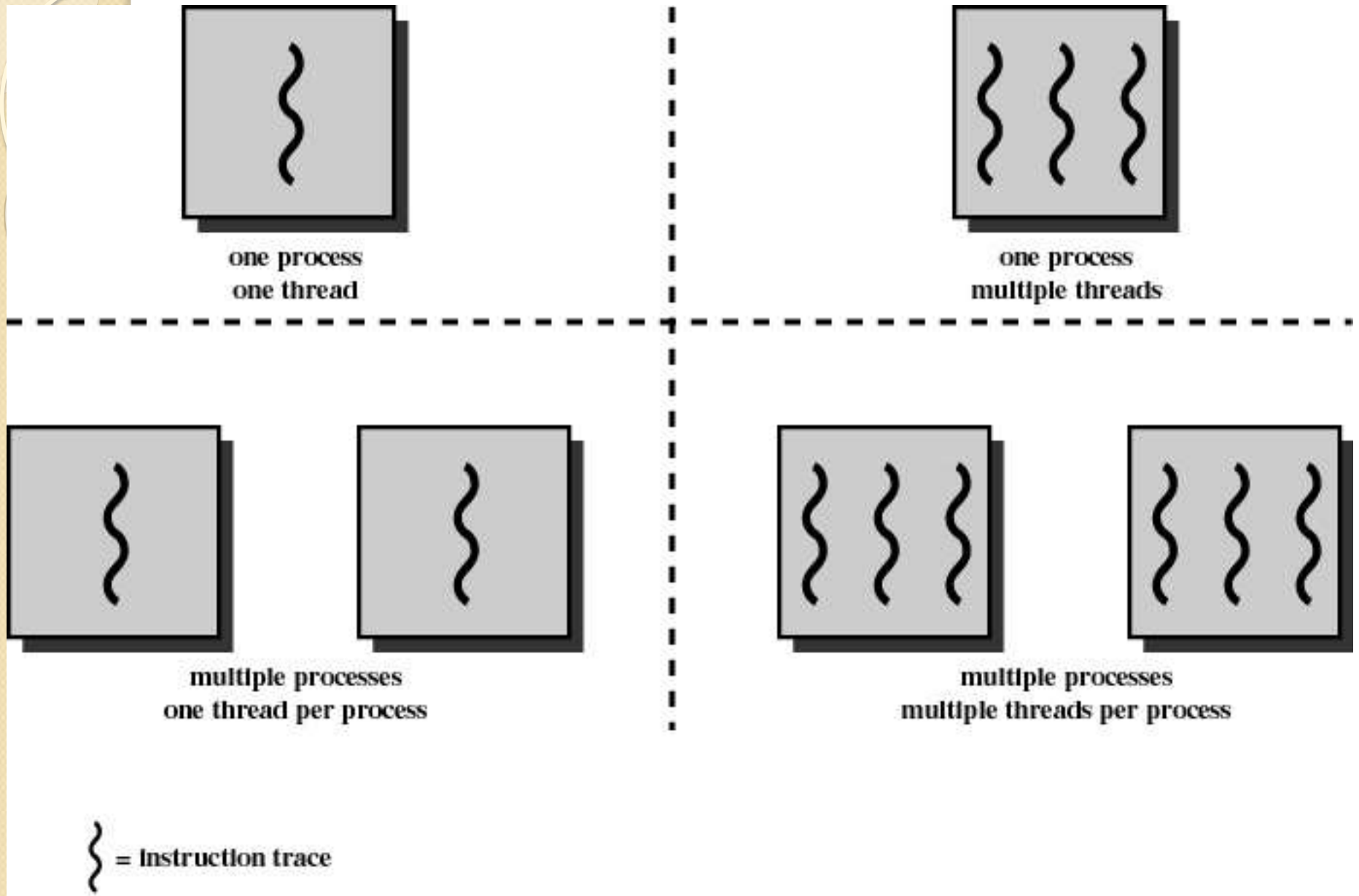
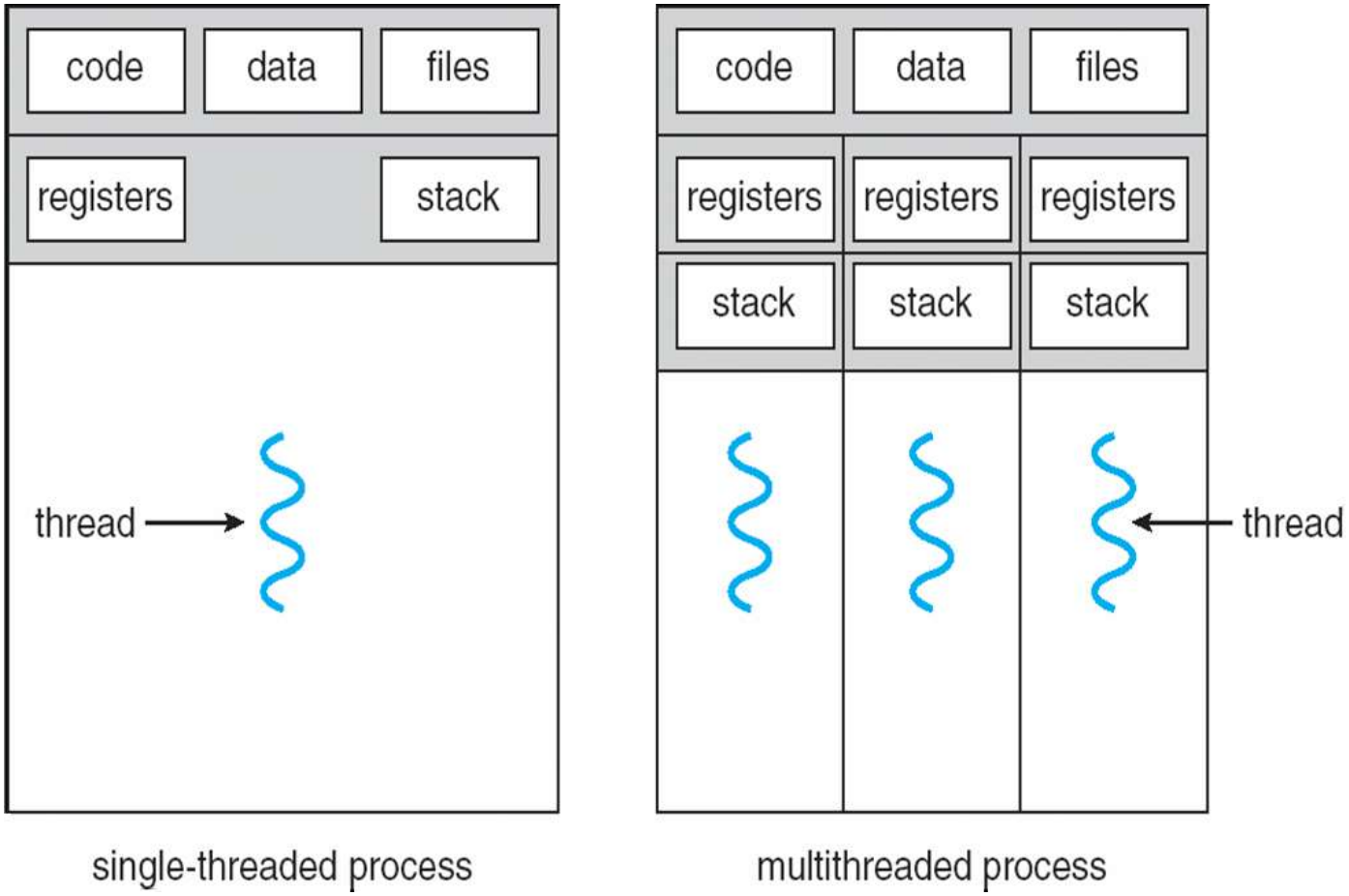


Figure 4.1 Threads and Processes [ANDE97]

Single and Multithreaded Processes



single-threaded process

multithreaded process

3. Thread

- An execution state (running, ready, etc.)
- Saved thread context when not running
- Has an execution stack
- Some per-thread static storage for local variables
- Access to the memory and resources of its process
  - all threads of a process share this

Threads

- Suspending a process involves suspending all threads of the process since all threads share the same address space
- Termination of a process, terminates all threads within the process

Benefits of Threads

- Takes less time to create a new thread than a process
- Less time to terminate a thread than a process
- Less time to switch between two threads within the same process
- Since threads within the same process share memory and files, they can communicate with each other without invoking the kernel

Uses of Threads in a Single-User Multiprocessing System

- Foreground to background work
- Asynchronous processing
- Speed execution
- Modular program structure

4. Thread States

- States associated with a change in thread state
  - Spawn
    - Issue another thread
  - Block
  - Unblock
  - Finish
    - Deallocate register context and stacks

5. Levels of Threads

5.1 User-Level Threads

- All thread management is done by the application
- The kernel is not aware of the existence of threads

5.2 Kernel-Level Threads

- W2K, Linux, and OS/2 are examples of this approach
- Kernel maintains context information for the process and the threads
- Scheduling is done on a thread basis

5.3 Combined Approaches

- Example is Solaris
- Thread creation done in the user space
- Bulk of scheduling and synchronization of threads done in the user space

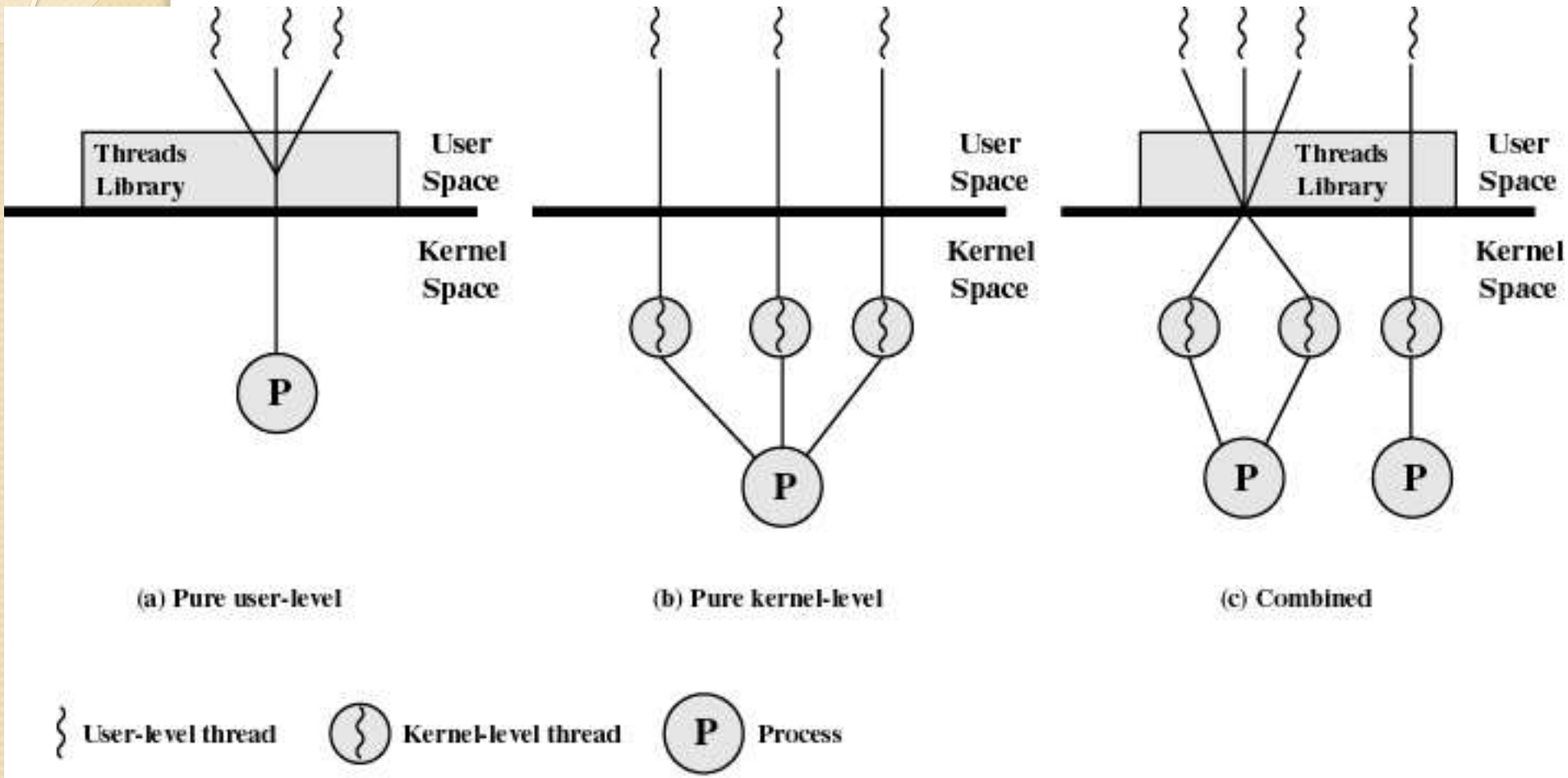


Figure 4.6 User-Level and Kernel-Level Threads

Relationship Between Threads and Processes

Threads:Process	Description	Example Systems
1:1	Each thread of execution is a unique process with its own address space and resources.	Traditional UNIX implementations
M:1	A process defines an address space and dynamic resource ownership. Multiple threads may be created and executed within that process.	Windows NT, Solaris, OS/2, OS/390, MACH

Relationship Between Threads and Processes

Threads:Process	Description	Example Systems
1:M	A thread may migrate from one process environment to another. This allows a thread to be easily moved among distinct systems.	Ra (Clouds), Emerald
M:M	Combines attributes of M:1 and 1:M cases	TRIX

Unit 3

Process Concurrency

(Inter-process Communication)

Mutual Exclusion

and

Synchronization

Outline

- Principles of Concurrency
- Mutual Exclusion : Hardware Support
- Semaphores
- Monitors
- Message Passing
- Readers/Writers Problem

I. Multiple Processes

- Central to the design of modern Operating Systems is managing multiple processes
  - Multiprogramming
  - Multiprocessing
  - Distributed Processing
- Big Issue is Concurrency
  - Managing the interaction of all of these processes

2. Concurrency

Concurrency arises in:

- Multiple applications
  - Sharing time
- Structured applications
  - Extension of modular design
- Operating system structure
  - OS themselves implemented as a set of processes or threads

Interleaving and Overlapping Processes

- Earlier we saw that processes may be interleaved on uniprocessors

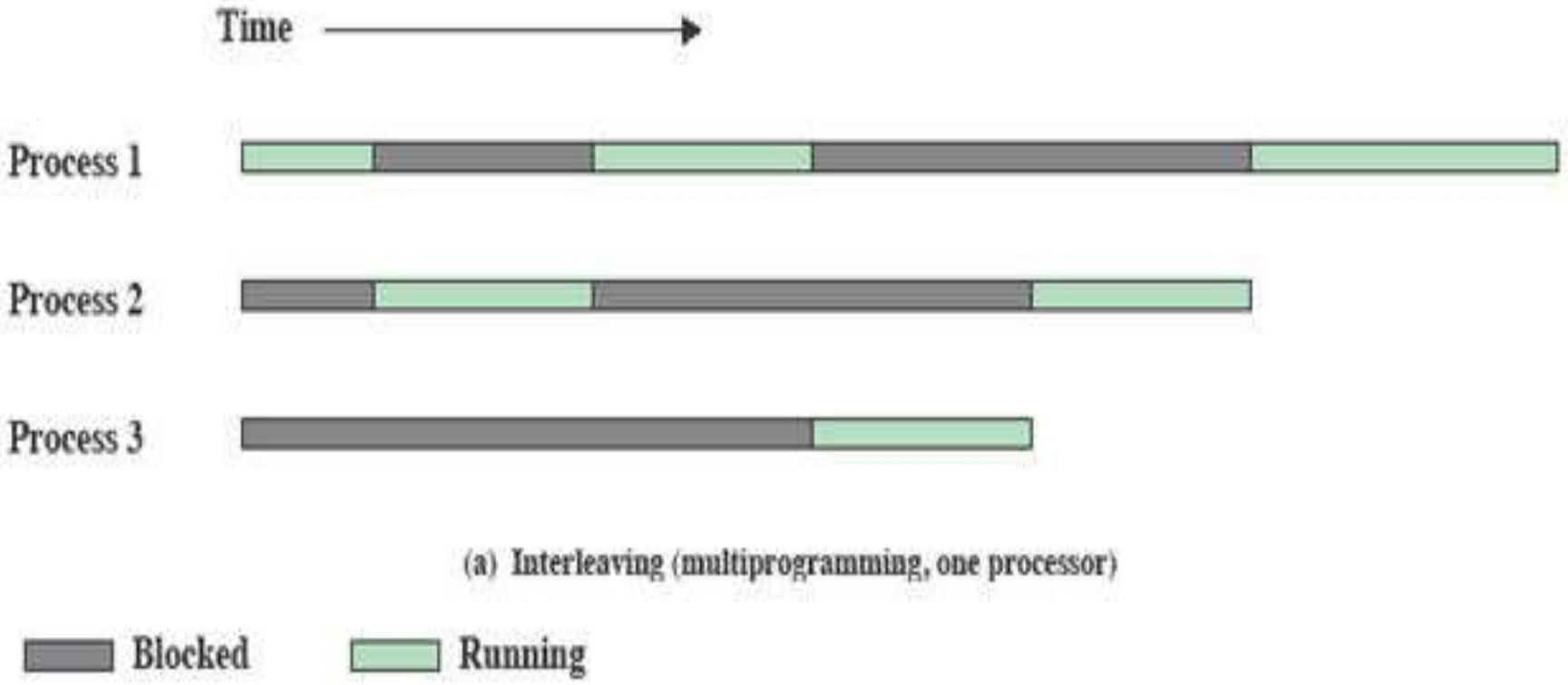


Figure 2.12 Multiprogramming and Multiprocessing

Interleaving and Overlapping Processes

- And not only interleaved but overlapped on multi-processors

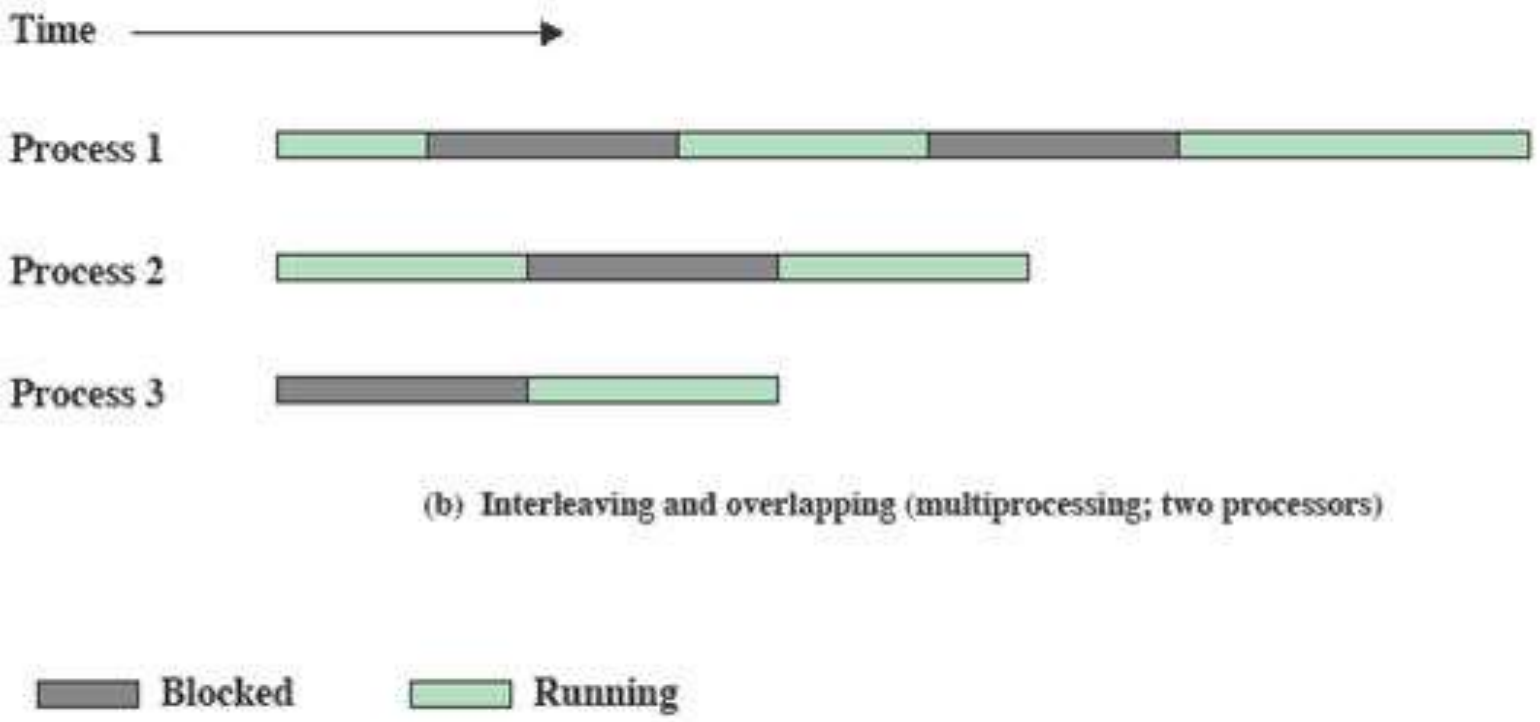


Figure 2.12 Multiprogramming and Multiprocessing

3. Difficulties of Concurrency

- Sharing of global resources (maintain the consistency)
- Optimally managing the allocation of resources (resource blocked)
- Difficult to locate programming errors (running infinite loop)

A Simple Example : Concurrency

```
void echo()
{
   chin = getchar();
   chout = chin;
   putchar(chout);
}
```

An example : Call of a function by two Processes

Process P1

```
{
.
.
echo(); // critical section
.
.
}
```

Process P2

```
{
.
.
echo();  // critical section
.
.
}
```

An example : On a Multiprocessor system

Process P1

```
        .
chin = getchar();
        .
chout = chin;
putchar(chout);
        .
        .
```

Process P2

```
        .
        .
chin = getchar();
chout = chin;
        .
putchar(chout);
        .
```

Solution: Enforce Single Access

- If we enforce a rule that only one process may enter the function at a time then :

Scenario

- P1 & P2 run on separate processors
- P1 enters echo first,
  - P2 tries to enter but is blocked – P2 suspends
- P1 completes execution
  - P2 resumes and executes echo

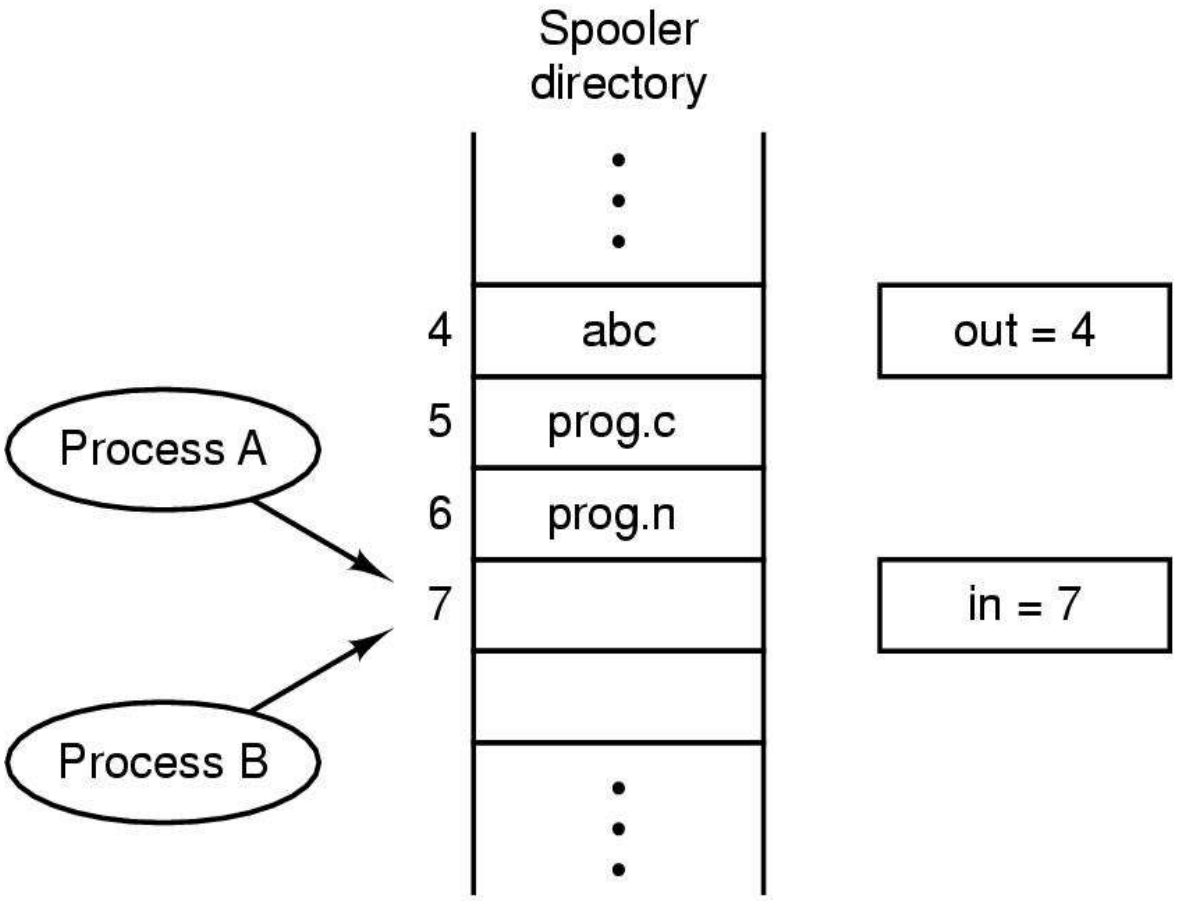
Race Condition

- A race condition occurs when
  - Multiple processes or threads read and write data items (Global resources)
  - Final result depends on the order of execution of the processes.
- The output depends on who finishes the race last.

IPC: Race Condition

Race Condition

The situation where 2 or more processes are reading or writing some shared data is called race condition



Two processes want to access shared memory at same time

4. Operating System Concerns

- What design and management issues are raised by the existence of concurrency?
- The OS must
  - Keep track of various processes
  - Allocate and de-allocate resources
  - Protect the data and resources against interference by other processes.
  - Ensure that the processes and outputs are independent of the processing speed

Process Interaction

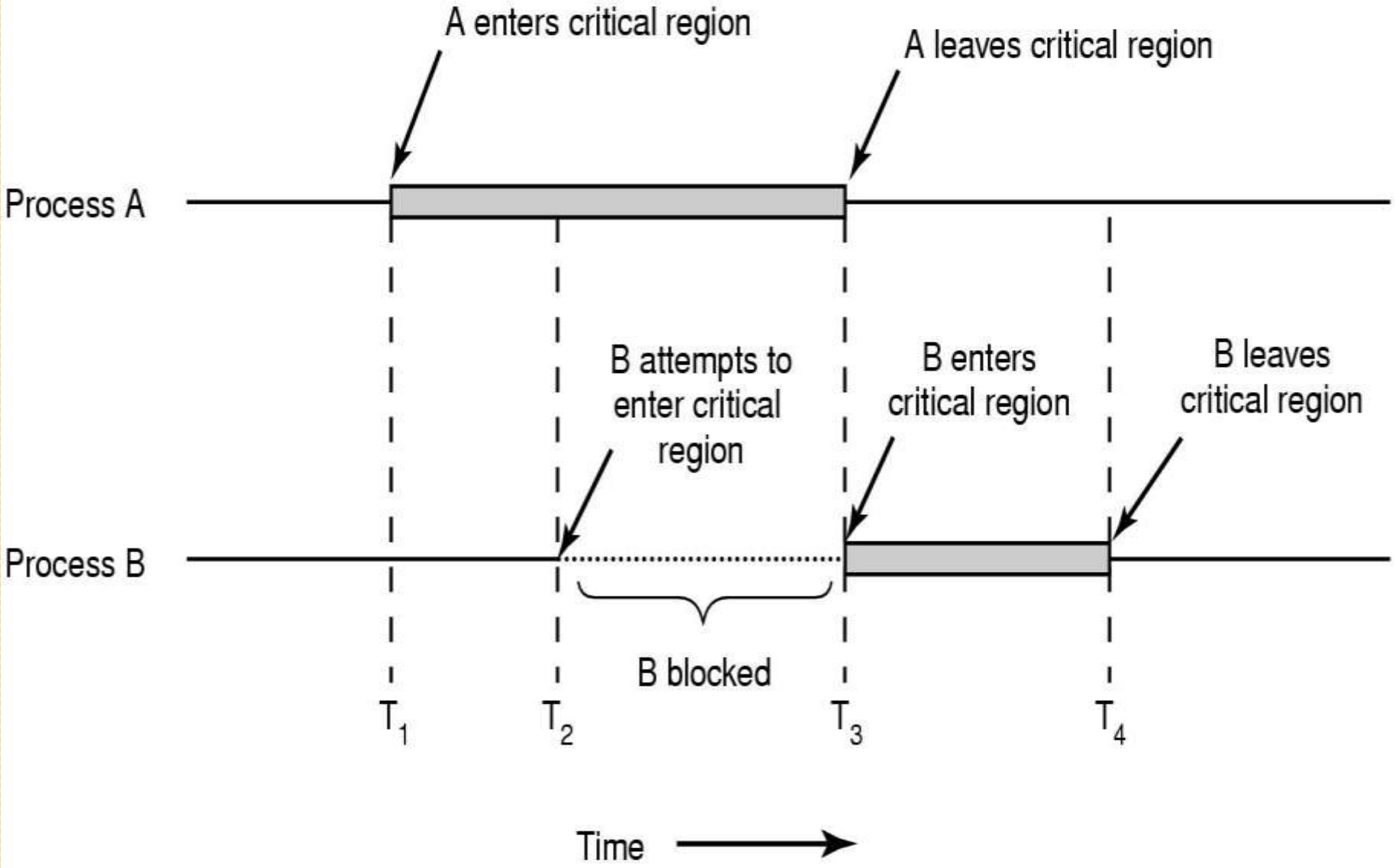
Table 5.2 Process Interaction

Degree of Awareness	Relationship	Influence That One Process Has on the Other	Potential Control Problems
Processes unaware of each other	Competition	• Results of one process independent of the action of others • Timing of process may be affected	• Mutual exclusion • Deadlock (renewable resource) • Starvation
Processes indirectly aware of each other (e.g., shared object)	Cooperation by sharing	• Results of one process may depend on information obtained from others • Timing of process may be affected	• Mutual exclusion • Deadlock (renewable resource) • Starvation • Data coherence
Processes directly aware of each other (have communication primitives available to them)	Cooperation by communication	• Results of one process may depend on information obtained from others • Timing of process may be affected	• Deadlock (consumable resource) • Starvation

5. Mutual Exclusion : Requirements

- Only one process at a time is allowed in the critical section for a resource
- A process that executes in its noncritical section must not interfere with other processes
- No deadlock or starvation
- A process must not be delayed access to a critical section when there is no other process using it
- No assumptions are made about relative process speeds or number of processes
- A process remains inside its critical section for a finite time only

Mutual exclusion using Critical Regions



Outline

- Principles of Concurrency
- Mutual Exclusion: Hardware Support
- Semaphores
- Monitors
- Message Passing
- Readers/Writers Problem

1. Disabling Interrupts

- Uniprocessors only allow interleaving
- Interrupt Disabling
  - A process runs until it invokes an operating system service or until it is interrupted
  - Disabling interrupts guarantees mutual exclusion
  - Will not work in multiprocessor architecture

Pseudo-Code

```
while (true)
{
  /* disable interrupts */;
  /* critical section */;
  /* enable interrupts */;
  /* remainder */;
}
```

Synchronization Hardware : Problems

- Many systems provide hardware support for critical section code
- Uniprocessor – could disable interrupts
  - Currently running code would execute without preemption
  - Not supporting in multiprogramming environment
- Multiprocessors -
  - Generally too inefficient on multiprocessor systems
  - Operating systems using this not broadly scalable

Machine Instructions

- Modern machines provide special atomic hardware instructions
  - Atomic = non-interruptable
  - **Either test memory word and set value**
  - **Or Swap contents of two memory words**

Mutual Exclusion: Hardware Support

- Test and Set Instruction

```
boolean TestAndSet (int lock)
{
        if (lock == 0)
        {
                lock = 1;
                return true;
        }
        else
        {
                return false;
        }
}
```

Mutual Exclusion: Hardware Support

- Exchange Instruction

```
void Swap(int register,
          int memory)
{
   int temp;
   temp = memory;
   memory = register;
   register = temp;
}
```

Solution using TestAndSet

- Shared boolean variable lock., initialized to false.
- Solution:

```
boolean TestAndSet (int lock) {
if (lock == 0)
{
        lock = 1;
        return true;
}
else
{
return false;
}
}
```

Process - 1

```
do {
        while ( TestAndSet (&lock ))
              ;  // do nothing

              //    critical section

        lock = FALSE;

              //      remainder section

    } while (TRUE);
```

Process - 2

```
do {
        while ( TestAndSet (&lock ))
              ;  // do nothing

              //    critical section

        lock = FALSE;

              //      remainder section

    } while (TRUE);
```

Solution using Swap

- Method:

1. Shared Boolean variable lock initialized to FALSE;
2. Each process has a local Boolean variable key

```
 void Swap(int register,
int memory)
{
int temp;
temp = memory;
memory = register;
register = temp;
}
```

Solution:

```
Process -I
do {
        key = TRUE;
         while ( key == TRUE && lock == FALSE)
               Swap (&lock, &key );

               //    critical section

               lock = FALSE;

              //      remainder section

          } while (TRUE);
```

Mutual Exclusion Machine Instructions

- Advantages
  - Applicable to any number of processes on either a single processor or multiple processors sharing main memory
  - It is simple and therefore easy to verify
  - It can be used to support multiple critical sections

Mutual Exclusion Machine Instructions

- Disadvantages
  - Busy-waiting consumes processor time
  - Starvation is possible when a process leaves a critical section and more than one processes are waiting.
  - Deadlock
    - If a low priority process has the critical region and a higher priority process needs, the higher priority process will obtain the processor to wait for the critical region

Outline

- Principals of Concurrency
- Mutual Exclusion: Hardware Support
- Semaphores
- Monitors
- Message Passing
- Readers/Writers Problem