



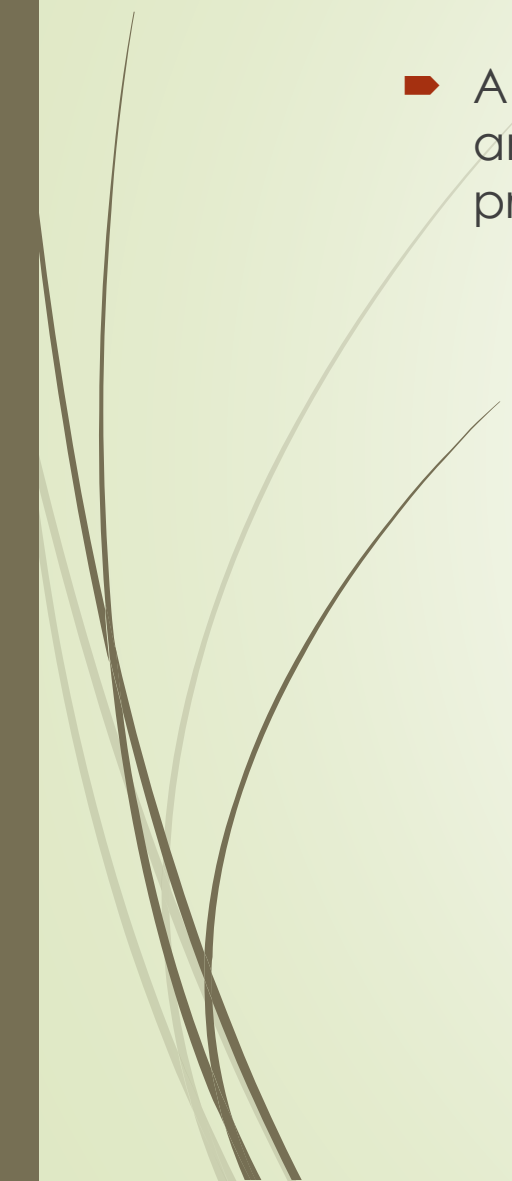


# **Unit 2 – Process Management**

- 
- 
- Introduction
  - Threads
  - IPC
  - Implementing Mutual Exclusion
  - Classical IPC Problems
  - Process Scheduling (Numerical)



# Process

- A process is basically a program in execution. In computing , a process is an instance of a computer program that is being executed. It contains program code and activity.
- 



# Process vs Program

## Process

- Process is activity.
- High resource requirements.
- Has its own control block called PCB.
- Process exists temporarily, it gets terminated on the completion of tasks.
- Dynamic Entity

## Program

- Program is the group of instructions.
- No resource requirements only memory.
- No control block
- Program exists at single place and continues to exist there until deleted.
- Static Entity



# Process vs Program (contd.)

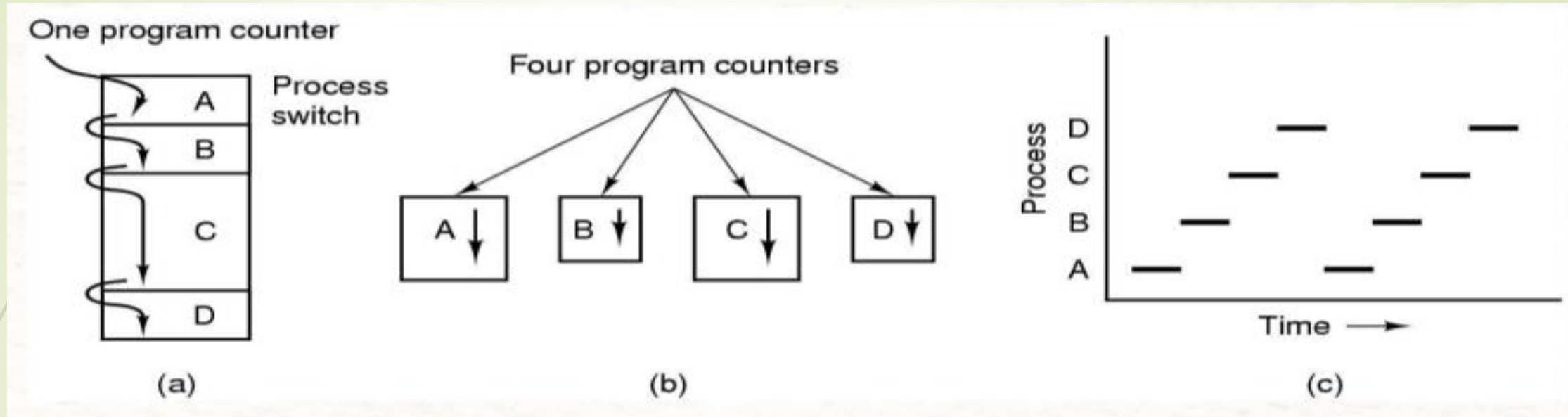
**Example:** Mom is cooking pasta for dinner.

- ➡ Mom: CPU
- ➡ Recipe: **Program** (method to cook [algorithm])
- ➡ Pasta Ingredients: Input Data

Activities involved are **processes**

- Reading the recipe
- Collecting the ingredients
- Cooking Pasta


# Process Models



- Multiprogramming of four programs in memory (a)
- Conceptual model of four independent, sequential process, each with its own flow of control and each one running independently (b)
- Only one process running at a time (c)

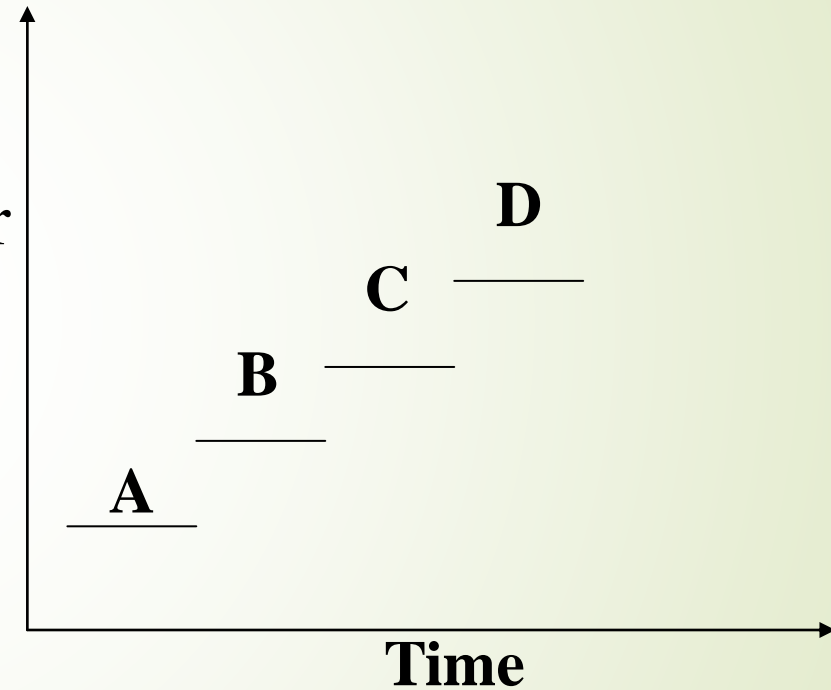


## Process Models (contd.)

- Uni Programming
  - Multi Programming
  - Multi Processing
- 

# Uni Programming

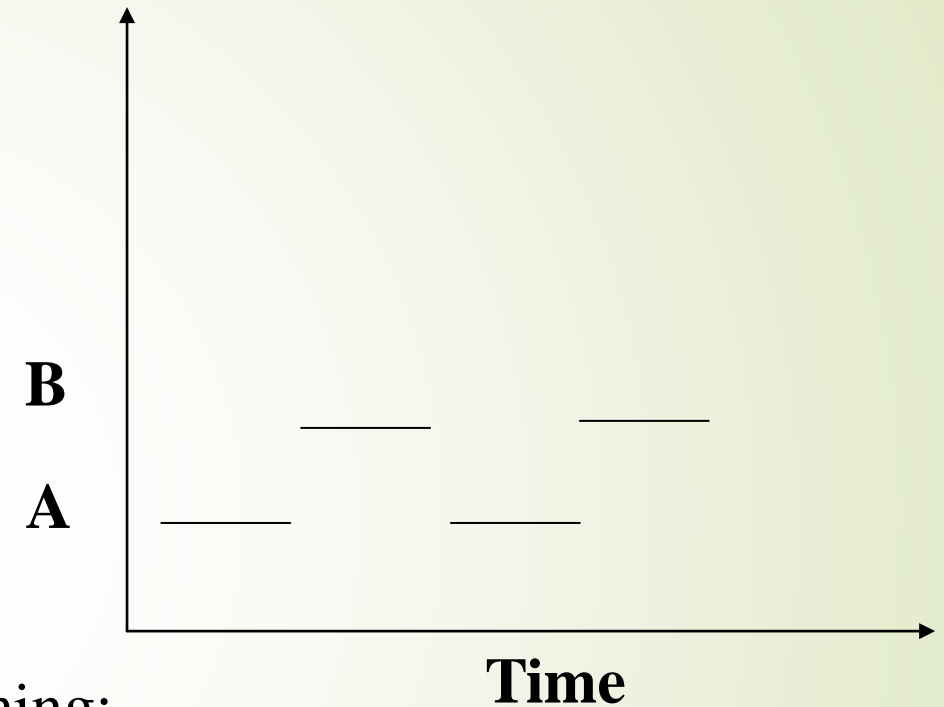
- Only one process at a time.
- Pons: Easier for OS designer
- Cons: Poor Performance





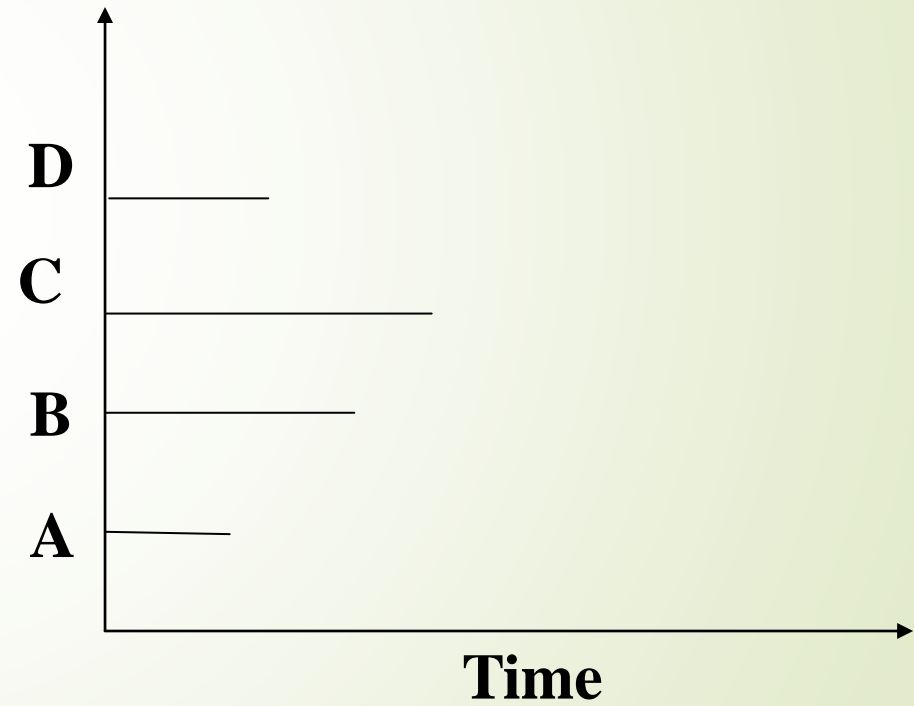
# Multi Programming

- Multiple processes at a time
- Pons: Better system performance
- Cons: Complexity in OS
- OS requirements for multiprogramming:
  - Policy: to determine which process is to schedule.
  - Mechanism: to switch between the processes.



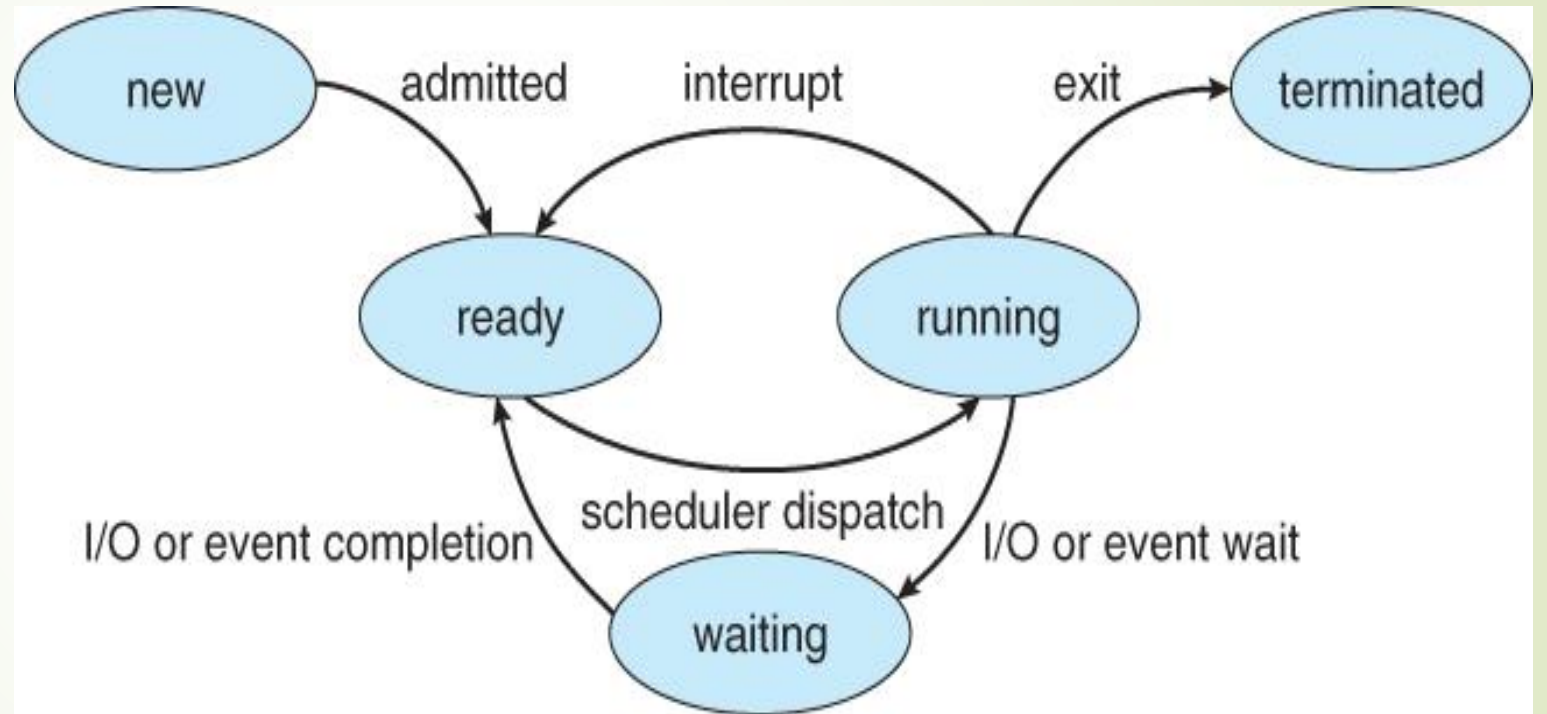
# Multi Processing

- System with multiple processors
- Multiprocessing system can have
  - Multiprocessor
  - Multi-core processor



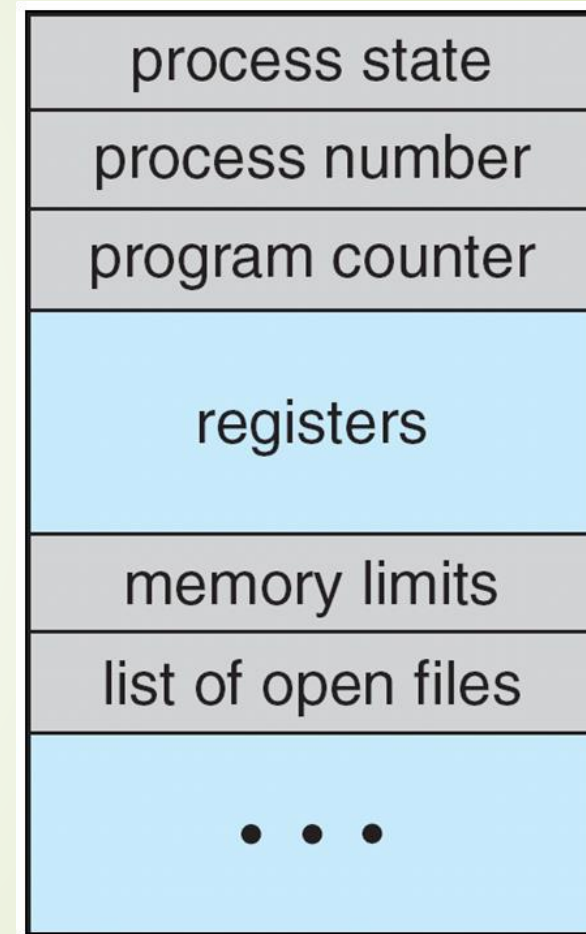
# Process States

- New
- Ready
- Running
- Waiting
- Terminated

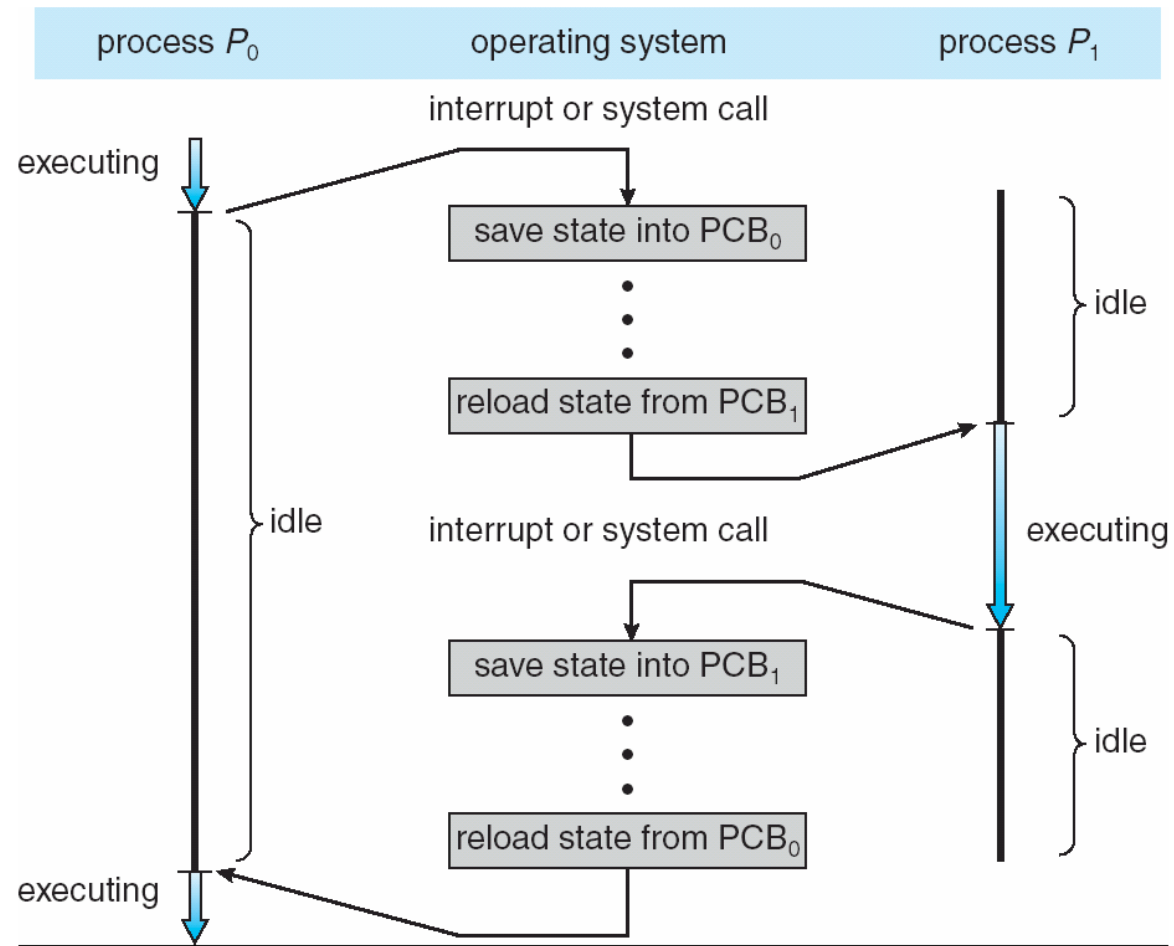


# Process Control Block / Process Table

- Information associated with each process (also called **task control block**)
- Process state – running, waiting, etc
- Program counter – location of instruction to next execute
- CPU registers – contents of all process-centric registers
- CPU scheduling information- priorities, scheduling queue pointers
- Memory-management information – memory allocated to the process
- Accounting information – CPU used, clock time elapsed since start, time limits
- I/O status information – I/O devices allocated to process, list of open files



# PCB (contd.)





# Threads

- Threads are like Process.
- Allows more than one job to be performed at a time.
- Thread – Lightweight Process whereas Process - Heavyweight Process.
- Many applications now-a-days are multithreaded
  - Web Browser
    - A Thread for retrieving data from network
    - Another Thread for displaying text and images
  - Word Processor
    - A Thread for displaying graphics
    - Another thread for reading keystrokes from user
    - Third Thread for performing grammar checks in background




# Thread vs Process

## Thread

- Thread is light weight, takes less resources than process
- Thread switching doesn't need interaction with OS
- All threads can share same file, memory and child processes
- Multiple threaded processes use fewer resources
- One thread can R-W or change another thread data
- Blocking a thread will block the entire process.

## Process

- Process is heavy weight or resource intensive
- Process switching needs interaction with OS
- In multi processing, each process executes same code but has its own file processes and memory
- Multiple processes without threads use more resources
- Processes are independent of one another
- Blocking a process will not block the another process.



# Types of Thread

- User-Level Thread
  - Kernel-Level Thread
- 



# User-Level Thread

- Management done by user-level threads library
- Implemented as a library
  - Library provides support for thread creation, scheduling and management with no support from the kernel.
- Three primary thread libraries:
  - POSIX Pthreads
  - Windows threads
  - Java threads
- Pons:
  - Thread switching doesn't require kernel mode privileges
  - Can run on any OS
  - Fast to create and manage
- Cons:
  - Multithreaded applications can't take advantage of multiprocessing

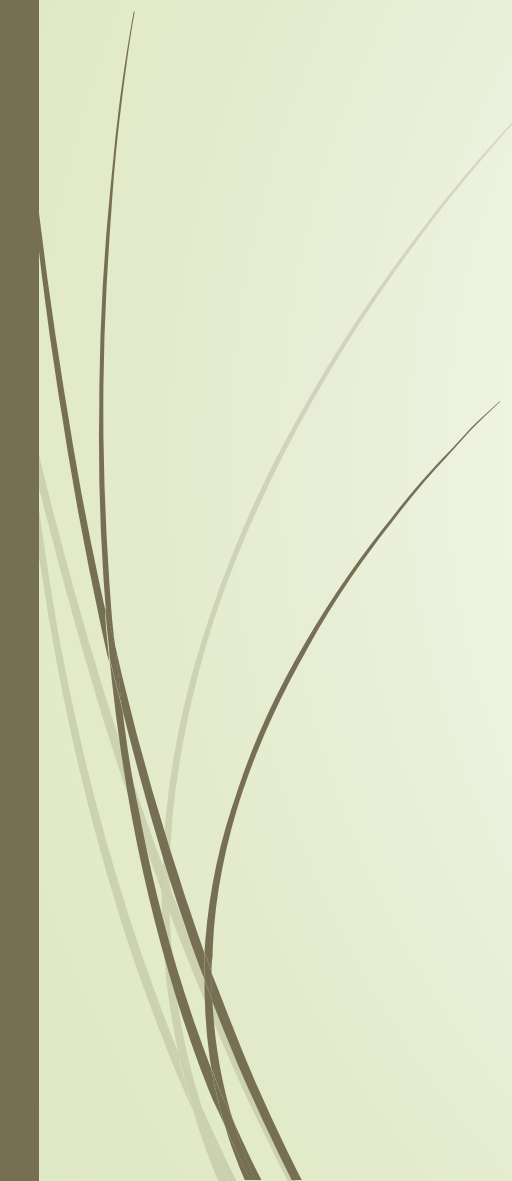


# Kernel Level Threads

- Supported by kernel
- Kernel performs thread creation, scheduling and management in kernel space.
- Slower to create and manage
- Blocking system calls are no problem
- Pons:
  - Can schedule multiple threads from same process on different processes
  - If one thread is blocked then kernel schedules another thread on the same process
- Cons:
  - Slower to create and manage than user thread
  - Specific to OS



# Multithreading Model

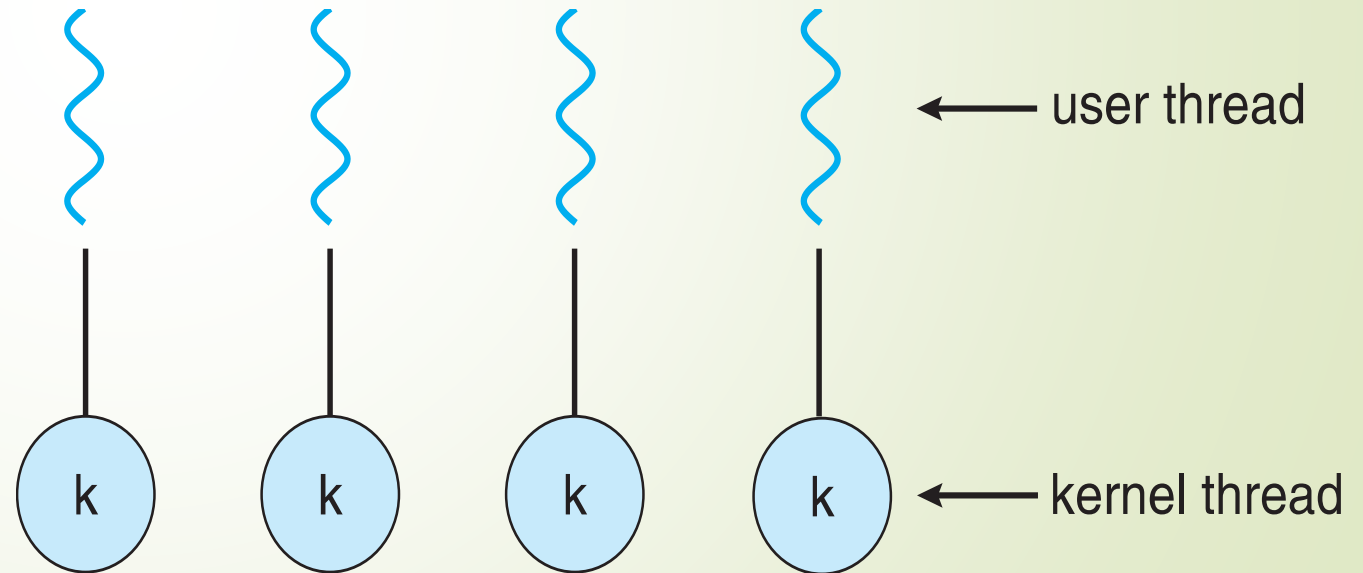
- One-to-One Model
  - Many-to-One Model
  - Many-to-Many Model
- 

# One-to-One Model

- Each user-level thread maps to kernel thread
- Creating a user-level thread creates a kernel thread
- More concurrency than many-to-one
- Number of threads per process sometimes restricted due to overhead

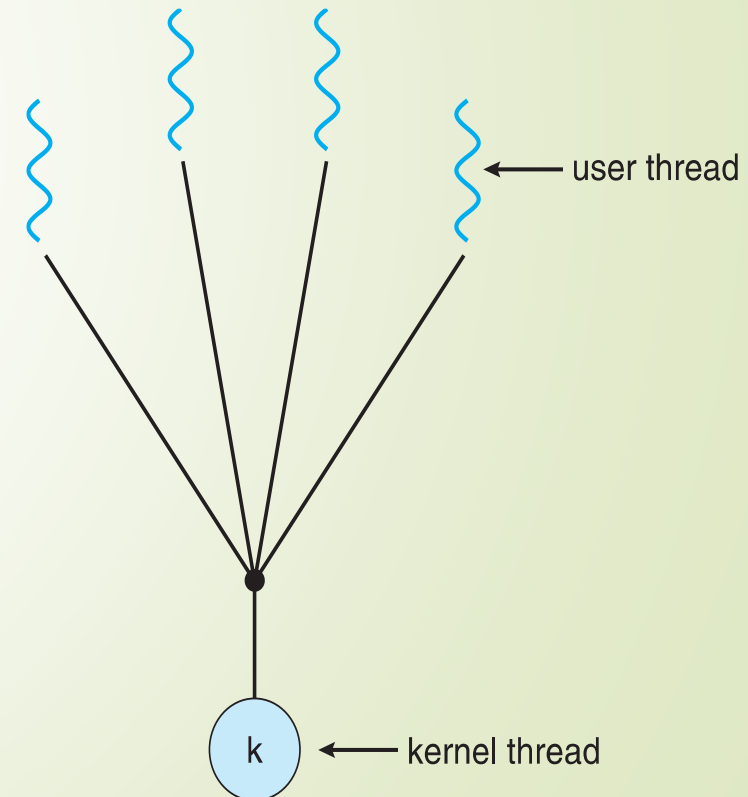
- Examples

- Windows
- Linux
- Solaris 9 and later



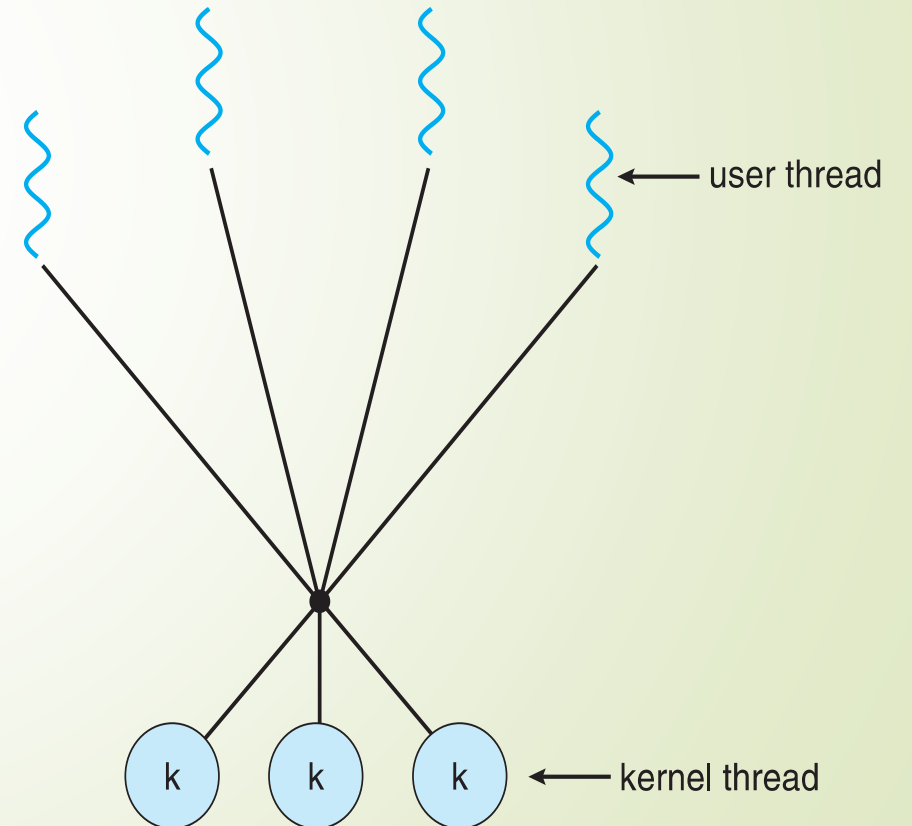
# Many-to-One Model

- Many user-level threads mapped to single kernel thread
- One thread blocking causes all to block
- Few systems currently use this model
- Examples:
  - **Solaris Green Threads**
  - **GNU Portable Threads**



# Many-to-Many Model


- Allows many user level threads to be mapped to many kernel threads
- Allows the operating system to create a sufficient number of kernel threads
- Solaris prior to version 9
- Windows with the *Thread Fiber* package





# Interposes Communication(IPC)





**Independent processes** - They cannot affect or be affected by the other processes executing in the system.

**Cooperating processes** - They can affect or be affected by the other processes executing in the system.

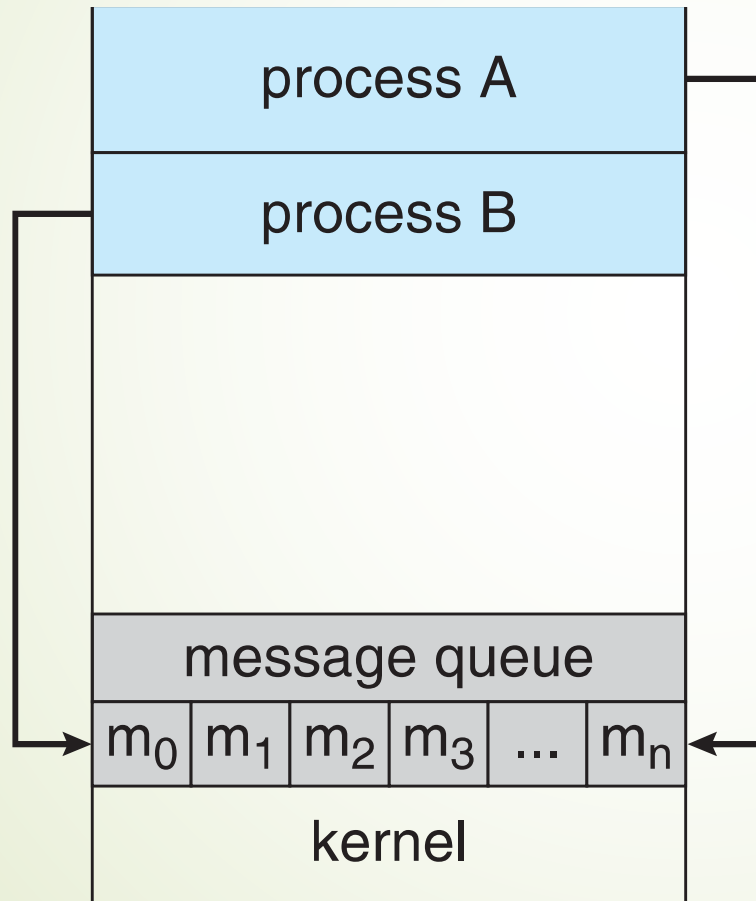


# Inter-Process Communication

- A mechanism which allows process to communicate with each other and synchronize their actions.
- Processes within a system may be *independent* or *cooperating*
- Cooperating process can affect or be affected by other processes, including sharing data
- Reasons for cooperating processes:
  - Information sharing
  - Computation speedup
  - Modularity
  - Convenience
- Two models of IPC
  - Shared memory
  - Message passing

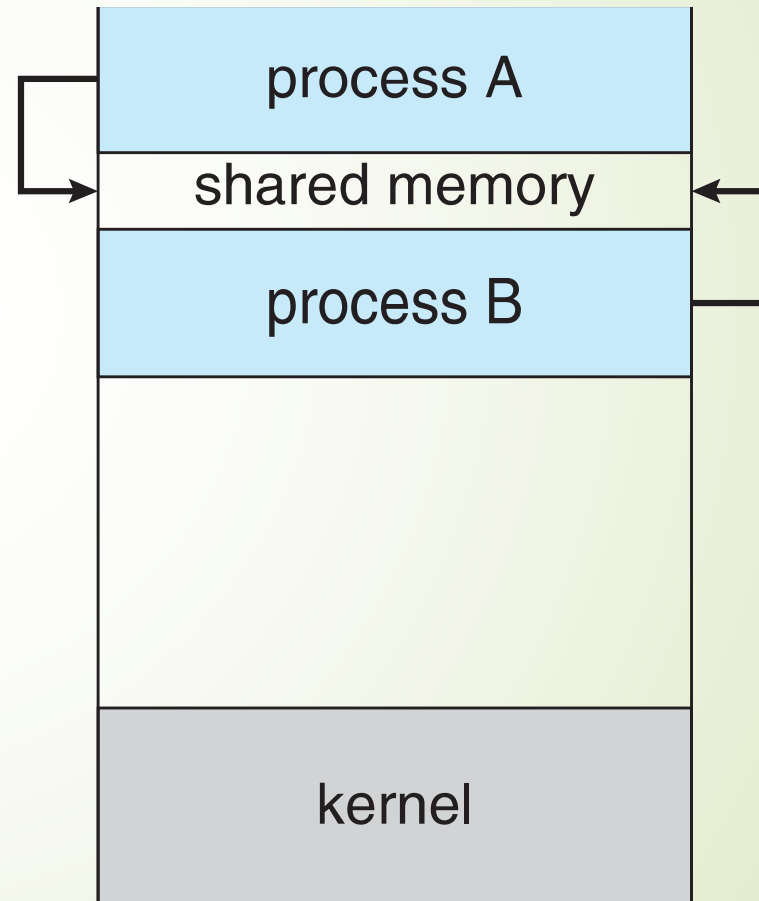
# IPC (contd.)

a. Message Passing



(a)

b. Shared Memory



(b)

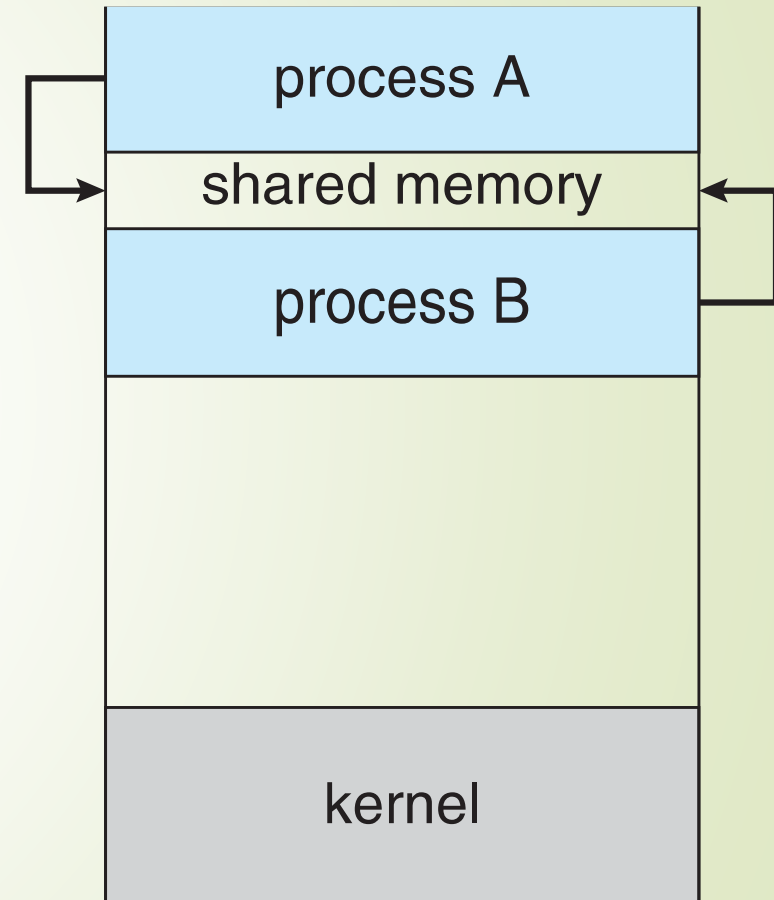


# Independent-Cooperating Process

- *Independent* process cannot affect or be affected by the execution of another process
- *Cooperating* process can affect or be affected by the execution of another process
- Advantages of process cooperation
  - Information sharing
  - Computation speed-up
  - Modularity
  - Convenience

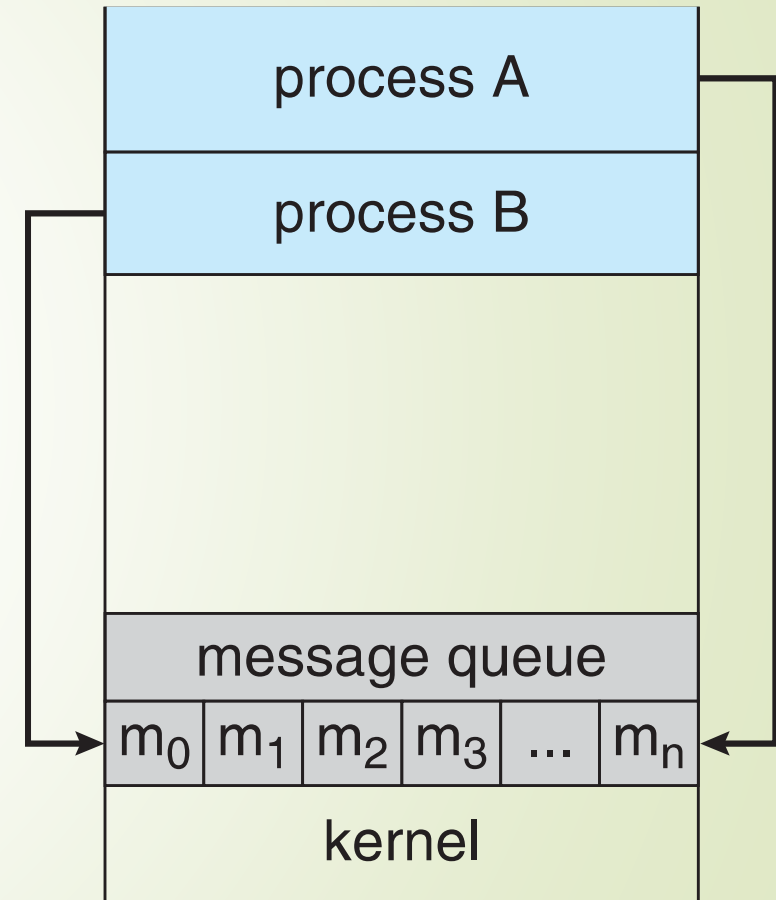
# IPC-Shared Memory

- An area of memory shared among the processes that wish to communicate
- The communication is under the control of the users processes not the operating system.
- Major issues is to provide mechanism that will allow the user processes to synchronize their actions when they access shared memory.



# IPC-Message Passing

- Mechanism for processes to communicate and to synchronize their actions
- Message system – processes communicate with each other without resorting to shared variables
- IPC facility provides two operations:
  - **send**(*message*)
  - **receive**(*message*)
- The *message* size is either fixed or variable



# IPC-Message Passing

- If processes  $P$  and  $Q$  wish to communicate, they need to:
  - Establish a *communication link* between them
  - Exchange messages via send/receive
- Implementation of communication link
  - Physical:
    - Shared memory
    - Hardware bus
    - Network
  - Logical:
    - Direct or indirect
    - Synchronous or asynchronous
    - Automatic or explicit buffering

# Direct Communication

- Processes must name each other explicitly:
  - **send** ( $P$ , *message*) – send a message to process  $P$
  - **receive**( $Q$ , *message*) – receive a message from process  $Q$
- Properties of communication link
  - Links are established automatically
  - A link is associated with exactly one pair of communicating processes
  - Between each pair there exists exactly one link
  - The link may be unidirectional, but is usually bi-directional



# Indirect Communication

- Messages are directed and received from mailboxes (also referred to as ports)
  - Each mailbox has a unique id
  - Processes can communicate only if they share a mailbox
- Properties of communication link
  - Link established only if processes share a common mailbox
  - A link may be associated with many processes
  - Each pair of processes may share several communication links
  - Link may be unidirectional or bi-directional
- Mailbox sharing
  - $P_1$ ,  $P_2$ , and  $P_3$  share mailbox A
  - $P_1$ , sends;  $P_2$  and  $P_3$  receive

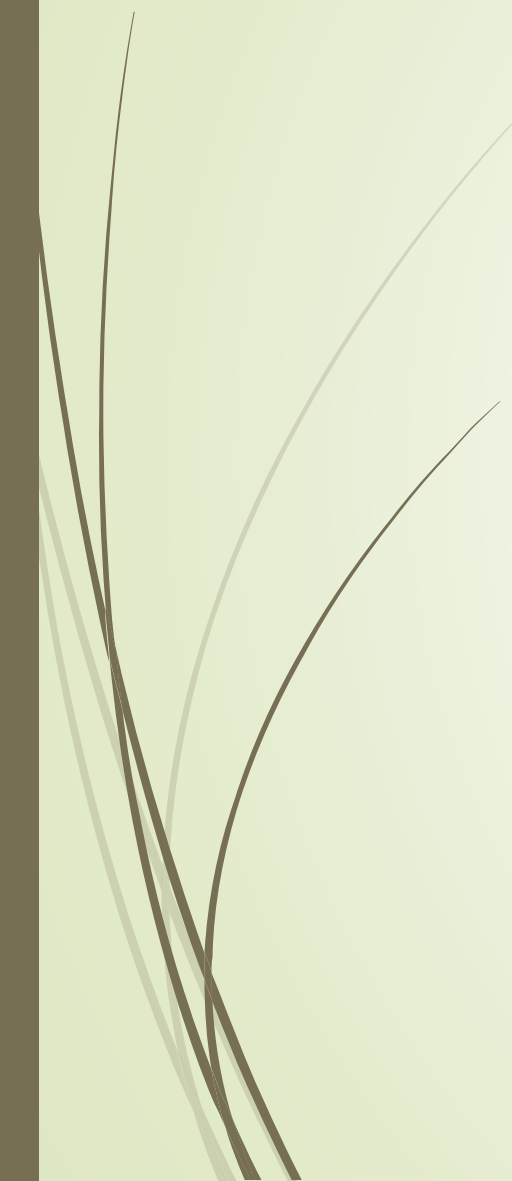


# Critical Section

- Process have to access shared memory or files or doing other critical tasks
- That part of the program where the shared memory is accessed is called **critical section or critical region**
- Consider system of  $n$  processes  $\{p_0, p_1, \dots, p_{n-1}\}$
- Each process has **critical section** segment of code
  - Process may be changing common variables, updating table, writing file, etc
  - When one process in critical section, no other may be in its critical section
- ***Critical section problem*** is to design protocol to solve this
- Each process must ask permission to enter critical section in **entry section**, may follow critical section with **exit section**, then **remainder section**



# CS-Conditions to have good solution

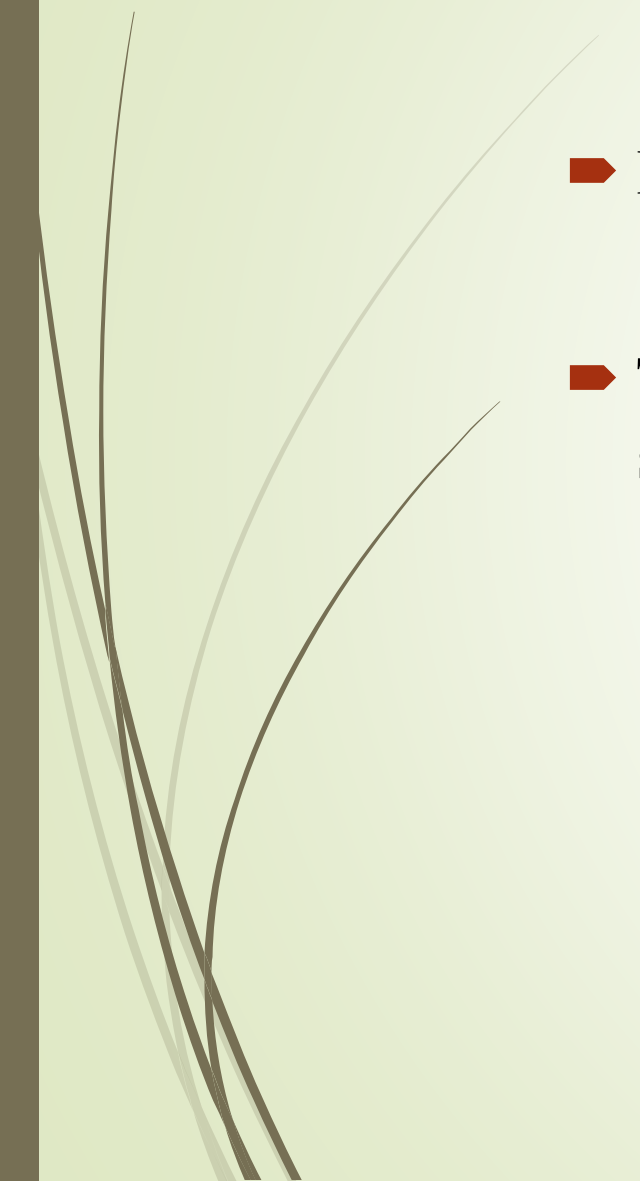
- No two processes may be in critical section
  - No assumptions may be made about the speed or number of CPU
  - No process running outside the critical section may block other process
  - No process should have wait forever to enter its critical section
- 

# Critical Section Solutions

- **Mutual Exclusion** - If process  $P_i$  is executing in its critical section, then no other processes can be executing in their critical sections
- **Progress** - If no process is executing in its critical section and there exist some processes that wish to enter their critical section, then the selection of the processes that will enter the critical section next cannot be postponed indefinitely
- **Bounded Waiting** - A bound must exist on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted
  - Assume that each process executes at a nonzero speed
  - No assumption concerning **relative speed** of the  $n$  processes



# Race Conditions

- It occurs inside a critical section
  - This happens when the result of multiple thread execution in critical section differs according to the order in which the threads execute
- 

# Race Condition (contd.)

- **counter++** could be implemented as

```
register1 = counter  
register1 = register1 + 1  
counter = register1
```

- **counter--** could be implemented as

```
register2 = counter  
register2 = register2 - 1  
counter = register2
```

- Consider this execution interleaving with "count = 5" initially:

S0: producer execute	<b>register1 = counter</b>	{ register1 = 5 }
S1: producer execute	<b>register1 = register1 + 1</b>	{ register1 = 6 }
S2: consumer execute	<b>register2 = counter</b>	{ register2 = 5 }
S3: consumer execute	<b>register2 = register2 - 1</b>	{ register2 = 4 }
S4: producer execute	<b>counter = register1</b>	{ counter = 6 }
S5: consumer execute	<b>counter = register2</b>	{ counter = 4 }

# Implementing Mutual Exclusion

Prohibiting more than one process from reading writing the same data at the same time

- How to Manage Race Condition
  - Mutual Exclusion with Busy Waiting
    - Lock Variable
    - Interrupt Disabling
    - Strict Alteration
    - Peterson's Algorithm
    - Test and Set Lock
  - Mutual Exclusion without Busy Waiting
    - Sleep and wakeup
    - Semaphore



# ME with Busy Waiting

- Continuously testing a variable until some value appears is called busy waiting
- If the entry is allowed it execute else it sits in tight loop and waits
- Busy-waiting: consumption of CPU cycles while a thread is waiting for a lock
  - Very inefficient
  - Can be avoided with a waiting queue

## ➤ Interrupt Disabling

- Each process disable all interrupts just after entering its CR and reenale them just before leaving it
- No clock interrupt, no other interrupt, no CPU switching to other process until the process turn on the interrupt

# ME with Busy Waiting

## ➤ Lock Variable

- A single, shared (lock) variable, initially 0. When a process wants to enter its CR, it first test the lock. If the lock is 0, the process set it to 1 and enters the CR. If the lock is already 1, the process just waits until it becomes 0.

## ➤ Strict Alteration

- Processes share a common integer variable turn.
  - If  $turn == i$  then process  $P_i$  is allowed to execute in its CR,
  - if  $turn == j$  then process  $P_j$  is allowed to execute

Process 0	Process 1
<pre>While (TRUE) {   while (turn != 0); //wait   critical_section();   turn = 1;   noncritical_section(); }</pre>	<pre>While (TRUE) {   while (turn != 1); // wait   critical_section();   turn = 0;   noncritical_section(); }</pre>



# ME with Busy Waiting

## ➤ Peterson's Algorithm

- A classical software-based solution to the critical-section problem.
- Before entering its CR, each process call `enter_region` with its own process number, 0 or 1 as parameter.
- Call will cause to wait, if need be, until it is safe to enter
- When leaving CR, the process calls `leave_region` to indicate that it is done and to allow other process to enter CR

# Peterson's Algorithm

```
#define FALSE 0
#define TRUE 1
#define N      2                /* number of processes */

int turn;                       /* whose turn is it? */
int interested[N];              /* all values initially 0 (FALSE) */

void enter_region(int process); /* process is 0 or 1 */
{
    int other;                  /* number of the other process */

    other = 1 - process;        /* the opposite of process */
    interested[process] = TRUE; /* show that you are interested */
    turn = process;             /* set flag */
    while (turn == process && interested[other] == TRUE) /* null statement */ ;
}

void leave_region(int process) /* process: who is leaving */
{
    interested[process] = FALSE; /* indicate departure from critical region */
}
```

# ME with Busy Waiting

## ➤ Test and Set Lock

- Help from the hardware
- Reads the content of the memory word lock into register RX and then stores a non-zero value at the memory address lock
- A shared variable (lock) is used to coordinate access to shared memory
- When lock is 0, any process may set it to 1 using TSL instructions and then read or write the shared memory
- When all the tasks are done, lock is set back to 0 using ordinary move instructions.



# ME without Busy Waiting

## Sleep and Wakeup

- Both Peterson's solution and TSL are correct but have defect requiring busy waiting
- These approaches waste CPU time
- Sleep and wakeup is simplest IPC primitives which blocks instead wasting CPU time
- When a process is not permitted to access its critical section, it uses a system call known as **Sleep**, which causes that process to block
- The process will not be scheduled to run again, until another process uses the **Wakeup** system call
- In most cases, **Wakeup** is called by a process when it leaves its critical section if any other processes have blocked

# ME without Busy Waiting (contd.)

## Sleep and Wakeup (Producer-Consumer Problem)

- Two process share a common, fixed sized buffer
  - Suppose one process, producer, is generating information that the second process, consumer, is using
  - Their speed may be mismatched, if producer insert item rapidly, the buffer will full and go to the sleep until consumer consumes some item, while consumer consumes rapidly, the buffer will empty and go to sleep until producer put something in the buffer
- 
- Code at [OS Sleep and Wake](#)



# ME without Busy Waiting (contd.)

## Semaphores

- E. W. Dijkstra (1965) suggested using an integer variable to count the number of wakeups, called a semaphore
- It could have the value 0, indicating no wakeups were saved, or some positive value if one or more wakeups were pending
- Operations: wait() and signal()
- Wait: checks if the value greater than 0, if true, decrements the value and continue
  - If value is 0, the process is put to sleep without completing down
  - Checking value, changing it, and possibly going to sleep, is all done as single action
- Signal() : increments the value by 1
  - If one or more processes were sleeping, unable to complete earlier down operation, one of them is chosen and is allowed to complete its down.



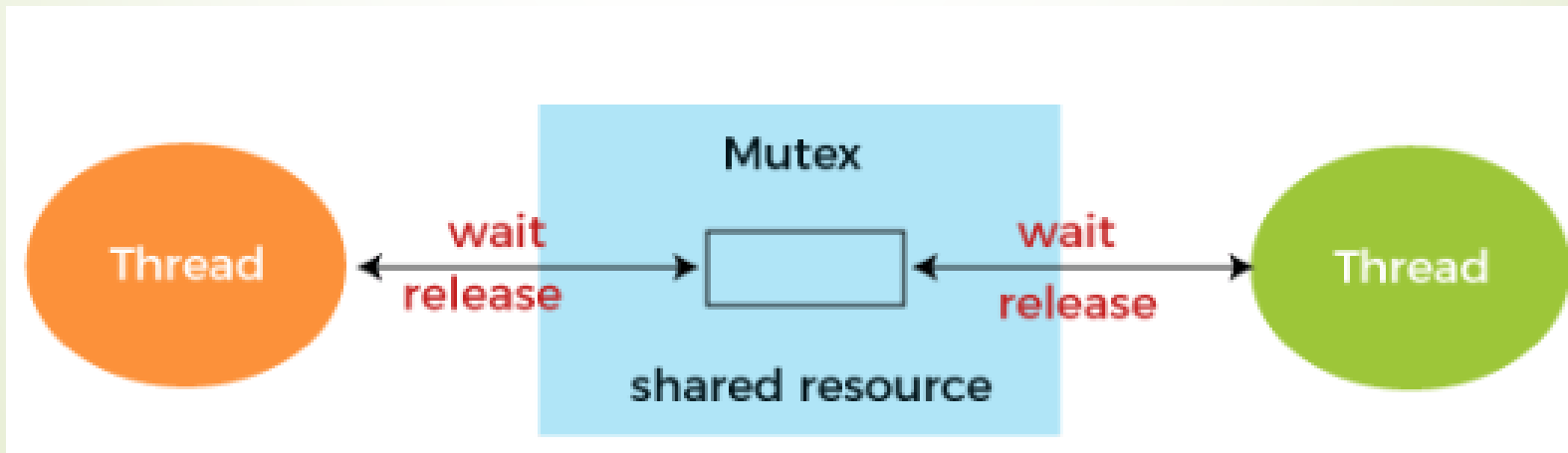
# Monitors

- A monitor is a programming language construct that guarantees appropriate access to the CR.
- None of the routine in the monitor can be executed by a process until that process acquires lock
- This means only one process can be executed within a monitor at a time
- Any other process must wait for the process that's currently executing to give up control of the lock
- Processes that wish to access the shared data, do through the execution of monitor functions
- It is a collection of procedures, variables , and data structures that are all grouped together in a special kinds of module or package



# Mutex

- Mutex is a mutual exclusion object that synchronizes access to a resource.
- It is created with a unique name at the start of a program.
- The mutex locking mechanism ensures only one thread can acquire the mutex and enter the critical section.
- This thread only releases the mutex when it exits in the critical section. A mutex can be unlocked only by the thread that locked it.

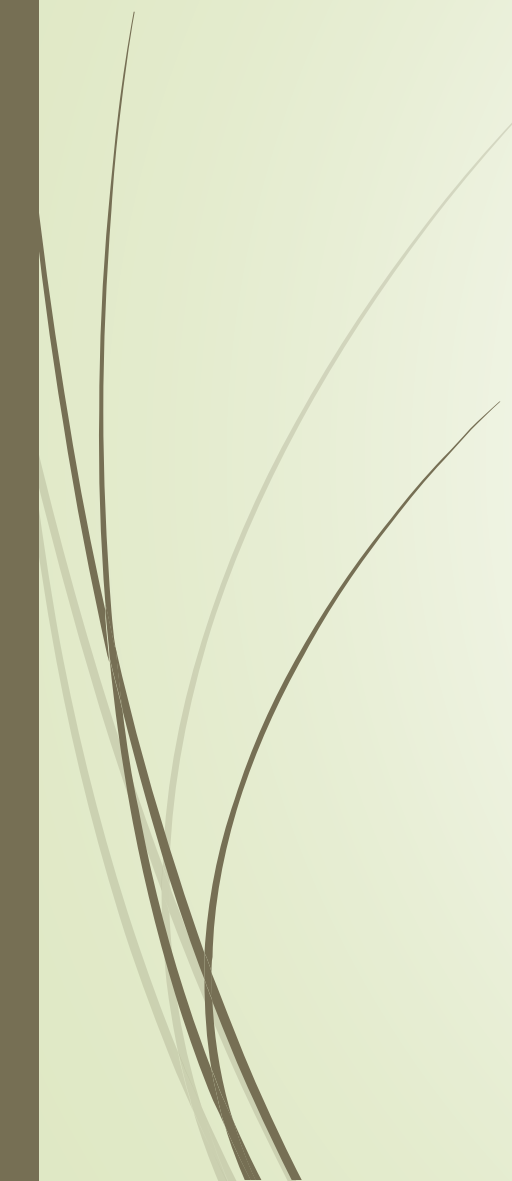


# Message Passing

- This method of IPC uses two primitives :
  - Send
  - Receive
- These are system calls rather than constructors
- Can be synchronous and Asynchronous.
- They can easily put into library procedures such as
  - Send(destination, &message)
  - Receive(source, &message)



# Classical IPC Problem

- Producer-Consumer Problem
  - Sleeping Barber Problem
  - Dining Philosopher Problem
  - Readers Writers Problem
- 

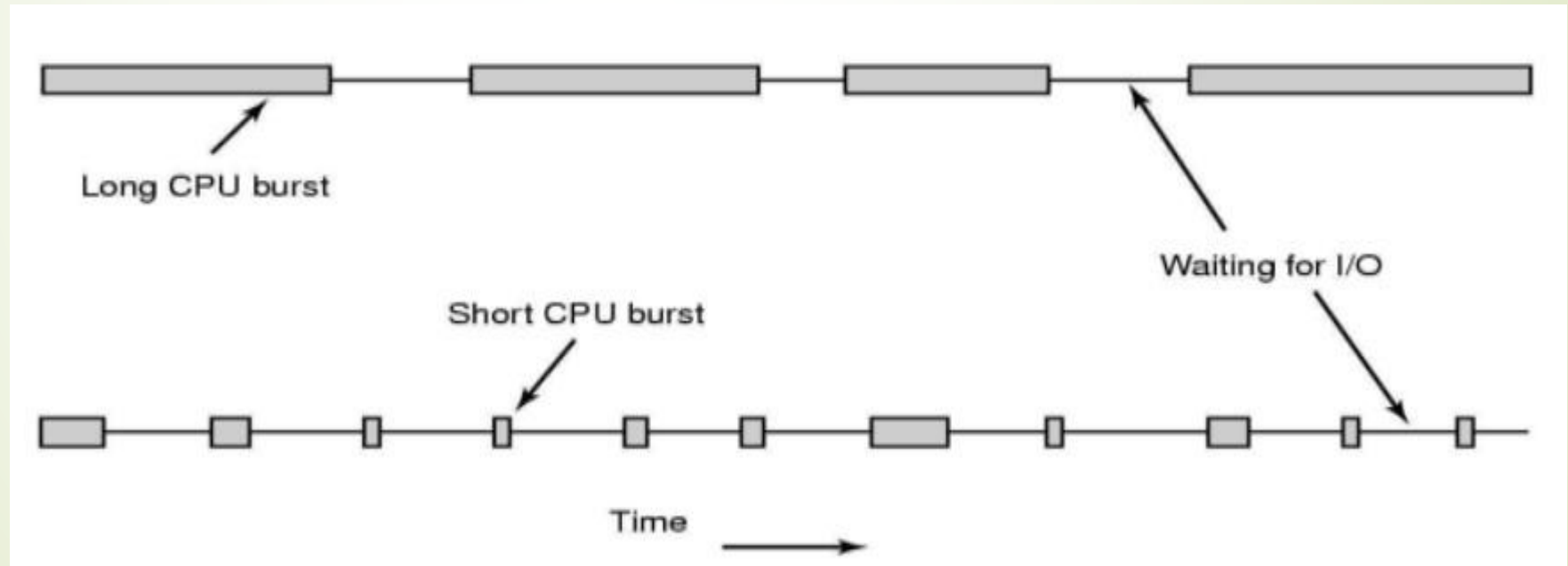


# Process Scheduling

- The part of OS that makes the choice which process to run next whenever two or more process are in ready state – **Scheduler**
- The algorithm it uses is called **Scheduling algorithm**
- Process scheduling is the activity of process manager that handles the removal of the running process from the CPU and the selection of another process on the basis of particular strategy

# Process Behavior

- The process that uses CPU until quantum(time) expires – CPU Bound
- The process that uses CPU briefly then generate I/O – I/O Bound
- CPU bound process have long CPU-burst while I/O bound process have short CPU burst



# Preemptive v/s Non-Preemptive

## ➤ Non-preemptive:

- Once a process has been given the CPU, it runs until blocks for I/O or termination
- Treatment of all processes is fair
- Response times are more predictable
- Useful in real-time system
- Short jobs are made to wait by longer jobs - no priority

## ➤ Preemptive:

- Processes are allowed to run for a maximum of some fixed time
- Useful in systems in which high-priority processes requires rapid attention
- In timesharing systems, preemptive scheduling is important in guaranteeing acceptable response times
- High overhead

# Scheduling Criteria

- **CPU utilization** – keep the CPU as busy as possible
- **Turnaround time** – amount of time to execute a particular process

$$\text{T.A.T} = \text{Completion time} - \text{Arrival Time}$$

- **Waiting time** – amount of time a process has been waiting in the ready queue

$$\text{W.T} = \text{T.A.T} - \text{Brust time}$$

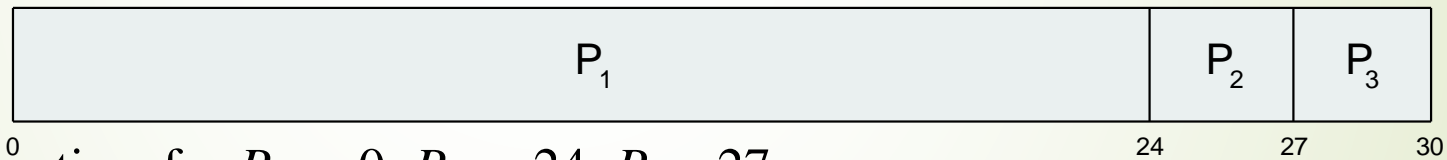
- **Response time** – amount of time it takes from when a request was submitted until the first response is produced, not output (for time-sharing environment)



# FCFS Scheduling

<u>Process</u>	<u>Burst Time</u>
$P_1$	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<sup>0</sup> time for  $P_1 = 0$ ;  $P_2 = 24$ ;  $P_3 = 27$
- Average waiting time:  $(0 + 24 + 27)/3 = 17$

# FCFS Scheduling (contd.)

Suppose that the processes arrive in the order:

$$P_2, P_3, P_1$$

➤ The Gantt chart for the schedule is:

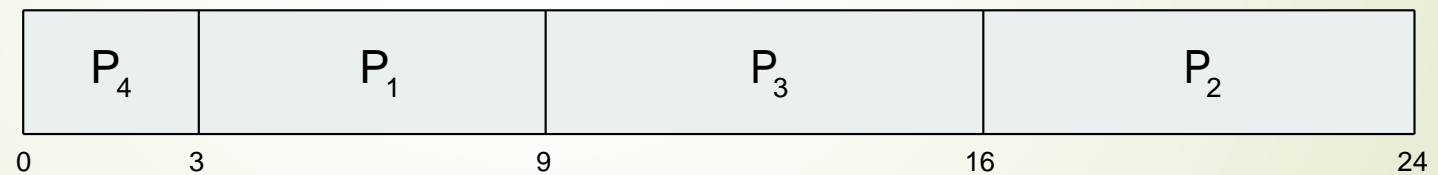


- Waiting time for  $P_1 = 6$ ;  $P_2 = 0$ ,  $P_3 = 3$
- Average waiting time:  $(6 + 0 + 3)/3 = 3$
- Much better than previous case
- **Convoy effect** - short process behind long process
  - Consider one CPU-bound and many I/O-bound processes

# SJF Scheduling

<u>Process</u>	<u>Burst Time</u>
$P_1$	6
$P_2$	8
$P_3$	7
$P_4$	3

➤ SJF scheduling chart



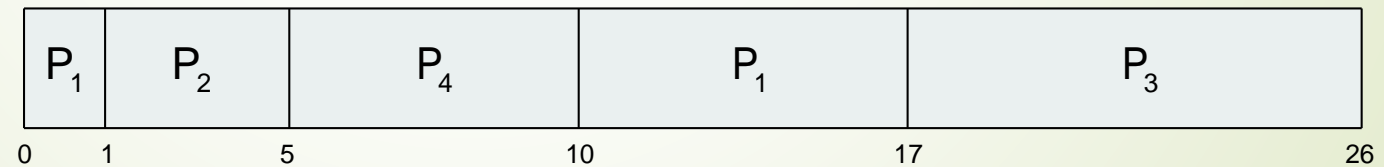
➤ Average waiting time =  $(3 + 16 + 9 + 0) / 4 = 7$

# SRTF Scheduling

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

<u>Process</u>	<u>Arrival Time</u>	<u>Burst Time</u>
$P_1$	0	8
$P_2$	1	4
$P_3$	2	9
$P_4$	3	5

- Gantt Chart



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



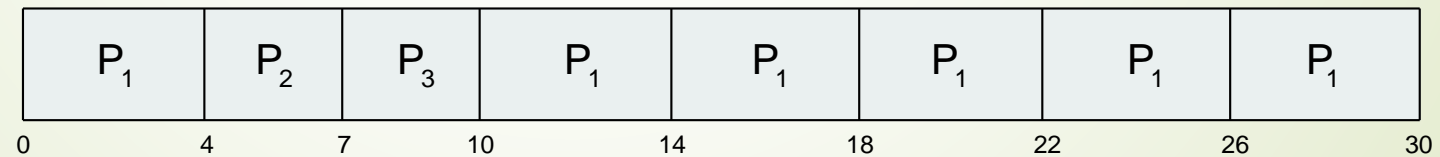
# Round Robin (RR) Scheduling

- Each process is assigned a time interval (quantum), after the specified quantum, the running process is preempted to the last and a new process is allowed to run
- Advantages:
  - Fair allocation of CPU across the process.
  - Used in timesharing system.
  - Low average waiting time when process lengths vary widely

# RR with quantum=4

<u>Process</u>	<u>Burst Time</u>
$P_1$	24
$P_2$	3
$P_3$	3

➡ The Gantt chart is:





# Priority ( Event-Driven [ED] )Scheduling

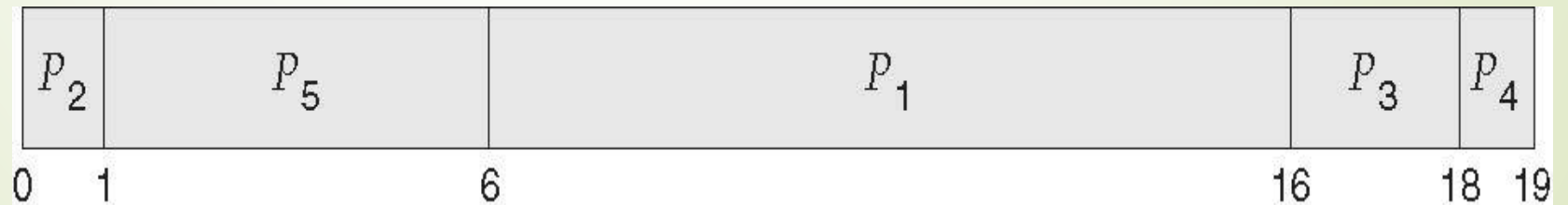
- A priority number (integer) is associated with each process
- The CPU is allocated to the process with the highest priority (smallest integer = highest priority)
- Problem: **Starvation** – low priority processes may never execute
- Solution: **Aging** – as time progresses increase the priority of the process



# Priority Scheduling (contd.)

<u>Process</u>	<u>Burst Time</u>	<u>Priority</u>
$P_1$	10	3
$P_2$	1	1
$P_3$	2	4
$P_4$	1	5
$P_5$	5	2

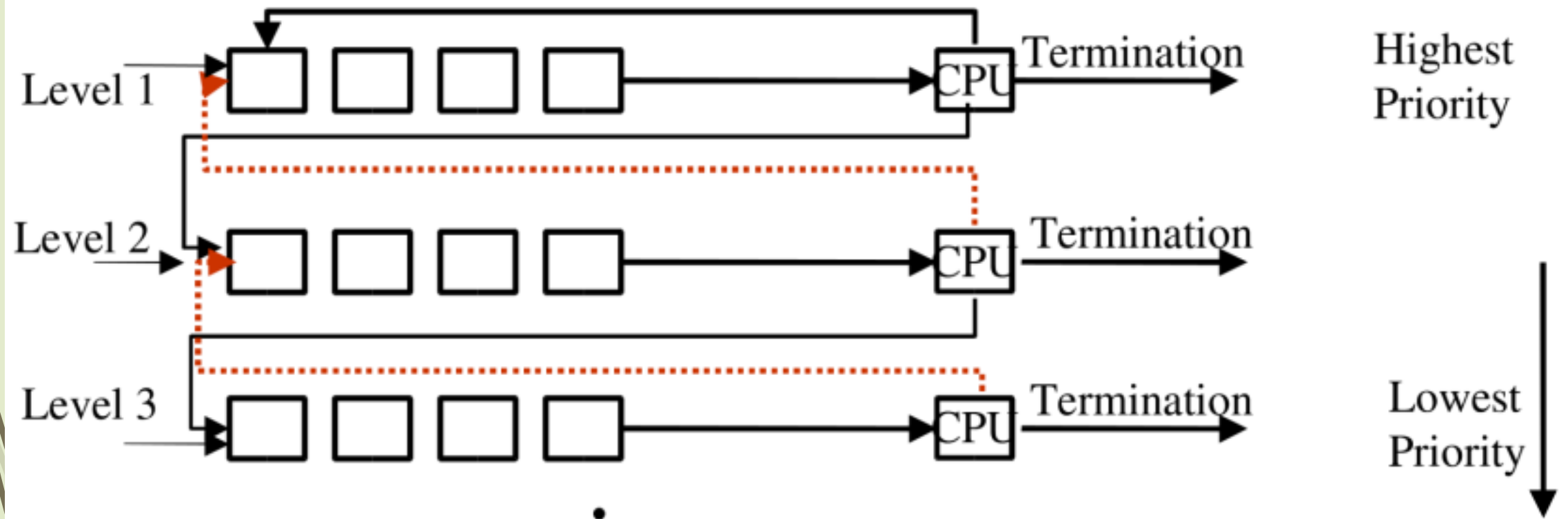
➤ Priority scheduling Gantt Chart



# Multilevel-Feedback-Queues (MFQ)

- A process can move between the various queues; aging can be implemented this way
- MFQ implements multilevel queues having different priority to each level (here lower level higher priority), and allows a process to move between the queues.
- If the process use too much CPU time, it will be moved to a lower-priority queue
- Higher the level of queue(lower the priority) : larger the quantum size
- This leaves the I/O-bound and interactive processes in the high priority queue

## MFQ (contd.)





# Real-Time CPU Scheduling

- **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