

# Process Management

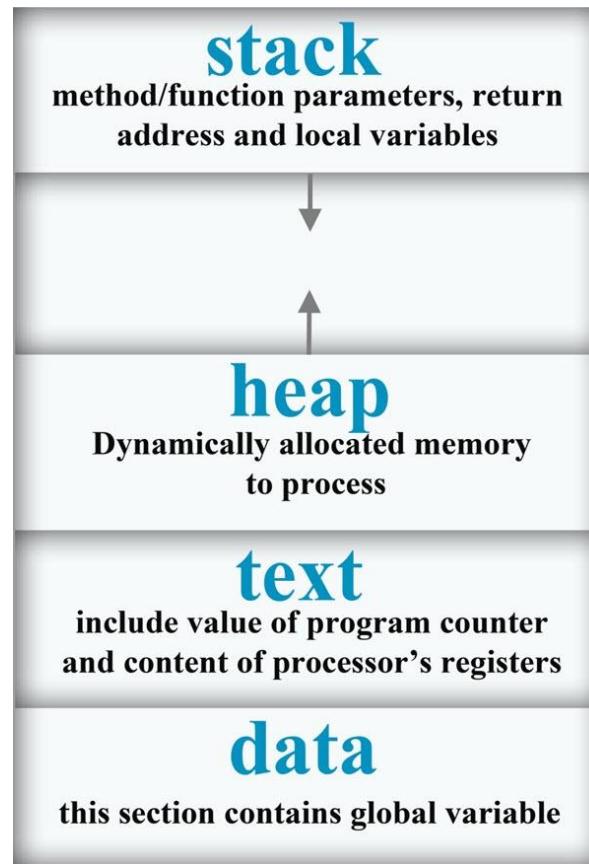
## Unit 2

# Process

- A process is an instance of a program running in a computer.
- It is close in meaning to task, a term used in some OS.
- In UNIX and some other OS, a process is started when a program is initiated (either by a user entering a shell command or by another program).
- A program by itself is not a process; a program is a *passive entity*, such as a file containing a list of instructions stored on disks. (often called an executable file)
- A program becomes a process when an executable file is loaded into memory and executed.

# Process..

- When a program is loaded into a memory and it becomes a process , it can be divided into four sections :



# Process..

- **Stack** : it contain temporary data such as method/ function parameters return address and local variables.
- **Heap** : this is dynamically allocated memory to a process during its run time.
- **Text** : this includes the current activity represented by the value of program counter and the contents of the processor's registers.
- **Data**: This section contain the global and static variable

# Program

- Program is an executable file containing the set of instructions written to perform a specific job on our computer.
- E.g. chrome.exe  
is an executable file containing the set of instruction written so that we can view web page.
- A program is piece of code which may be a single line or millions of lines.
- Program are not stored on the primary memory on our computer. They are stored on a disk or secondary memory of our computer

# Process Vs Program

BASIS FOR COMPARISON	PROGRAM	PROCESS
Basic	Program is a set of instruction.	When a program is executed, it is known as process.
Nature	Passive	Active
Lifespan	Longer	Limited
Required resources	Program is stored on disk in some file and does not require any other resources.	Process holds resources such as CPU, memory address, disk, I/O etc.

# Key Differences Between Program and Process

- A program is a definite group of **ordered operations** that are to be performed. On the other hand, an **instance** of a program being executed is a process.
- The nature of the program is passive as it does nothing until it gets executed whereas a process is dynamic or active in nature as it is an instance of executing program and perform the specific action.
- A program has a **longer** lifespan because it is stored in the memory until it is not manually deleted while a process has a shorter and **limited** lifespan because it gets terminated after the completion of the task.
- The resource requirement is much higher in case of a process; it could need processing, memory, I/O resources for the successful execution. In contrast, a program just requires memory for storage.

## *The example of difference between a program and a process*

```
main ()  
{  
    int i , prod =1;  
    for (i=1;i<=100;i++)  
        prod = prod*i;  
}
```

- It is a program containing one multiplication statement  
(prod = prod \* i)
- but the process will execute 100 multiplication, one at a time through the 'for' loop.

# Process...

- Although two processes may be associated with the same program, they are nevertheless considered two separate execution sequences.
- For instance several users may be running different copies of mail program, or the same user may invoke many copies of web browser program.
- Each of these is a separate process, and although the text sections are equivalent, the data, heap and stack section may vary.

# Process Creation:

There are four principal events that cause processes to be created:

1. System initialization.
2. Execution of a process creation system call by a running process.
3. A user request to create a new process.
4. Initiation of a batch job.

# Process Creation..

- Parent process create children processes, which, in turn create other processes, forming a tree of processes .
- Generally, process identified and managed via a process identifier (pid).
- When an operating system is booted, often several processes are created.
- Some of these are foreground processes, that is, processes that interact with (human) users and perform work for them.

# Process Creation..

## Daemons

( a **daemon** is a computer program that runs as a Hardware activity, or other programs by performing some task. )

- Processes, which are not associated with particular users, but instead have some specific function.
- Processes that stay in the background to handle some activity such as web pages, printing, and so on.

# Process Creation..

In UNIX there is only one system call to create a new process:  
**fork()**.

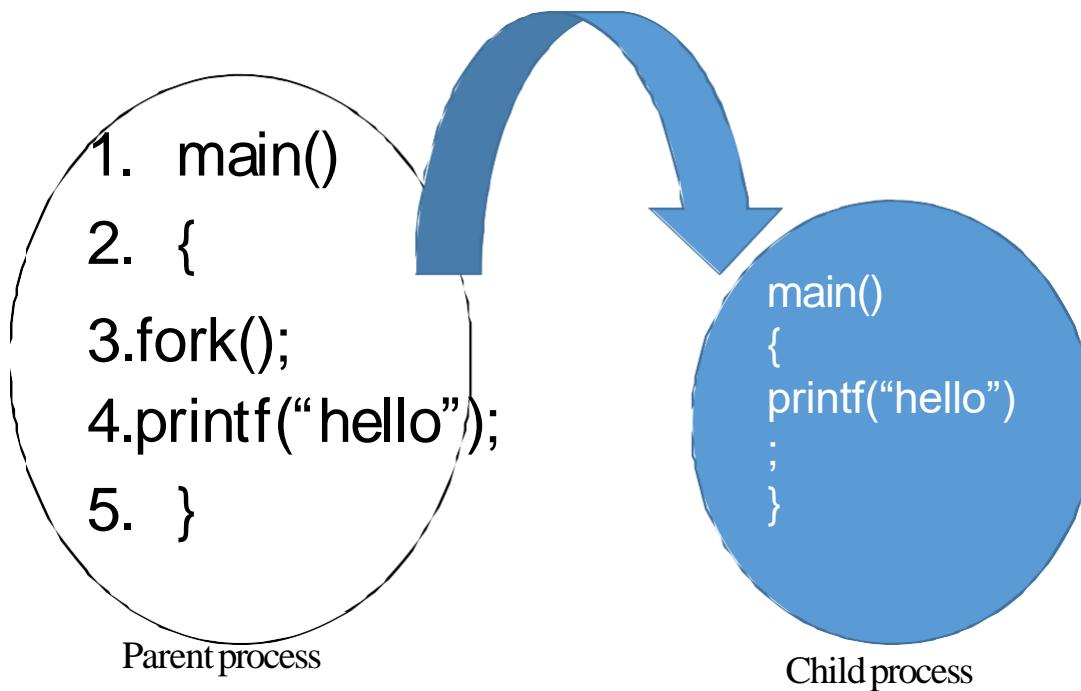
- This call creates an exact clone of the calling process.
- After the fork, the two processes, the parent and the child, have the same memory image, the same environment strings, and the same open files.
- Usually, the child process then executes **execve or a similar system** call to change its memory image and run a new program.
- Fork() returns a negative value if the child creation is unsuccessful.
- Fork() returns the value zero if the child creation is successful.
- Fork() returns a positive integer (the process id of the parent process).

# Process Creation

```
1. #include<stdio.h>
2. #include<unistd.h>
3. int main() {
4.     int pid;
5.     pid = fork(); /* fork another process */
6.     if (pid < 0) { /* error occurred */
7.         printf("child process creation is Failed");
8.     }
9.     else if (pid == 0) { /* child process */
10.         printf("child process");
11.         printf("%d",pid);
12.     }
13.     else { /* parent process */
14.         printf ("parent process");
15.         printf("%d",pid);
16.     }
17. }
```

# Process Creation..

- fork() system call implementation for process creation



- Here are two processes one is parent and next is newly created by fork system call.
- printf("hello") statement is executed twice. One by child process and another by parent process itself

# Process Creation..

If so what will be the o/p for this???

1. main()
2. {
3.     fork();
4.     fork();
5.     printf("hello");
6. }

# Process Creation..

And this???

```
1. main()  
2. {  
3.     fork();  
4.     fork();  
5.     fork();  
6.     printf("hello");  
7. }
```

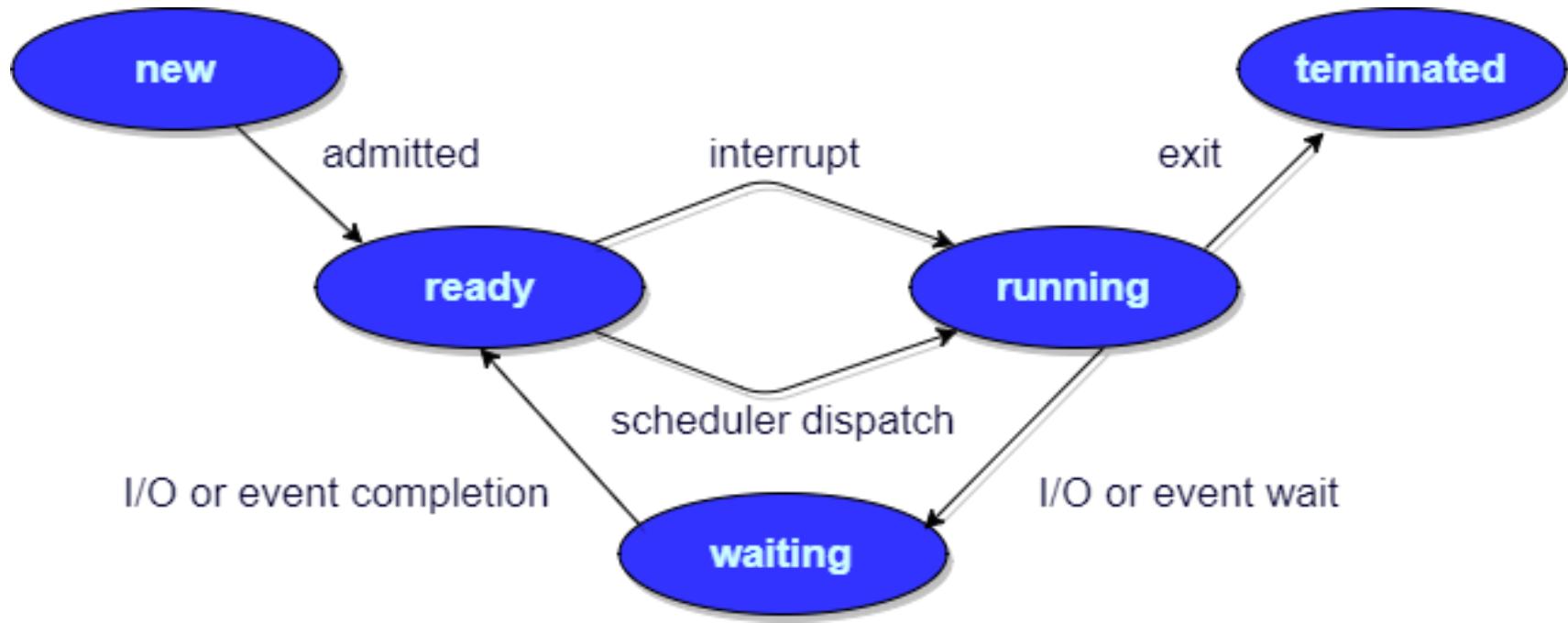
- If a program consists of **n** **fork()** calls then it will create  $2^n - 1$  childprocess

# Different Process States (Process life cycle)

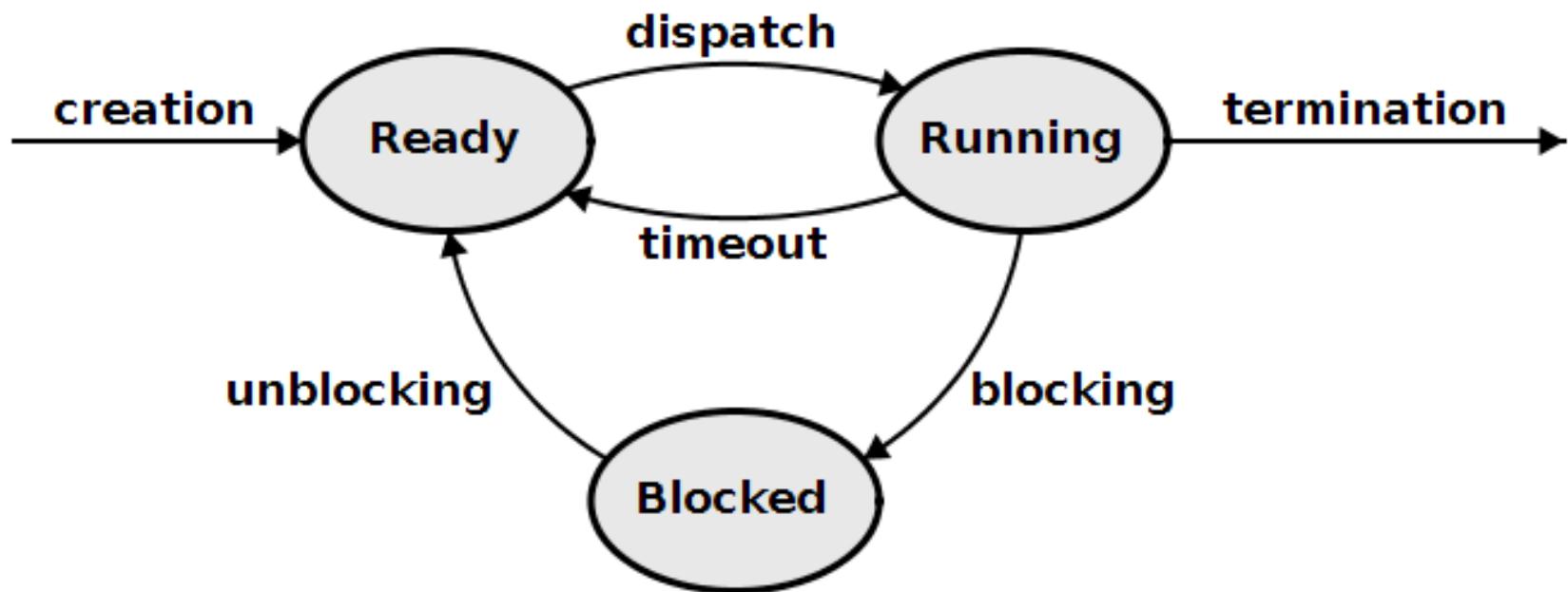
Processes in the operating system can be in any of the following states:

- **NEW-** The process is being created.
- **READY-** The process is waiting to be assigned to a processor.
- **RUNNING-** Instructions are being executed.
- **WAITING-** The process is waiting for some event to occur(such as an I/O completion or reception of a signal).
- **TERMINATED-** The process has finished execution.

# Different Process States (Process life cycle)...



# Process life cycle



# Process Control Block (PCB)

- Process Control Block (PCB, also called Task Controlling Block, Entry of the Process Table, Task Struct, or Switchframe) is a data structure in the operating system kernel containing the information needed to manage the scheduling of a particular process.
- The PCB is "the manifestation(expression) of a process in an operating system."

# Process Control Block

- While creating a process the operating system performs several operations. To identify these processes, it must identify each process, hence it assigns a process identification number (PID) to each process.
- As the operating system supports multi-programming, it needs to keep track of all the processes.
- For this task, the process control block (PCB) is used to track the process's execution status.
- Each block of memory contains information about the process state, program counter, stack pointer, status of opened files, scheduling algorithms, etc.
- All this information is required and must be saved when the process is switched from one state to another.
- When the process made transitions from one state to another, the operating system must update information in the process's PCB.

# Role of PCB

- The role or work of **process control block** (PCB) in process management is that it can access or modified by most OS utilities including those are involved with memory, scheduling, and input / output resource access.
- It can be said that the set of the process control blocks give the information of the current state of the operating system.
- Data structuring for processes is often done in terms of process control blocks.
- For example, pointers to other process control blocks inside any process control block allows the creation of those queues of processes in various scheduling states.

# Role of PCB..

**The following are the various information that is contained by process control block:**

- Naming the process
- State of the process
- Resources allocated to the process
- Memory allocated to the process
- Scheduling information
- Input / output devices associated with process

# Components of PCB

- The following are the various components that are associated with the process control block PCB:
  - 1. Process ID:**
  - 2. Process State**
  - 3. Program counter**
  - 4. Register Information**
  - 5. Scheduling information**
  - 6. Memory related information**
  - 7. Accounting information**
  - 8. Status information related to input/output**

# Components of PCB..

Process Id
Process state
Program counter
Register information
Scheduling information
Memory related information
Accounting information
Status information related to I/O

# Components of PCB..

## 1. Process ID:

In computer system there are various process running simultaneously and each process has its unique ID. This Id helps system in scheduling the processes. This Id is provided by the process control block.

In other words, it is an identification number that uniquely identifies the processes of computer system.

# Components of PCB..

## 2. Process state:

As we know that the process state of any process can be New, running, waiting, executing, blocked, suspended, terminated.

For more details regarding process states you can refer process management of an Operating System. Process control block is used to define the process state of any process.

In other words, process control block refers the states of the processes.

# Components of PCB..

- **3. Program counter:**

Program counter is used to point to the address of the next instruction to be executed in any process. This is also managed by the process control block.

- **4. Register Information:**

This information is comprising with the various registers, such as index and stack that are associated with the process. This information is also managed by the process control block.

# Components of PCB..

- **5. Scheduling information:**

Scheduling information is used to set the priority of different processes. This is very useful information which is set by the process control block. In computer system there were many processes running simultaneously and each process have its priority. The priority of primary feature of RAM is higher than other secondary features. Scheduling information is very useful in managing any computer system.

# Components of PCB..

- **6. Memory related information:**

This section of the process control block comprises of page and segment tables. It also stores the data contained in base and limit registers.

- **7. Accounting information:**

This section of process control block stores the details relate to central processing unit (CPU) utilization and execution time of a process.

- **8. Status information related to input / output:**

This section of process control block stores the details pertaining to resource utilization and file opened during the process execution.

# Process Table

- The operating system maintains a table called **process table**, which stores the process control blocks related to all the processes.
- The **process table** is a data structure maintained by the operating system to facilitate context switching and scheduling, and other activities discussed later.
- Each entry in the table, often called a **context block**, contains information about a process such as process name and state (discussed above), priority (discussed above), registers, and a semaphore it may be waiting on . The exact contents of a context block depends on the operating system. For instance, if the OS supports paging, then the context block contains an entry to the page table.

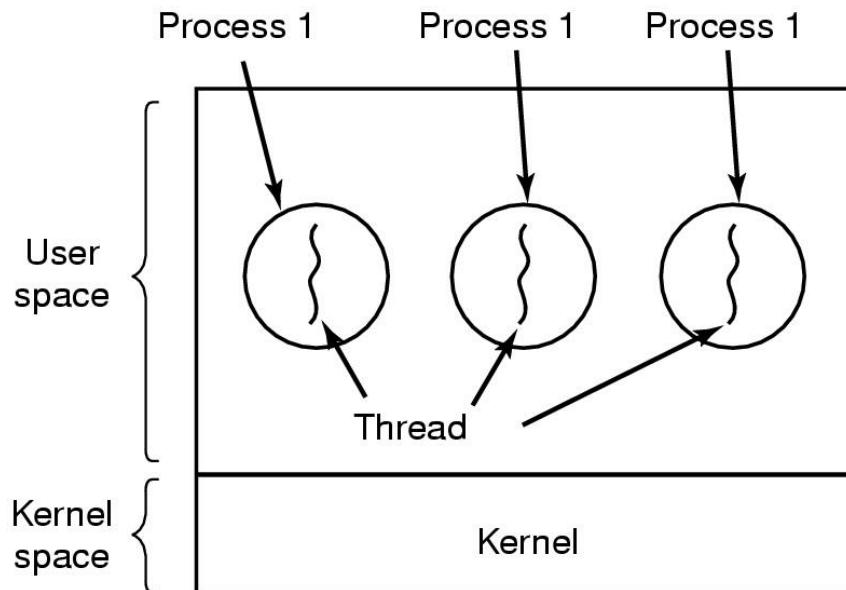
# Thread

- A thread is the smallest unit of processing that can be performed in an OS.
- In most modern operating systems, a thread exists within a process - that is, a single process may contain multiple threads.
- A thread is a basic unit of CPU utilization, it comprises a thread ID, a program counter, a register set, and a stack.
- It shares with other threads belonging to the same process its code section, data section, and other operating system resources, such as open files and signals.

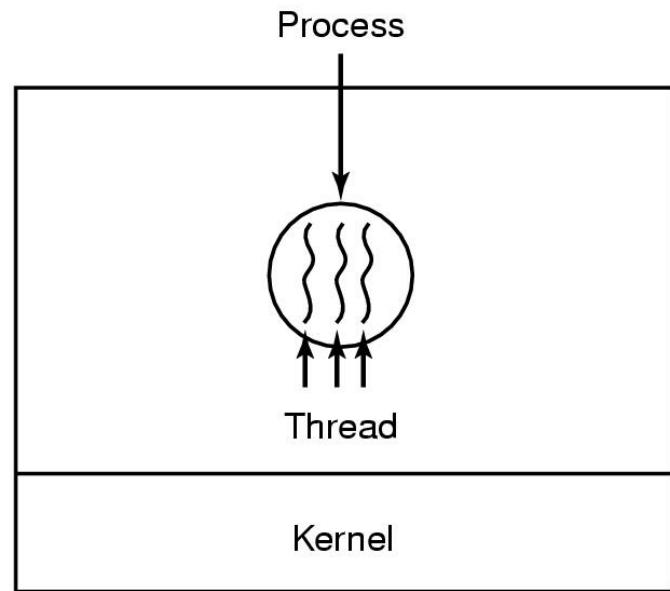
# Threads..

- A traditional ( or heavy weight) process has a single thread of control.
- If a process has multiple thread of control, it can perform more than one task at a time.
- Fig below illustrate the difference between single threaded process and a multithreaded process.

# Threads..



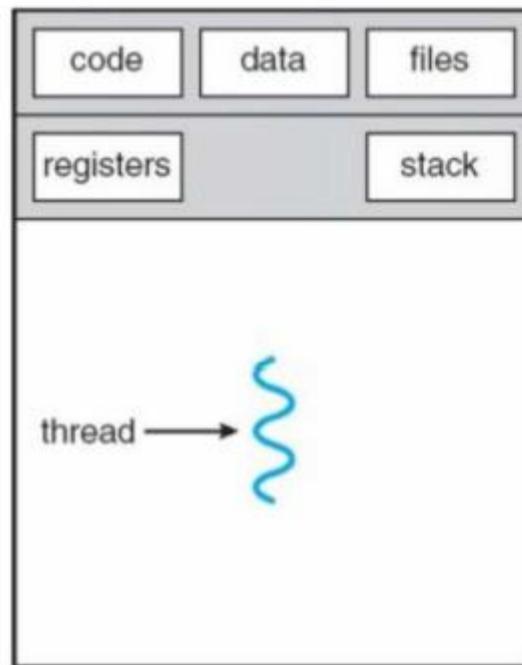
(a)



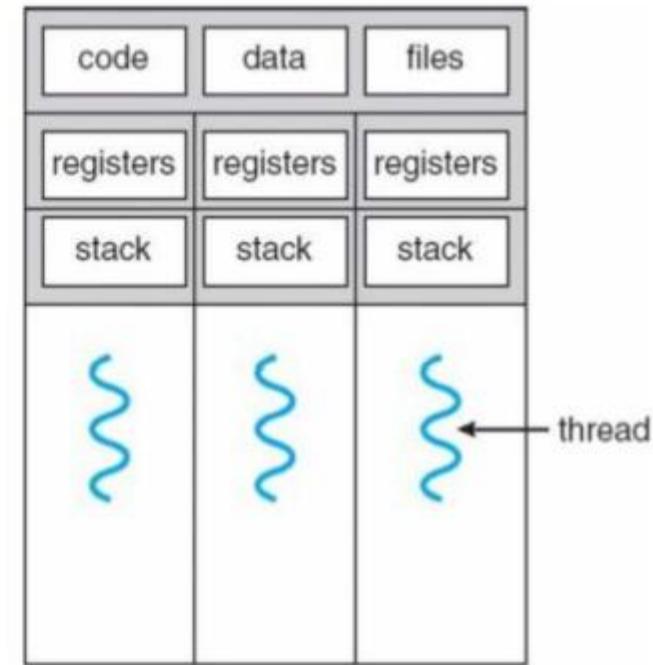
(b)

- (a) Three processes each with one thread
- (b) One process with three threads

# Threads

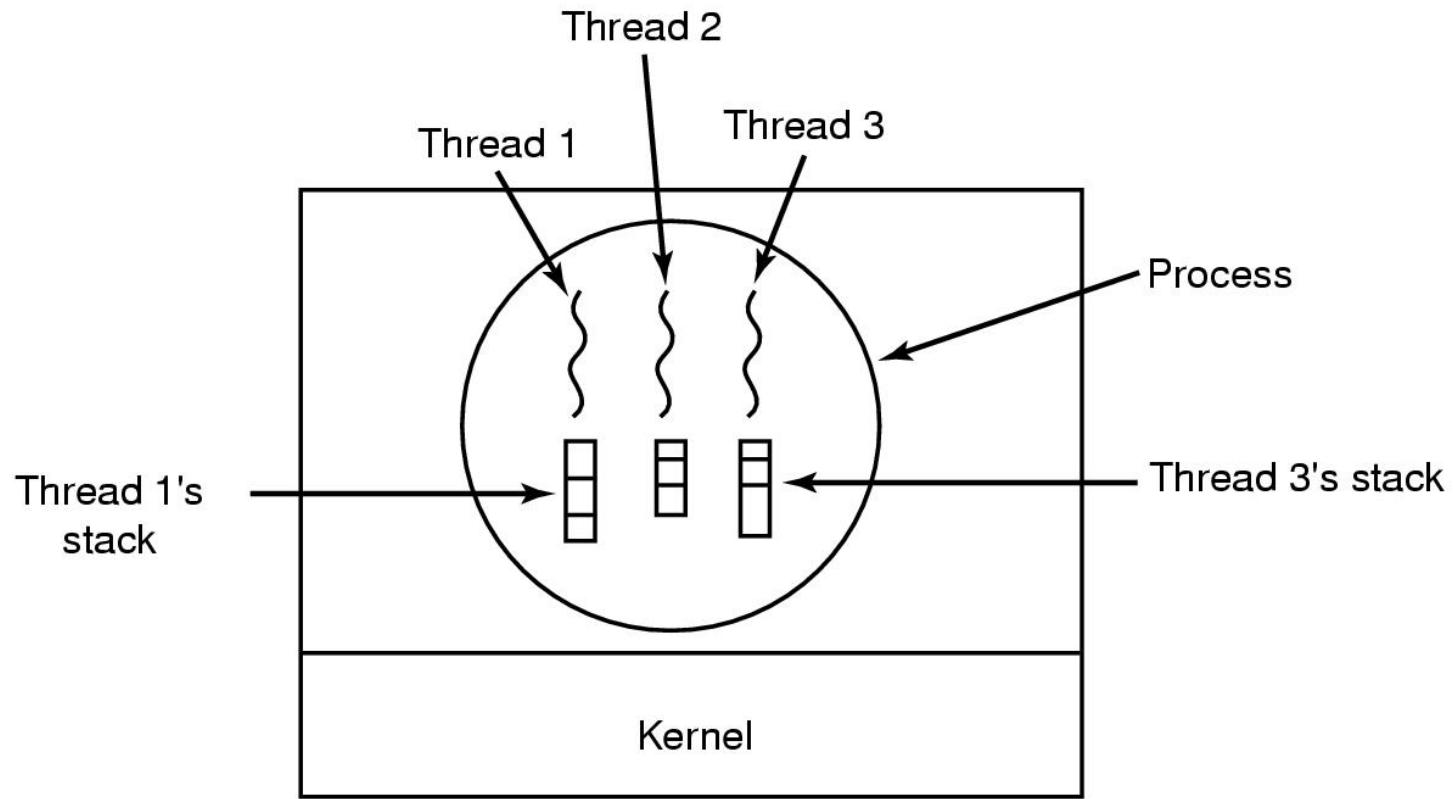


single-threaded process



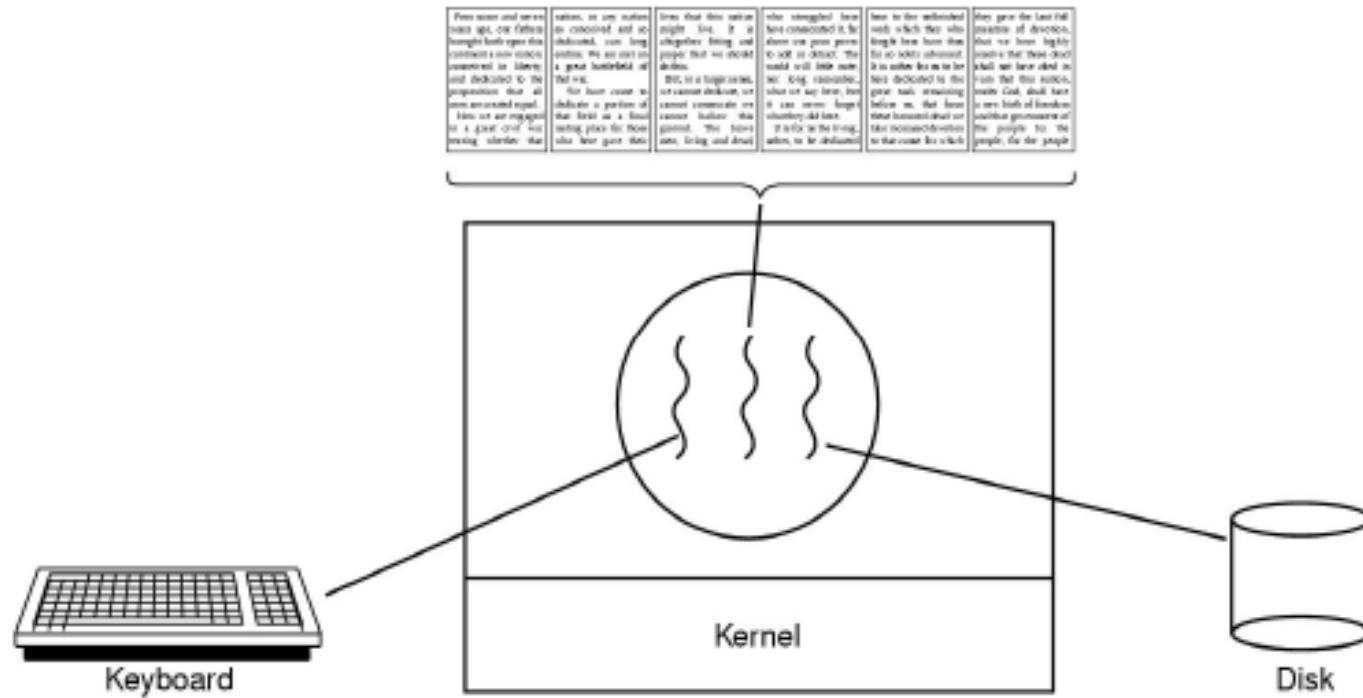
multithreaded process

# Threads..



Each thread has its own stack

# Thread Usage example



## A word processor with three threads

# Process Vs Threads

BASIS FOR COMPARISON	PROCESS	THREAD
Basic	Program in execution.	Lightweight process or part of it.
Memory sharing	Completely isolated and do not share memory.	Shares memory with each other.
Resource consumption	More	Less
Efficiency	Less efficient as compared to the process in the context of communication.	Enhances efficiency in the context of communication.
Time required for creation	More	Less
Context switching time	Takes more time.	Consumes less time.
Uncertain termination	Results in loss of process.	A thread can be reclaimed.
Time required for termination	More	

# Key Differences Between Process and Thread

- All threads of a program are logically contained within a process.
- A process is heavy weighted, but a thread is light weighted.
- A program is an isolated execution unit whereas thread is not isolated and shares memory.
- A thread cannot have an individual existence; it is attached to a process. On the other hand, a process can exist individually.
- At the time of expiration of a thread, its associated stack could be recovered as every thread has its own stack. In contrast, if a process dies, all threads die including the process.

# Properties of a Thread:

- Only one system call can create more than one thread (Lightweight process).
- Threads share data and information.
- Threads shares instruction, global and heap regions but has its own individual stack and registers.
- Thread management consumes no or fewer system calls as the communication between threads can be achieved using shared memory.
- The isolation property of the process increases its overhead in terms of resource consumption.

# Types of Thread

There are two types of threads:

- User Threads
- Kernel Threads

# User Level thread (ULT)

Is implemented in the user level library, they are not created using the system calls. Thread switching does not need to call OS and to cause interrupt to Kernel. Kernel doesn't know about the user level thread and manages them as if they were single-threaded processes.

## Advantages of ULT –

- Can be implemented on an OS that doesn't support multithreading.
- Simple representation since thread has only program counter, register set, stack space.
- Simple to create since no intervention of kernel.
- Thread switching is fast since no OS calls need to be made.

## Disadvantages of ULT –

- No or less co-ordination among the threads and Kernel.
- If one thread causes a page fault, the entire process blocks.

# Kernel Level Thread (KLT) –

Kernel knows and manages the threads. Instead of thread table in each process, the kernel itself has thread table (a master one) that keeps track of all the threads in the system. In addition kernel also maintains the traditional process table to keep track of the processes. OS kernel provides system call to create and manage threads.

## Advantages of KLT –

- Since kernel has full knowledge about the threads in the system, scheduler may decide to give more time to processes having large number of threads.
- Good for applications that frequently block.

## Disadvantages of KLT –

- Slow and inefficient.
- It requires thread control block so it is an overhead.

# Multithreading

- Many software packages that run on modern desktop PCs are multithreaded.
- An application is implemented as a separate process with several threads of control.
- A web browser might have one thread to display images or text while other threads retrieve data from the network.
- A word-processor may have a thread for displaying graphics, another thread for reading the characters entered by user through the keyboard, and a third thread for performing spelling and grammar checking in the background.

# Why Multithreading

- In certain situations, a single application may be required to perform several similar task such as a web server accepts client requests for web pages, images, sound, graphics etc.
- A busy web server may have several clients concurrently accessing it.
- So if the web server runs on traditional single threaded process, it would be able to service only one client at a time.
- The amount of time that the client might have to wait for its request to be serviced is enormous.
- One solution of this problem can be thought by creation of new process.

# Why Multithreading ...

- When the server receives a new request, it creates a separate process to service that request. But this method is heavy weight.
- In fact this process creation method was common before threads become popular.
- Process creation is time consuming and resource intensive.
- It is generally more efficient for one process that contains multiple threads to serve the same purpose.
- This approach would multithread the web server process. The server would create a separate thread that would listen for clients requests.
- When a request is made by client, rather than creating another process, server will create a separate thread to service the request.

# Benefits of Multi-threading:

## **Responsiveness:**

- Multithreaded interactive application continues to run even if part of it is blocked or performing a lengthy operation, thereby increasing the responsiveness to the user.

## **Resource Sharing:**

- By default, threads share the memory and the resources of the process to which they belong.
- It allows an application to have several different threads of activity within the same address space.
- These threads running in the same address space do not need a context switch.

# Benefits of Multi-threading...

## Economy:

- Allocating memory and resources for each process creation is costly.
- Since thread shares the resources of the process to which they belong, it is more economical to create and context switch threads.
- Shorter context switching time. Less overhead than running several processes doing the same task.

## Utilization of multiprocessor architecture:

- The benefits of multi threading can be greatly increased in multiprocessor architecture, where threads may be running in parallel on different processors.
- Multithreading on a multi-CPU increases concurrency.

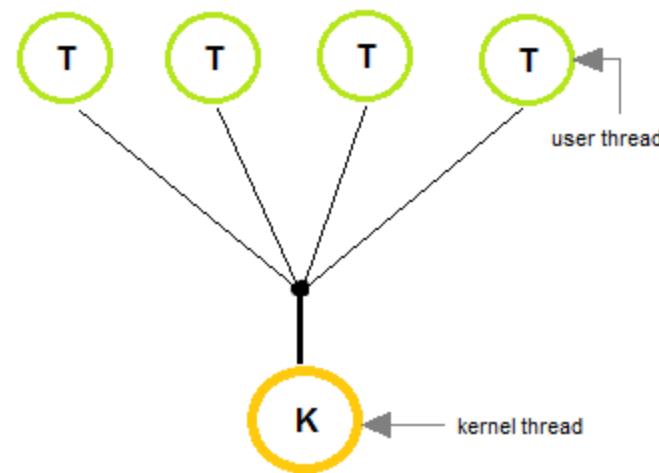
# Multithreading Model

The user threads must be mapped to kernel threads, by one of the following strategies:

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

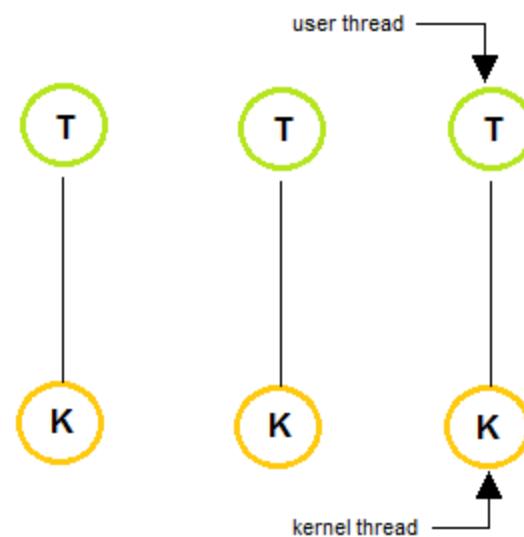
# Many to One Model

- In the **many to one** model, many user-level threads are all mapped onto a single kernel thread.
- Thread management is handled by the thread library in user space, which is efficient in nature.



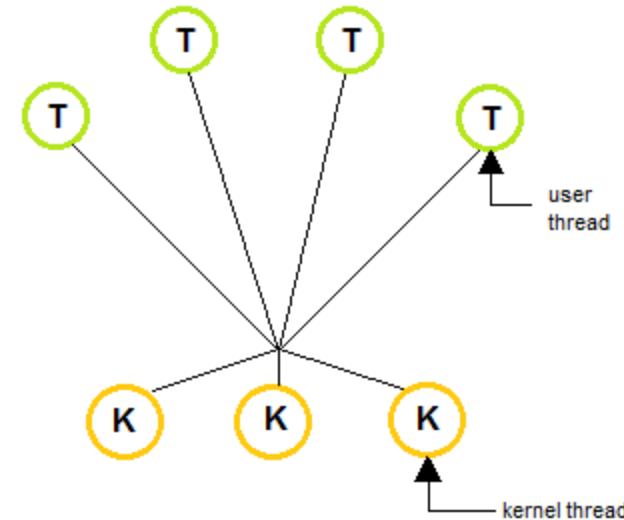
# One to One Model

- The **one to one** model creates a separate kernel thread to handle each and every user thread.
- Most implementations of this model place a limit on how many threads can be created.
- Linux and Windows from 95 to XP implement the one-to-one model for threads.



# Many to Many Model

- The **many to many** model multiplexes any number of user threads onto an equal or smaller number of kernel threads, combining the best features of the one-to-one and many-to-one models.
- Users can create any number of the threads.
- Blocking the kernel system calls does not block the entire process.
- Processes can be split across multiple processors.



# Inter Process Communication

- IPC is a mechanism that allows the exchange of data between processes.
- Processes frequently need to communicate with each other. For example, the output of the first process must be passed to the second process and so on.
- Thus there is a need for communication between the process, preferably in a well-structured way not using the interrupts.
- IPC enables one application to control another application, and for several applications to share the same data without interfering with one another.
- Inter-process communication (IPC) is a set of techniques for the exchange of data among multiple threads in one or more processes.
- Processes may be running on one or more computers connected by a network.
- Processes executing concurrently in the operating system may be either **independent process or co-operating process**

# Inter Process Communication

## **Independent process:**

- A process is independent if it can't affect or be affected by another process.

## **Co-operating Process:**

- A process is co-operating if it can affects other or be affected by the other process.
- Any process that shares data with other process is called co-operating process.

# Reasons for providing an environment for process co-operation:

## 1. Information sharing:

- Several users may be interested to access the same piece of information( for instance a shared file).
- We must allow concurrent access to such information.

## 2. Computation Speedup:

- To run the task faster we must breakup tasks into sub-tasks.
- Such that each of them will be executing in parallel to other, this can be achieved if there are multiple processing elements.

## 3. Modularity:

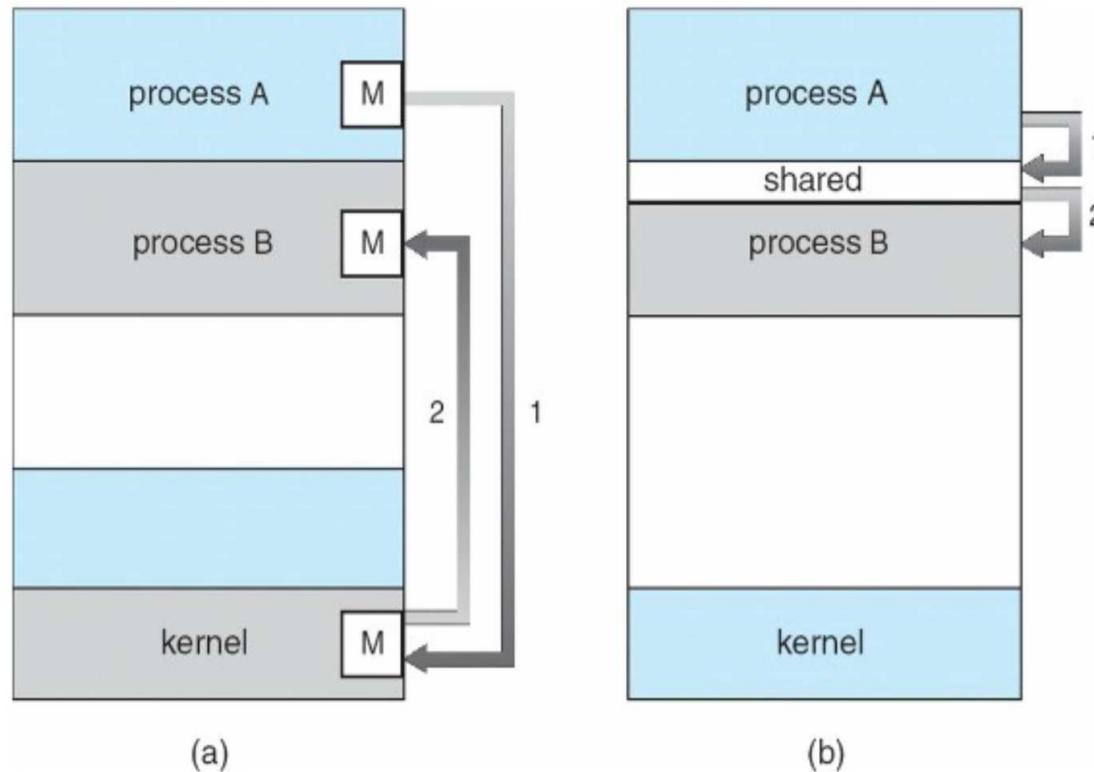
- construct a system in a modular fashion which makes easier to deal with individual.

## 4. convenience:

- Even an individual user may work on many tasks at the same time. For instance, a user may be editing, printing, and compiling in parallel.

There are two fundamental ways of IPC.

- a. Message Passing
- b. Shared Memory



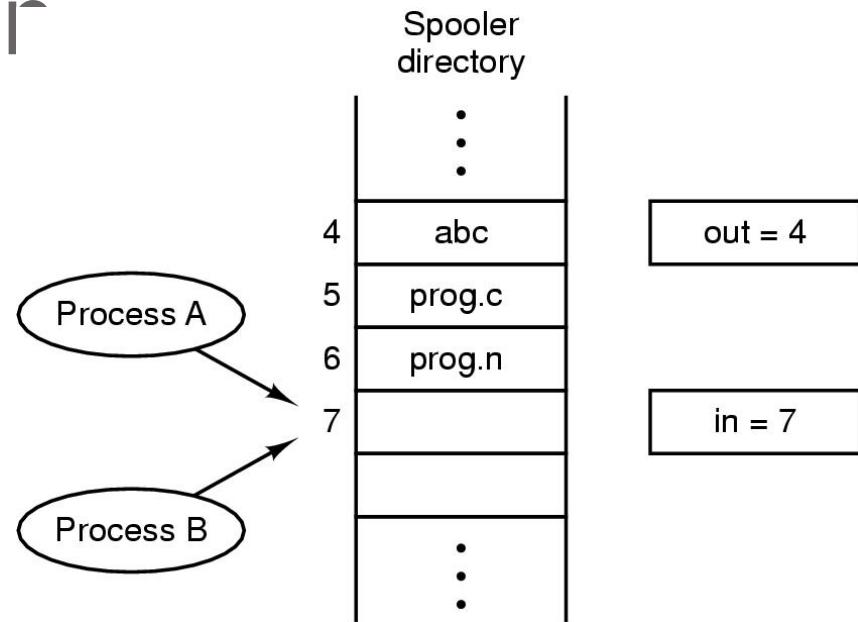
# Shared Memory:

- Here a region of memory that is shared by co-operating process is established.
- Process can exchange the information by reading and writing data to the **shared region**.
- Shared memory allows maximum speed and convenience of communication as it can be done at the speed of memory within the computer.
- System calls are required only to establish shared memory regions.
- Once shared memory is established no assistance from the kernel is required, all access are treated as routine memory access.

# Message Passing:

- Communication takes place by means of messages exchanged between the co-operating process
- Message passing is useful for exchanging the smaller amount of data.
- Easier to implement than shared memory.
- Slower than that of Shared memory as message passing system are typically implemented using system call
- Which requires more time consuming task of Kernel intervention.

# IPC: Race Condition



**Two processes want to access shared memory at same time**

## Race Condition

The situation where two or more processes are reading or writing some shared data, but not in proper sequence is called race Condition

# Race Condition:

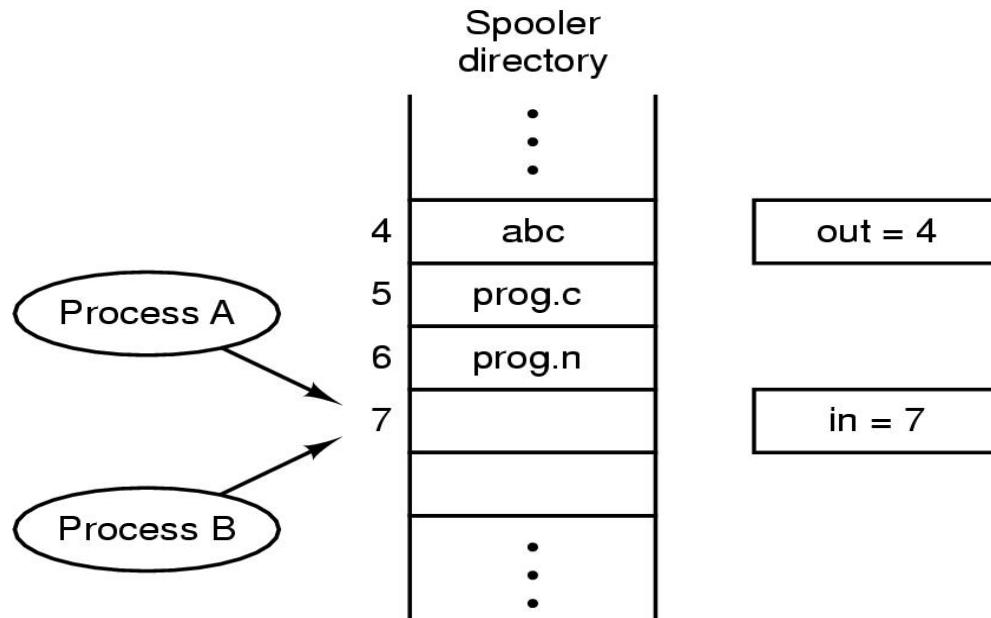
- The situation where 2 or more processes are reading or writing some shared data, but not in proper sequence is called **race Condition**.
- The final results depends on who runs precisely(accurately) when.

## example, a print spooler.

- When any process wants to print a file, it enters the file name in a special **spooler directory**.
- Another process, the **printer daemon, periodically checks to see if** there are any files to be printed, and if there are, it prints them and removes their names from the directory.

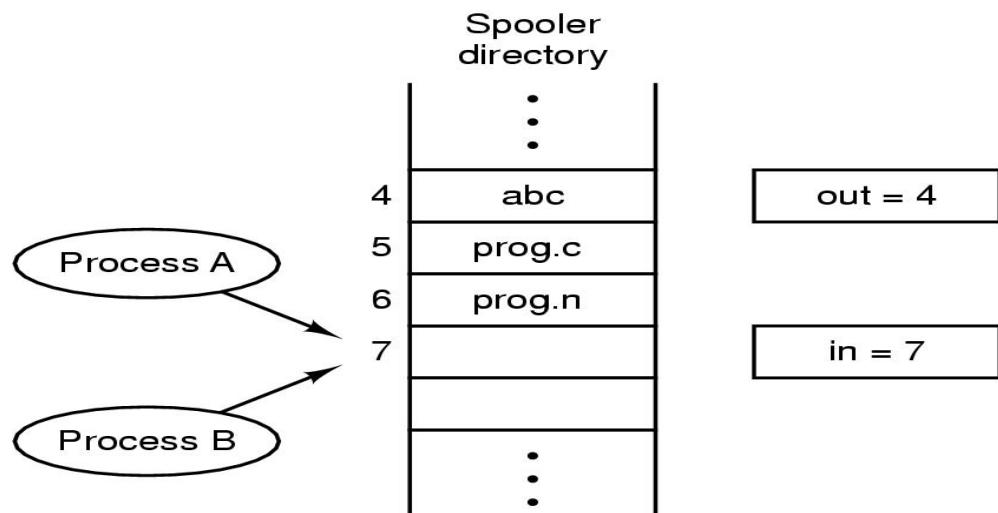
# Race Condition:

- Imagine that our spooler directory has a large number of slots, numbered 0, 1, 2, ..., each one capable of holding a file name. Also imagine that there are two shared variables,
- **out: which points to the next file to be printed**
- **in: which points to the next free slot in the directory.**



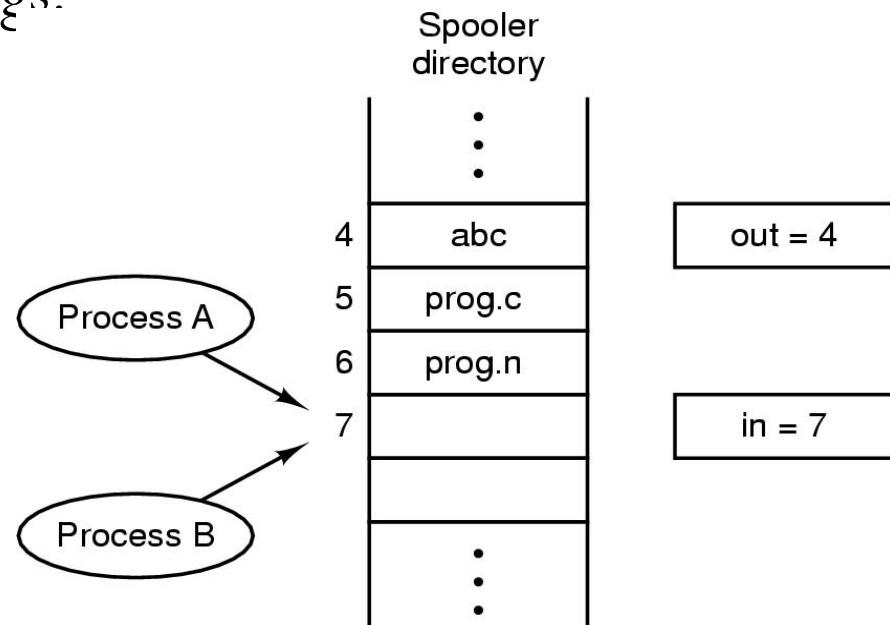
# Race Condition:

- At a certain instant, slots 0 to 3 are empty (the files have already been printed) and slots 4 to 6 are full (with the names of files to be printed).
- More or less simultaneously, processes A and B decide they want to queue a file for printing as shown in the fig.
- Process A reads in and stores the value, 7, in a local variable called **next\_free\_slot**.



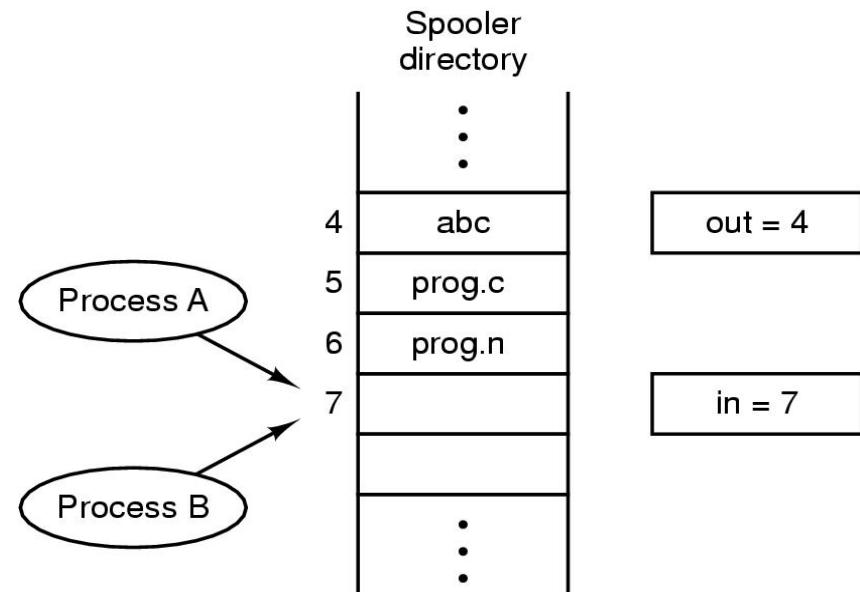
# Race Condition:

- Just then a clock interrupt occurs and the CPU decides that process A has run long enough, so it switches to process B.
- Process B also reads in, and also gets a 7, so it stores the name of its file in slot 7 and updates in to be an 8. Then it goes off and does other things.



# Race Condition

- Eventually, process A runs again, starting from the place it left off last time. It looks at **next\_free\_slot**, **finds a 7 there, and writes its file name** in slot 7, erasing the name that process B just put there.
- Then it computes **next\_free\_slot + 1**, which is 8, and sets **in to 8**.
- The spooler directory is now internally consistent, so the printer daemon will not notice anything wrong, but process B will never receive any output.



# Implementation Mutual Exclusion

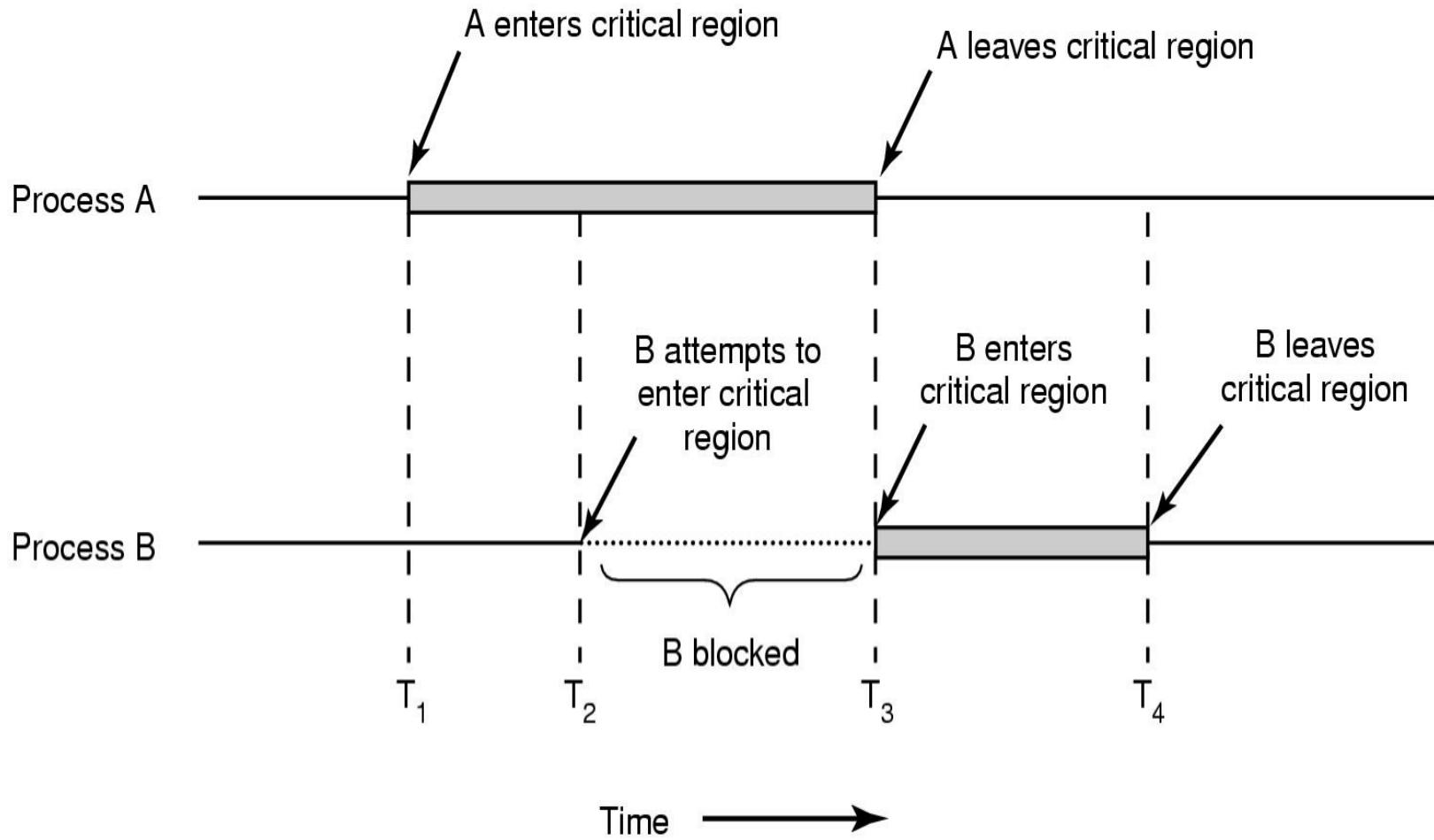
# IPC: Critical Regions and solution

The part of the program where the shared memory is accessed that must not be concurrently accessed by more than one processes is called the critical region

**Four conditions to provide mutual exclusion Or (Rules for avoiding Race Condition) Solution to Critical section problem:**

- No two processes simultaneously in critical region
- No assumptions made about speeds or numbers of CPUs
- No process running outside its critical region may block another process
- No process must wait forever to enter its critical region

# IPC: Critical Regions and solution



# IPC: Critical Regions and solution

## 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

# How to Manage Race Condition

## **Mutual Exclusion with Busy Waiting**

- Strict Alternation
- Peterson's Solution

## **Mutual Exclusion without Busy Waiting**

- Sleep and wakeup
- Semaphore
- Message Passing
- Monitor

# Strict Alternation

Turn=0

(a)

```
while (1)
    while(turn != 0) /* loop* */;
    critical_region();
    turn = 1;
    noncritical_region();
}
```

(a) Process 0.

(b)

```
{    while (1) { //repeat forever
    while(turn != 1) /* loop* */;
    critical_region();
    turn = 0;
    noncritical_region();
}
```

(b) Process 1.

A proposed solution to the critical region problem.

In both cases, be sure to note the semicolons terminating the while statements, this ensures waiting by a process.

- Integer variable turn is initially 0.
- It keeps track of whose turn it is to enter the critical region and examine or update the shared memory.
- Initially, process 0 inspects turn, finds it to be 0, and enters its critical region.
- Process 1 also finds it to be 0 and therefore sits in a tight loop continually testing turn to see when it becomes 1.
- Continuously testing a variable until some value appears is called **busy waiting**. It should usually be avoided, since it wastes CPU time.
- A lock that uses busy waiting is called a spin lock. When process 0 leaves the critical region, it sets turn to 1, to allow process 1 to enter its critical region.
- This way no two process can enter critical region simultaneously i.e mutual exclusion is fulfilled.

# Drawbacks:

- Taking turn is not a good idea when one of the process is much slower than other.
- This situation requires that two processes strictly alternate in entering their critical region.

## Example:

- Process 0 finishes the critical region it sets turn to 1 to allow process 1 to enter critical region.
- Suppose that process 1 finishes its critical region quickly so both process are in their non critical region with turn sets to 0.
- Process 0 executes its whole loop quickly, exiting its critical region & setting turn to 1.
- At this point turn is 1 and both processes are executing in their noncritical regions.

- Suddenly, process 0 finishes its noncritical region and goes back to the top of its loop.
- Unfortunately, it is not permitted to enter its critical region now since turn is 1 and process 1 is busy with its noncritical region.
- This situation violates the condition 3 set above: No process running outside the critical region may block other process.
- The concept of mutual exclusion is fulfilled here, but the concept of progress is not fulfilled.
- The process who is not interested to enter the critical section is blocking the next process to enter the critical section.

# Second attempt(algorithm 2)

- In this algorithm the variable turn is replaced by flag.

```
flag[0]=flag[1]=F; // Boolean value initially representing false.
```

```

while (1) {
    flag[0] = T;
    while(flag[1]) /* loop* */;
    critical_region();
    flag[0] = F;
    noncritical_region();
}
while (1) { //repeat forever
    flag[1]=T; // interested to enter c.s
    while(flag[0]) /* loop* */;
    critical_region();
    flag[1] = F;
    noncritical_region();
}

```

(a) Process 0.

(b) Process 1.

## Problem with this solution (2nd attempt)

- **Deadlock can occur if**

$\text{flag}[0]=\text{flag}[1]=T;$

**during the case of context switching, here initially process p0 makes its flag to TRUE after that context switches and p1 and makes its flag to TRUE**

- At this situation both the process have set their own value of flag to TRUE they both will go into loop forever as shown in the code segment above.

# Peterson's Solution:

By combination the idea of taking turns with the idea of lock variables and warning variables.:

```
while (1){ //repeat forever OR while(TRUE)
flag[0] =T; // interested to enter c.s
turn=1;
while(turn==1 && flag[1]==T) /* loop to give chance to other*/;
critical_region();
flag[0] =F;
noncritical_region();
}
```

(a) Process 0.

## Peterson's Solution:

```
while (1){ //repeat forever  
flag[1] =T; // interested to enter c.s  
turn=0;  
while(turn==0 && flag[0]==T) /* loop to give chance to other*/;  
critical_region();  
flag[1] =F;  
noncritical_region();  
}  
(b) Process 1.
```

# Another Example of Peterson's solution

```
... /* required header files */
#define FALSE 0
#define TRUE 1
#define N 2 /* number of processes */
int turn; /* whose turn is it? */
int interested[N] /* all elements initialized to FALSE */
void CS_entry (int process) /* process: Who is entering (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 ) { }; /* wait */
}
void CS_exit (int process) /* process: Who is leaving (0 or 1) ? */
{
    interested[process] = FALSE; /* indicate departure from CS */
}
```

Let us see how this solution works. Initially, neither process is in its CS. Now process 0 calls CS\_entry. It indicates its interest by setting its array element, and sets turn to 0. Since process 1 is not interested, CS\_entry returns immediately. If process 1 now calls CS\_entry, it will hang there until interested[0] goes to FALSE, an event that only happens when process 0 calls exit.

# The TSL (Test and Set Lock )Instruction

- Many computers, especially those designed with multiple processors in mind, have an instruction:

TSL RX,LOCK

- (Test and Set Lock) that works as follows.
- It reads the contents of the memory word *lock* into register RX and then stores a nonzero value at the memory address *lock*.
- The operations of reading the word and storing into it are guaranteed to be indivisible—no other **processor** can access the memory word until the instruction is finished.
- The CPU executing the TSL instruction locks the memory bus to prohibit other CPUs from accessing memory until it is done.

- When *lock* is 0, any process may set it to 1 using the TSL instruction and then read or write the shared memory. When it is done, the process sets *lock* back to 0 using an ordinary move instruction.

```
enter_region:  
    TSL REGISTER,LOCK      | copy lock to register and set lock to 1  
    CMP REGISTER,#0        | was lock zero?  
    JNE enter_region       | if it was non zero, lock was set, so loop  
    RET | return to caller; critical region entered  
  
leave_region:  
    MOVE LOCK,#0           | store a 0 in lock  
    RET | return to caller
```

# Sleep and Wakeup:

- Sleep and wakeup are system calls that blocks process instead of wasting CPU time when they are not allowed to enter their critical region.
- Sleep is a system call that causes the caller to block, that is, be suspended until another process wakes it up.
- The wakeup call has one parameter, the process to be awakened eg. wakeup(consumer) or wakeup(producer).

# Sleep and Wakeup

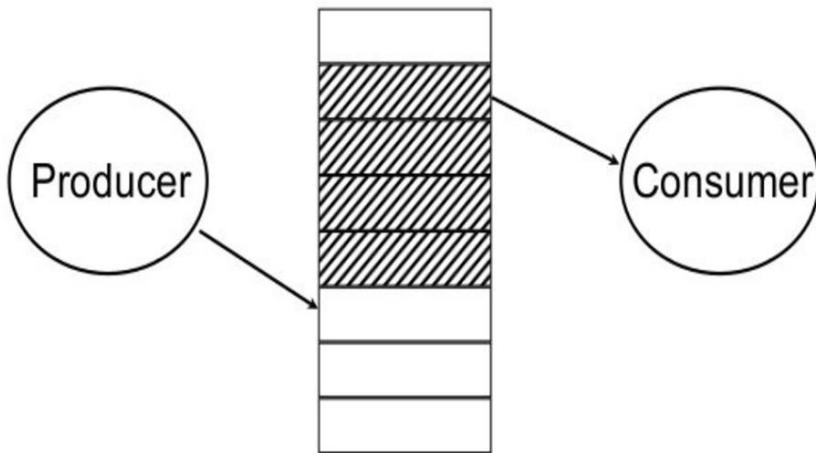
- Both Peterson's solution and the solution using TSL are correct, but both have the defect of requiring busy waiting.
- When a process wants to enter its critical region, it checks to see if the entry is allowed. If it is not allowed, the process just sits in a tight loop waiting until it is allowed.
- Beside of wasting CPU time, this approach can also have unexpected effects.
- Consider a computer with two processes,  $H$ , with high priority and  $L$ , with low priority.
- The scheduling rules are such that  $H$  runs whenever it is in ready state. At a certain moment, with  $L$  is in its critical region,  $H$  becomes ready to run.  $H$  now begins busy waiting. Before  $H$  is completed,  $L$  can not be scheduled. So  $L$  never gets the chance to leave its critical region, so  $H$  loops forever.

- Now let me look at some interprocess communication primitives that block processes when they are not allowed to enter their critical regions, instead of wasting CPU time in an empty loop.
- One of the simplest primitives is the pair **sleep** and **wakeup**. Sleep is a system call that causes the caller to block, the caller is suspended until another process wakes it up. The wakeup call has one parameter, the process to be awakened.

# Examples to use Sleep and Wakeup primitives:

## Producer-consumer problem (Bounded Buffer):

- Two processes share a common, fixed-size buffer.
- One of them, the producer, puts information into the buffer, and the other one, the consumer, takes it out



# Trouble arises when

1. The producer wants to put a new data in the buffer, but buffer is already full.

## Solution:

- Producer goes to sleep and to be awakened when the consumer has removed data.
- 
2. The consumer wants to remove data from the buffer but buffer is already empty.
- ## Solution:
- Consumer goes to sleep until the producer puts some data in buffer and wakes consumer up.

```
#define N 100 /* number of slots in the buffer */
int count = 0; /* number of items in the buffer */
void producer(void)
{
    int item;
    while (TRUE)
    { /* repeat forever */
        item = produce_item(); /* generate next item */
        if (count == N) sleep(); /* if buffer is full, go to sleep */
        insert_item(item); /* put item in buffer */
        count = count + 1; /* increment count of items in buffer */
        if (count == 1) wakeup(consumer); /* was buffer empty? ie.
                                         initially */
    }
}
```

```
void consumer(void)
{
    int item;
    while (TRUE)
    { /* repeat forever */
        if (count == 0) sleep(); /* if buffer is empty, got to sleep */
        item = remove_item(); /* take item out of buffer */
        count = count - 1; /* decrement count of items in buffer */
        if (count == N - 1) wakeup(producer); /* was buffer full? */
        consume_item(item); /*consume item */
    }
}
```

Count--> a variable to keep track of the no. of items in the buffer.  
N → Size of Buffer

## Producers code:

- The producers code is first test to see if count is N.
- If it is, the producer will go to sleep ; if it is not the producer will add an item and increment count.

## Consumer code:

- It is similar as of producer.
- First test count to see if it is 0. If it is, go to sleep; if it nonzero remove an item and decrement the counter.
- Each of the process also tests to see if the other should be awakened and if so wakes it up.
- This approach sounds simple enough, but it leads to the same kinds of **race conditions as we saw in the spooler directory.**

## Lets see how the race condition arises

1. The buffer is empty and the consumer has just read count to see if it is 0.
2. At that instant, the scheduler decides to stop running the consumer temporarily and start running the producer.  
(Consumer is interrupted and producer resumed)
3. The producer creates an item, puts it into the buffer, and increases count.
4. Because the buffer was empty prior to the last addition (count was just 0), the producer tries to wake up the consumer.

5. Unfortunately, the consumer is not yet logically asleep, so the **wakeup signal is lost**.
  6. When the consumer next runs, it will test the value of count it previously read, find it to be 0, and go to sleep.
  7. Sooner or later the producer will fill up the buffer and also go to sleep. Both will sleep forever.
- 
- The problem here is that a wakeup sent to a process that is not (yet) sleeping is lost.

# Semaphores

- Semaphore is an integer variable to count the number of wakeup processes saved for future use.
- A semaphore could have the value 0, indicating that no wakeup processes were saved, or some positive value if one or more wakeups were pending.
- There are two operations, down (wait) and up(signal) (generalizations of sleep and wakeup, respectively).
- The down operation on a semaphore checks if the value is greater than 0. If so, it decrements the value and just continues.
- If the value is 0, the process is put to sleep without completing the down for the moment.

- The up operation increments the value of the semaphore addressed. If one or more processes were sleeping on that semaphore, unable to complete an earlier down operation, one of them is chosen by the system (e.g., at random) and is allowed to complete its down.
- Checking the value, changing it and possibly going to sleep is as a single indivisible **atomic action**. It is guaranteed that when a semaphore operation has started, no other process can access the semaphore until the operation has completed or blocked.

# Atomic operations:

- When one process modifies the semaphore value, no other process can simultaneously modify that same semaphore value.
- In addition, in case of the P(S) operation the testing of the integer value of S ( $S \leq 0$ ) and its possible modification ( $S = S - 1$ ), must also be executed without interruption.
- Modification to the integer value of the semaphore in the wait  $\{p(s)\}$  and signal  $\{V(s)\}$  operation must be executed indivisibly(only one process can modify the same semaphore value at a time)

# Semaphore operations:

- P or Down, or Wait: **P stands for proberen ("to test")**
- V or Up or Signal: Dutch words. **V stands for verhogen ("increase")**

## **wait(sem)**

- decrement the semaphore value. if negative, suspend the process and place in queue. (Also referred to as *P()*, *down in literature.*)

## **signal(sem)**

- increment the semaphore value, allow the first process in the queue to continue. (Also referred to as *V()*, *up in literature.*)

## Counting semaphore

- integer value can range over an unrestricted domain

## Binary semaphore

- integer value can range only between 0 and 1
- can be simpler to implement Also known as mutex locks

## Provides mutual exclusion

- **Semaphore gives solution to n process.**

Semaphore S; // initialized to 1

Do {

    wait (S);

        CriticalSection

    signal (S);

} while(T)

## Implementation of wait:

```
wait(s)
{
    while(s<=0); // do nothing
    s=s-1;
}
```

## Implementation of signal:

```
signal(s)
{
    s=s+1;
}
```

# The producer-consumer problem using semaphores.

```
#define N 100 /* number of slots in the buffer */  
typedef int semaphore; /* semaphores are a special kind of int */  
semaphore mutex = 1; /* controls access to critical region */  
semaphore empty = N; /* counts empty buffer slots */  
semaphore full = 0; /* counts full buffer slots */
```

```
void producer(void)
{
    int item;
    while (TRUE){ /*TRUE is the constant 1 */
        item = produce_item(); /* generate something to put in buffer */
        down(&empty); /* decrement empty count */
        down(&mutex); /* enter critical region */
        insert_item(item); /* put new item in buffer */
        up(&mutex); /* leave critical region */
        up(&full); /* increment count of full slots */
    }
}
```

```
void consumer(void)
{
    int item;
    while (TRUE){ /* infinite loop */
        down(&full); /* decrement full count */
        down(&mutex); /* enter critical region */
        item = remove_item(); /* take item from buffer */
        up(&mutex); /* leave critical region */
        up(&empty); /* increment count of empty slots */
        consume_item(item); /* do something with the item */
    }
}
```

This solution uses three semaphore.

1. Full:

For counting the number of slots that are full, initially 0

2. Empty:

For counting the number of slots that are empty, initially equal to the no. of slots in the buffer.

3. Mutex:

To make sure that the producer and consumer do not access the buffer at the same time, initially 1.

Here in this example semaphores are used in two different ways.

## 1. For mutual Exclusion:

- The mutex semaphore is for mutual exclusion.
- It is designed to guarantee that only one process at a time will be reading or writing the buffer and the associated variable.

## 2. For synchronization:

- Ensures who can work when and who have to stop when
- The full and empty semaphores are needed to guarantee that certain event sequences do or do not occur.
- In this case, they ensure that producer stops running when the buffer is full and the consumer stops running when it is empty.
- The above definition of the semaphore suffer the problem of busy wait.

# Advantages of semaphores

- Processes do not busy wait while waiting for resources.
- While waiting, they are in a "suspended" state, allowing the CPU to perform other work.
- Works on (shared memory) multiprocessor systems.
- User controls synchronization.

# Disadvantage of semaphores

- can only be invoked by processes--not interrupt service routines because interrupt routines cannot block
- user controls synchronization--could mess up.

# Solving the Producer-Consumer Problem using Semaphores

- Semaphores solve the lost-wakeup problem. It is essential that they are implemented in an indivisible way. The normal way is to implement up and down as system calls, with the operating system briefly disabling all interrupts while it is testing the semaphore, updating it, and putting the process to sleep.
- If multiple CPUs are being used, each semaphore should be protected by a lock variable, with the TSL instruction used to make sure that only one CPU at a time examines the semaphore.

- This solution uses three semaphores: one called *full* for counting the number of slots that are full, one called *empty* for counting the number of slots that are empty, and one called *mutex* to make sure the producer and consumer do not access the buffer at the same time. *Full* is initially 0, *empty* is initially equal to the number of slots in the buffer (N), and *mutex* is initially 1.
- The *mutex* semaphore is used for mutual exclusion. It is designed to guarantee that only one process at a time will be reading or writing the buffer and the associated variables. The other use of semaphores is for **synchronization**. The *full* and *empty* semaphores are used to guarantee synchronization. In this case, they ensure that the producer stops running when the buffer is full, and the consumer stops running when it is empty.

```
#define N 100
```

<https://genuineNotes.com>

```
/* number of slots in the buffer */

typedef int semaphore;          /* semaphores are a special kind of int */
semaphore mutex = 1;            /* controls access to critical region */
semaphore empty = N;           /* counts empty buffer slots */
semaphore full = 0;            /* counts full buffer slots */

void producer(void)
{
    int item;

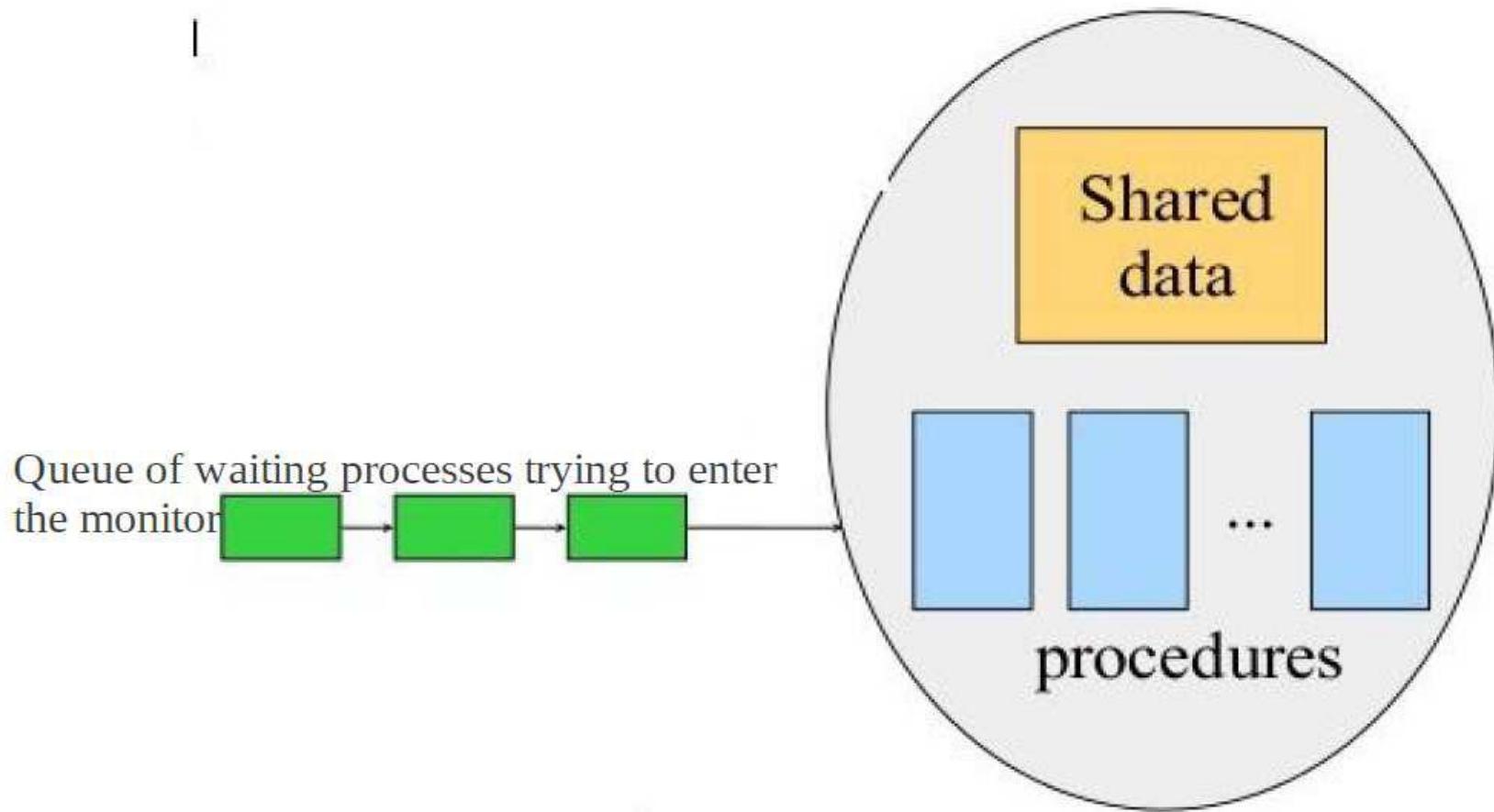
    while (TRUE) {                  /* TRUE is the constant 1 */
        item = produce_item();      /* generate something to put in buffer */
        down(&empty);              /* decrement empty count */
        down(&mutex);              /* enter critical region */
        insert_item(item);          /* put new item in buffer */
        up(&mutex);                /* leave critical region */
        up(&full);                 /* increment count of full slots */
    }
}

void consumer(void)
{
    int item;

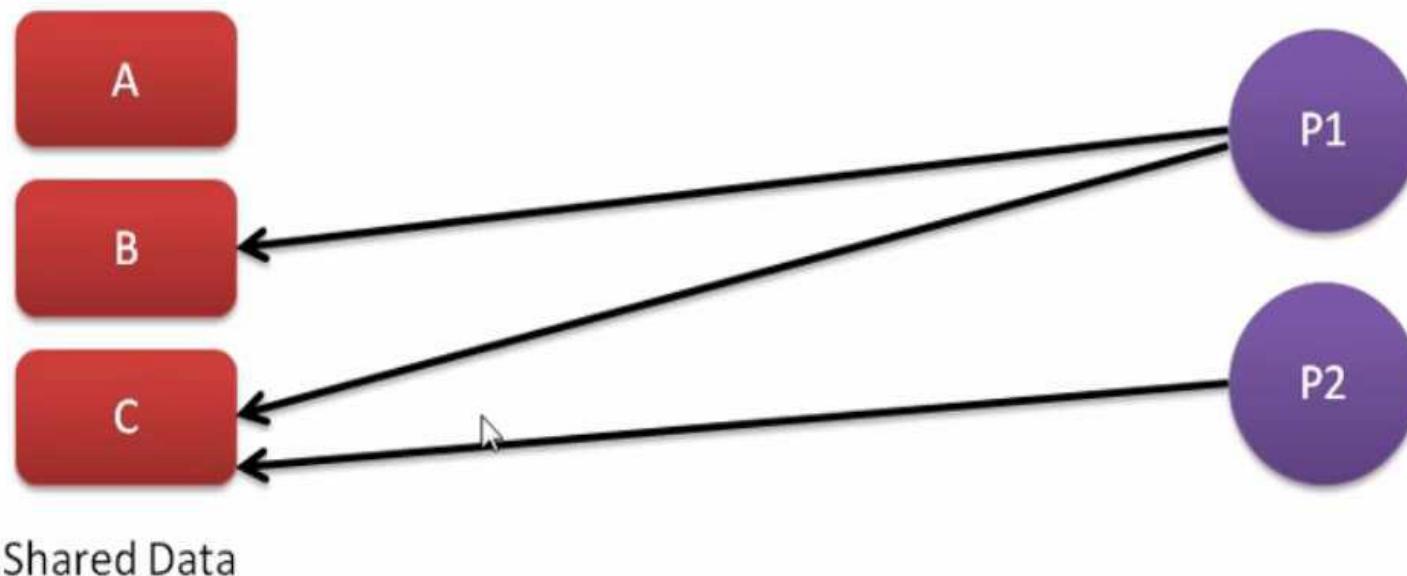
    while (TRUE) {                  /* infinite loop */
        down(&full);               /* decrement full count */
        down(&mutex);              /* enter critical region */
        item = remove_item();       /* take item from buffer */
        up(&mutex);                /* leave critical region */
        up(&empty);                /* increment count of empty slots */
        consume_item(item);         /* do something with the item */
    }
}
```

# Monitors:

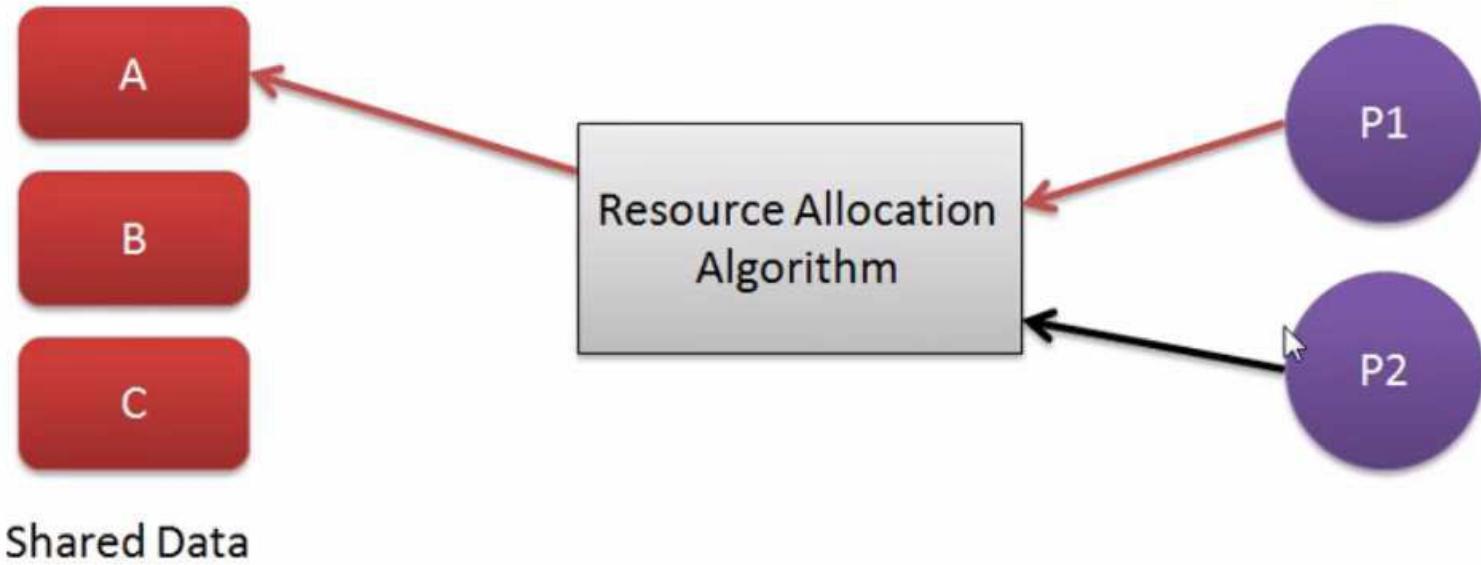
- In concurrent programming, a **monitor is an object or module** intended to be used safely by more than one thread.
- The defining characteristic of a monitor is that its methods are executed with mutual exclusion.
- That is, at each point in time, at most one thread may be executing any of its methods.
- Monitors also provide a mechanism for threads to temporarily give up exclusive access, it has to wait for some condition to be met, before regaining exclusive access and resuming their task.
- Monitors also have a mechanism for signaling other threads that such conditions have been met.



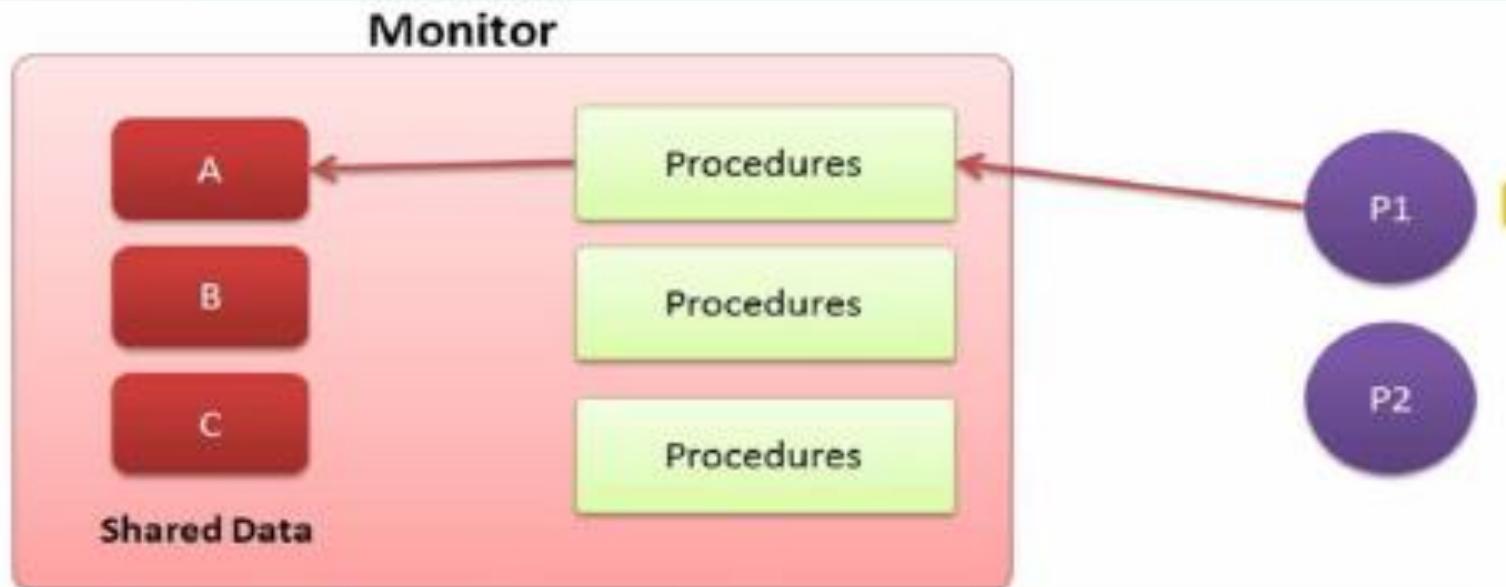
- A higher level synchronization primitive.
- A monitor is a collection of procedures, variables, and data structures that are all grouped together in a special kind of module or package.
- Processes may call the procedures in a monitor whenever they want to, but they cannot directly access the monitor's internal data structures from procedures declared outside the monitor.
- This rule, which is common in modern object-oriented languages such as Java.



When multiple processes access shared data simultaneously , create problem of race condition



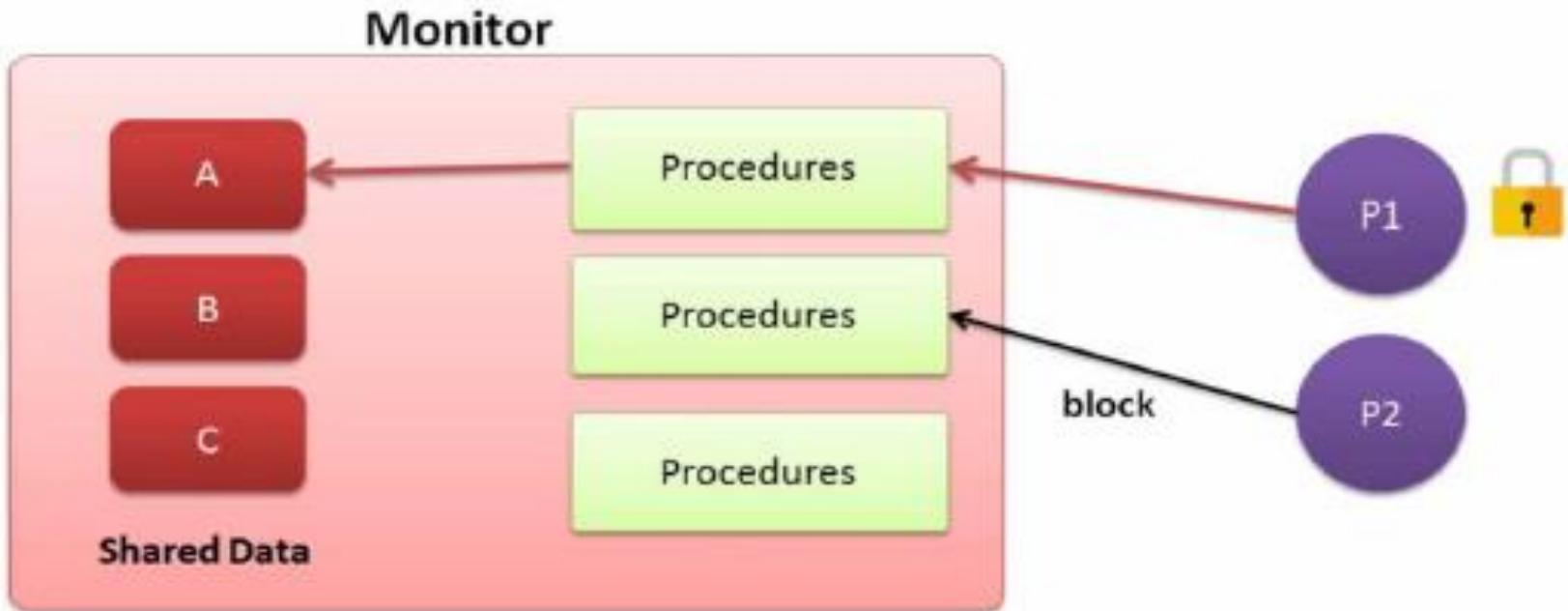
# Monitor



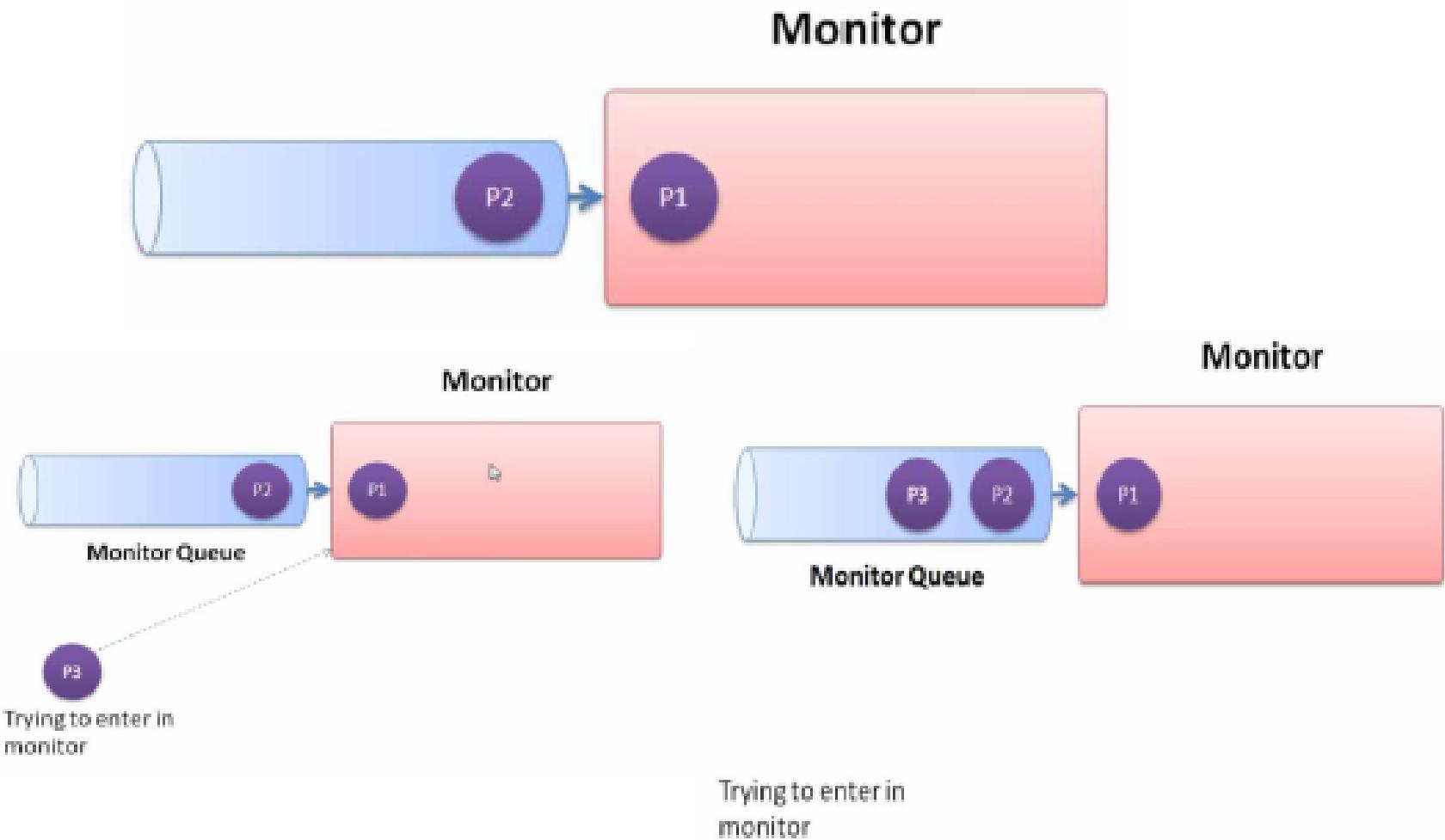
**A monitor is a module that encapsulates**

- Shared data structures
- Procedures that operate on the shared data
- Synchronization between concurrent procedure invocation

# Monitor



Only one thread can enter in monitor at any time.



# Monitor

```
Monitor account
{
    double balance;

    withdraw(amount)
    {
        balance = balance - amount;
        return balance;
    }
}
```



# Monitor

T1

Withdraw(w)

b balance=balance - w

Thread Switch -> T2

return balance

T2

Withdraw(a)

Blocked , switch -> T3

resume

T3

Withdraw(b)

Blocked, switch -> T1

resume

# Message Passing

- Interprocess communication uses two primitives, send and receive.
- They can easily be put into library procedures, such as  
*send(destination, &message);*  
*receive(source, &message);*
- The former call sends a message to a given destination and the latter one receives a message from a given source.
- If no message is available, the receiver can block until one arrives. Alternatively, it can return immediately with an error code.

## Message passing systems have many problems,

- Especially if the communicating processes are on different machines connected by a network. For example, messages can be lost on the network.
- To solve this problem, as soon as a message has been received, the receiver will send back a special **acknowledgement** message.
- If the sender has not received the acknowledgement within a certain time interval, it retransmits the message.
- Now consider what happens if the message itself is received correctly, but the acknowledgement is lost. The sender will retransmit the message, so the receiver will get it twice.
- It is essential that the receiver is able to distinguish a new message from the retransmission of an old one.

- Usually, this problem is solved by putting consecutive sequence numbers in each original message.
- If the receiver gets a message bearing the same sequence number as the previous message, it knows that the message is a duplicate that can be ignored.
- There are also design issues that are important when the sender and receiver are on the same machine. One of these is performance. Copying messages from one process to another is always slower than doing a semaphore operation.

# The Producer-Consumer Problem with Message Passing

- We assume that all messages are the same size and that messages sent but not yet received are buffered automatically by the operating system.
- In this solution, a total of  $N$  messages are used, analogous to the  $N$  slots in a shared memory buffer.
- The consumer starts out by sending  $N$  empty messages to the producer. Whenever the producer has an item to give to the consumer, it takes an empty message and sends back a full one.
- If the producer works faster than the consumer, all the messages will be full, waiting for the consumer: the producer will be blocked, waiting for an empty to come back.
- If the consumer works faster, all the messages will be empty waiting for the producer to fill them up: the consumer will be blocked, waiting for a full message.

```
#define N 100      /* number of slots in the buffer */
void producer(void)
{
    int item;
    message m;      /* message buffer */

    while (TRUE) {
        item = produce_item( );          /* generate something to put in buffer */
        receive(consumer, &m);          /* wait for an empty to arrive */
        build_message (&m, item);       /* construct a message to send */
        send(consumer, &m);            /* send item to consumer */
    }
}

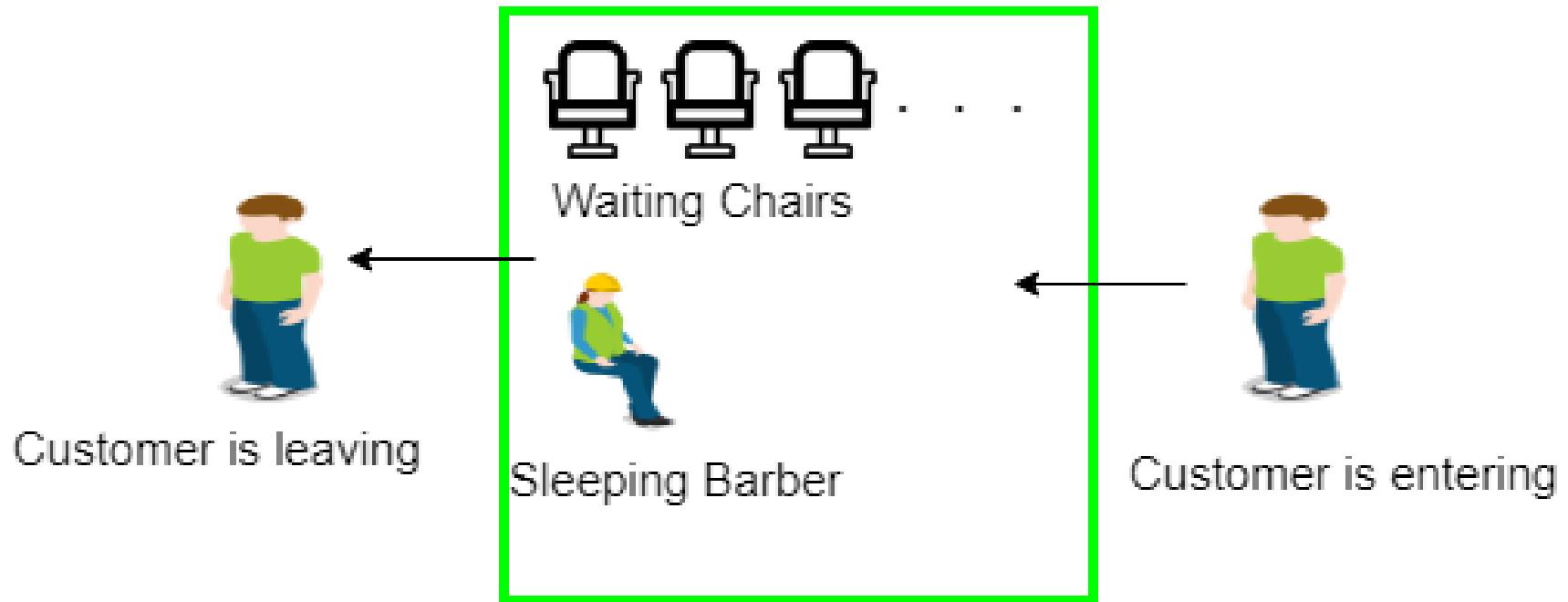
void consumer(void) {
    int item, i;
    message m;

    for (i = 0; i < N; i++) send(producer, &m); /* send N empties */
    while (TRUE) {
        receive(producer, &m);          /* get message containing item */
        item = extract_item(&m);        /* extract item from message */
        send(producer, &m);            /* send back empty reply */
        consume_item(item);           /* do something with the item */
    }
}
```

# Sleeping Barber problem

**Problem :** The analogy is based upon a hypothetical barber shop with one barber. There is a barber shop which has one barber, one barber chair, and n chairs for waiting for customers if there are any to sit on the chair.

- If there is no customer, then the barber sleeps in his own chair.
- When a customer arrives, he has to wake up the barber.
- If there are many customers and the barber is cutting a customer's hair, then the remaining customers either wait if there are empty chairs in the waiting room or they leave if no chairs are empty.



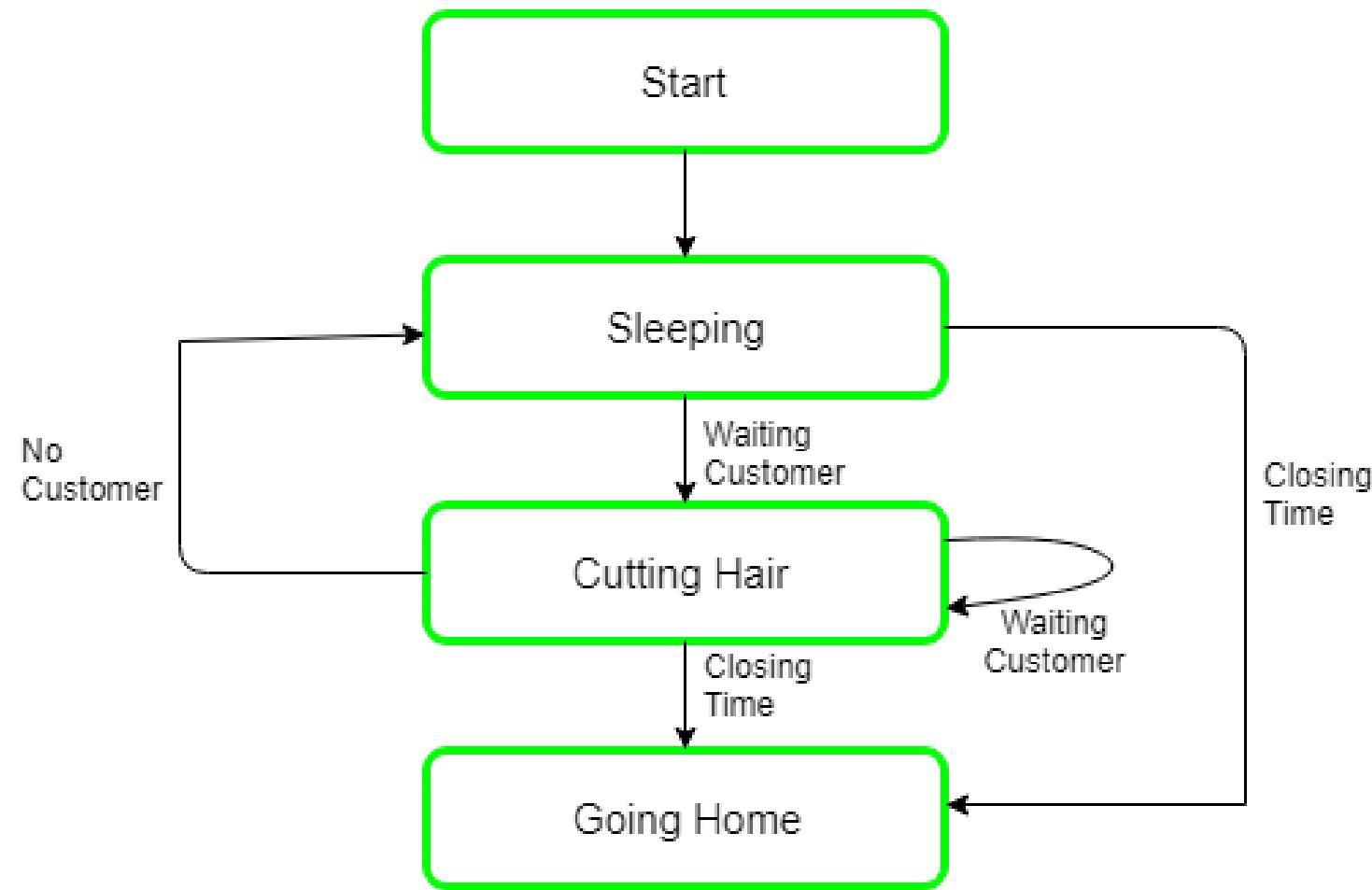
## Solution :

The solution to this problem includes three semaphores.

- First is for the customer which counts the number of customers present in the waiting room (customer in the barber chair is not included because he is not waiting).
- Second, the barber 0 or 1 is used to tell whether the barber is idle or is working, And the third mutex is used to provide the mutual exclusion which is required for the process to execute.
- In the solution, the customer has the record of the number of customers waiting in the waiting room if the number of customers is equal to the number of chairs in the waiting room then the upcoming customer leaves the barbershop.

- When the barber shows up in the morning, he executes the procedure `barber`, causing him to block on the semaphore `customers` because it is initially 0. Then the barber goes to sleep until the first customer comes up.
- When a customer arrives, he executes `customer` procedure the customer acquires the `mutex` for entering the critical region, if another customer enters thereafter, the second one will not be able to anything until the first one has released the `mutex`.
- The customer then checks the chairs in the waiting room if waiting customers are less then the number of chairs then he sits otherwise he leaves and releases the `mutex`.

- If the chair is available then customer sits in the waiting room and increments the variable waiting value and also increases the customer's semaphore this wakes up the barber if he is sleeping.
- At this point, customer and barber are both awake and the barber is ready to give that person a haircut. When the haircut is over, the customer exits the procedure and if there are no customers in waiting room barber sleeps.



# Algorithm for Sleeping Barber problem:

```
Semaphore Customers = 0;
```

```
Semaphore Barber = 0;
```

```
Mutex Seats = 1;
```

```
int FreeSeats = N;
```

```
Barber {
```

```
    while(true) { /* waits for a customer (sleeps). */
```

```
        down(Customers); /* mutex to protect the number of available seats.*/
```

```
        down(Seats); /* a chair gets free.*/
```

```
        FreeSeats++; /* bring customer for haircut.*/
```

```
        up(Barber); /* release the mutex on the chair.*/
```

```
        up(Seats); /* barber is cutting hair.*/
```

```
}
```

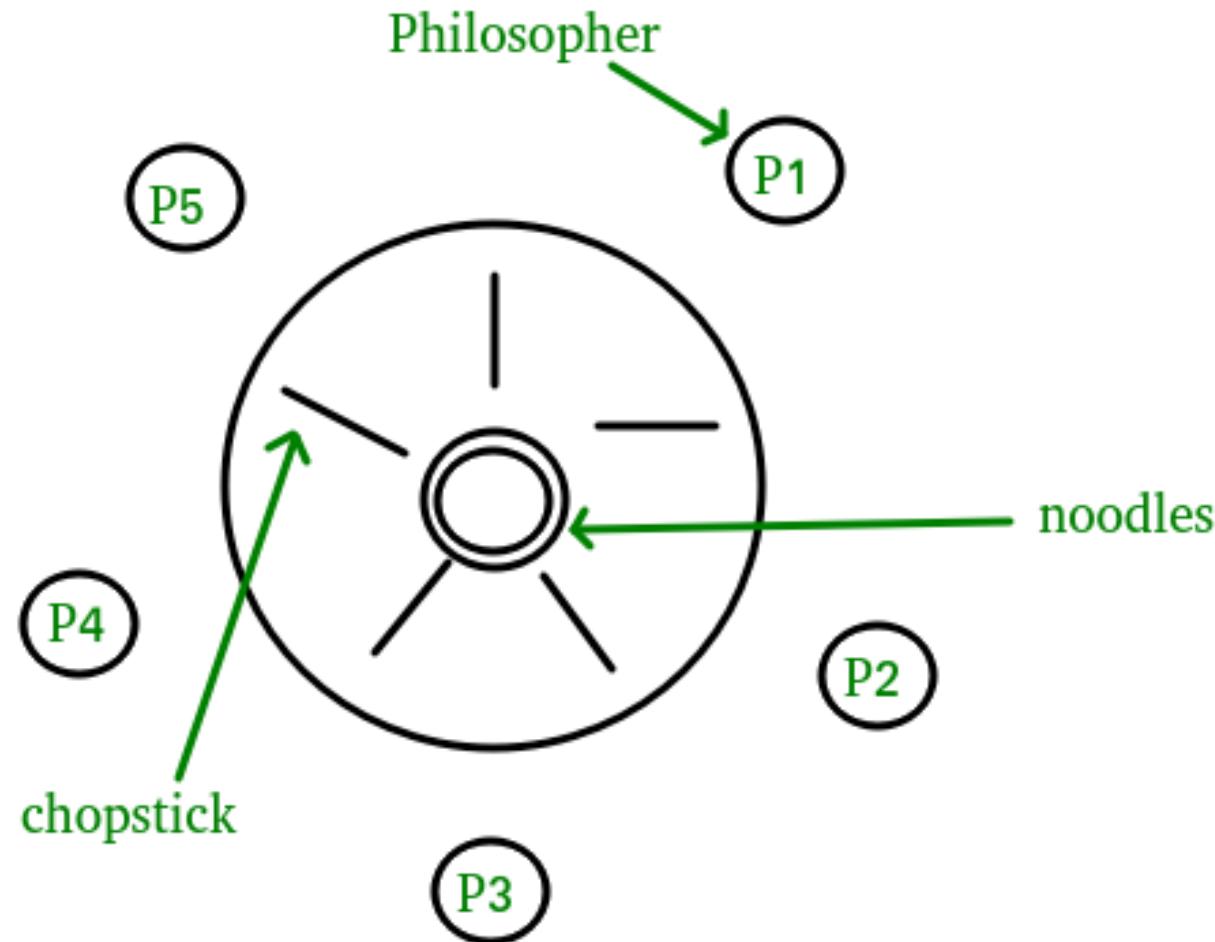
```
}
```

```
Customer {  
    while(true) { /* protects seats so only 1 customer tries to sit  
    in a chair if that's the case.*/  
        down(Seats); //This line should not be here.  
        if(FreeSeats > 0) { /* sitting down.*/  
            FreeSeats--; /* notify the barber. */  
            up(Customers); /* release the lock */  
            up(Seats); /* wait in the waiting room if barber is busy. */  
            down(Barber); // customer is having hair cut  
        } else { /* release the lock */  
            up(Seats); // customer leaves  
        }  
    }  
}
```

# Dining Philosopher Problem

- The Dining Philosopher Problem states that N philosophers seated around a circular table with one chopstick between each pair of philosophers.
- There is one chopstick between each philosopher. A philosopher may eat if he can pickup the two chopsticks adjacent to him. One chopstick may be picked up by any one of its adjacent followers but not both.

# Dining Philosopher Problem



# Dining Philosopher Problem

- There is one chopstick between each philosopher
- A philosopher must pick up its two nearest chopsticks in order to eat
- A philosopher must pick up first one chopstick, then the second one, not both at once

# Dining Philosopher Problem

- We need an algorithm for allocating these limited resources(chopsticks) among several processes(phiosophers) such that solution is free from deadlock and free from starvation.
- There exist some algorithm to solve Dining – Philosopher Problem, but they may have deadlock situation. Also, a deadlock-free solution is not necessarily starvation-free. Semaphores can result in deadlock due to programming errors. Monitors alone are not sufficiency to solve this, we need monitors with *condition variables*

# Dining Philosopher Problem

## Monitor-based Solution to Dining Philosophers

We illustrate monitor concepts by presenting a deadlock-free solution to the dining-philosophers problem. Monitor is used to control access to state variables and condition variables. It only tells when to enter and exit the segment. This solution imposes the restriction that a philosopher may pick up her chopsticks only if both of them are available.

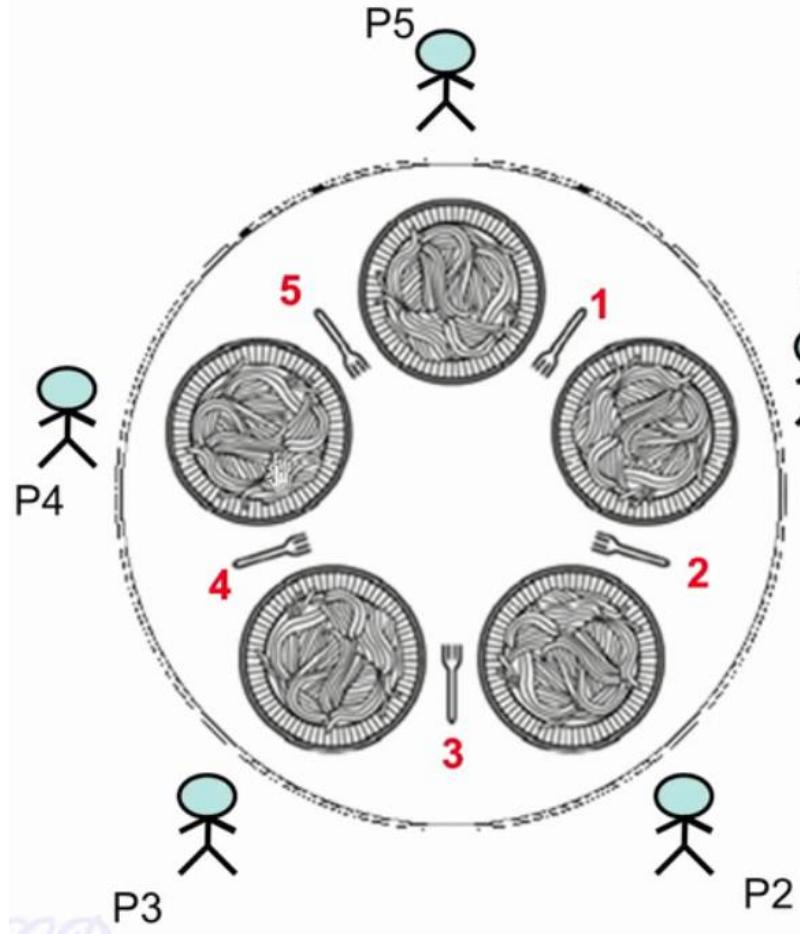
# Dining Philosopher Problem

To code this solution, we need to distinguish among three states in which we may find a philosopher. For this purpose, we introduce the following data structure:

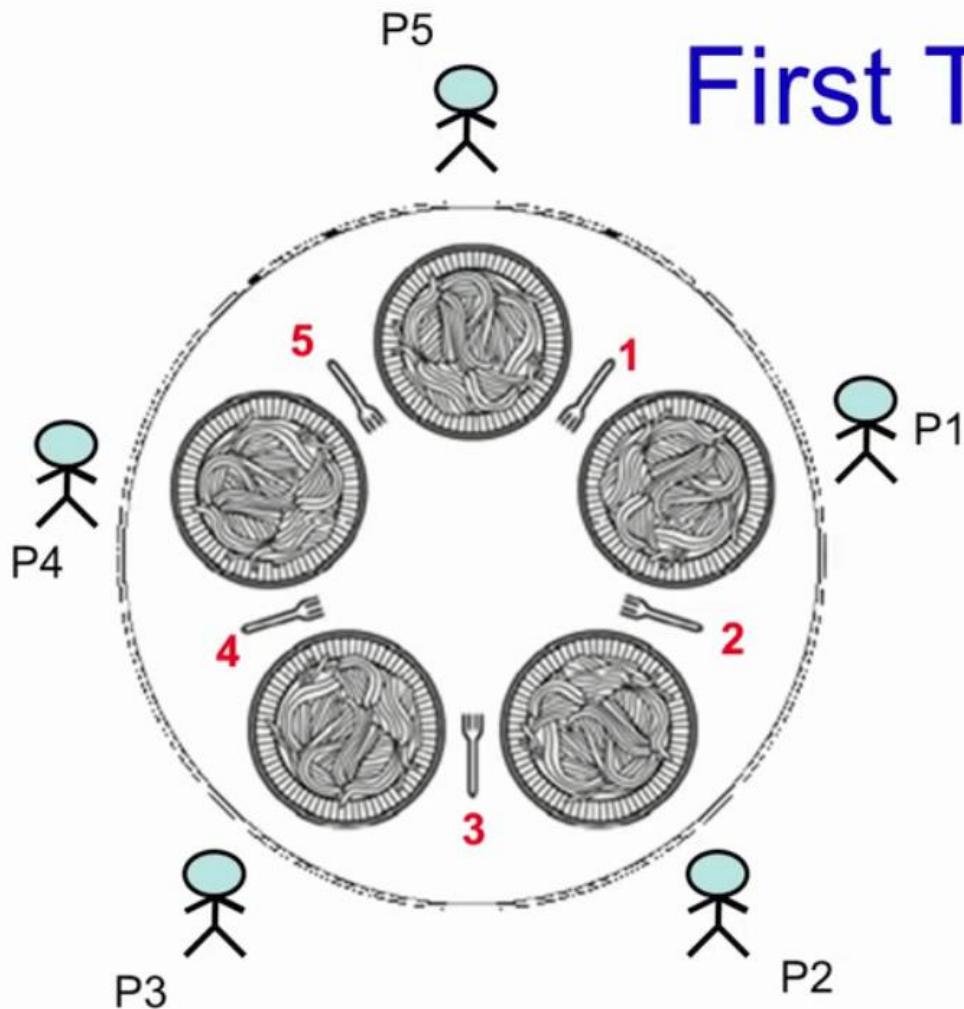
- **THINKING** – When philosopher doesn't want to gain access to either fork.
- **HUNGRY** – When philosopher wants to enter the critical section.
- **EATING** – When philosopher has got both the forks, i.e., he has entered the section.

Philosopher  $i$  can set the variable  $\text{state}[i] = \text{EATING}$  only if her two neighbors are not eating ( $\text{state}[(i+4) \% 5] \neq \text{EATING}$ ) and ( $\text{state}[(i+1) \% 5] \neq \text{EATING}$ ).

# Dining Philosophers Problem



- Philosophers either think or eat
- To eat, a philosopher needs to hold both forks (the one on his left and the one on his right)
- If the philosopher is not eating, he is thinking.
- **Problem Statement :** Develop an algorithm where no philosopher starves.

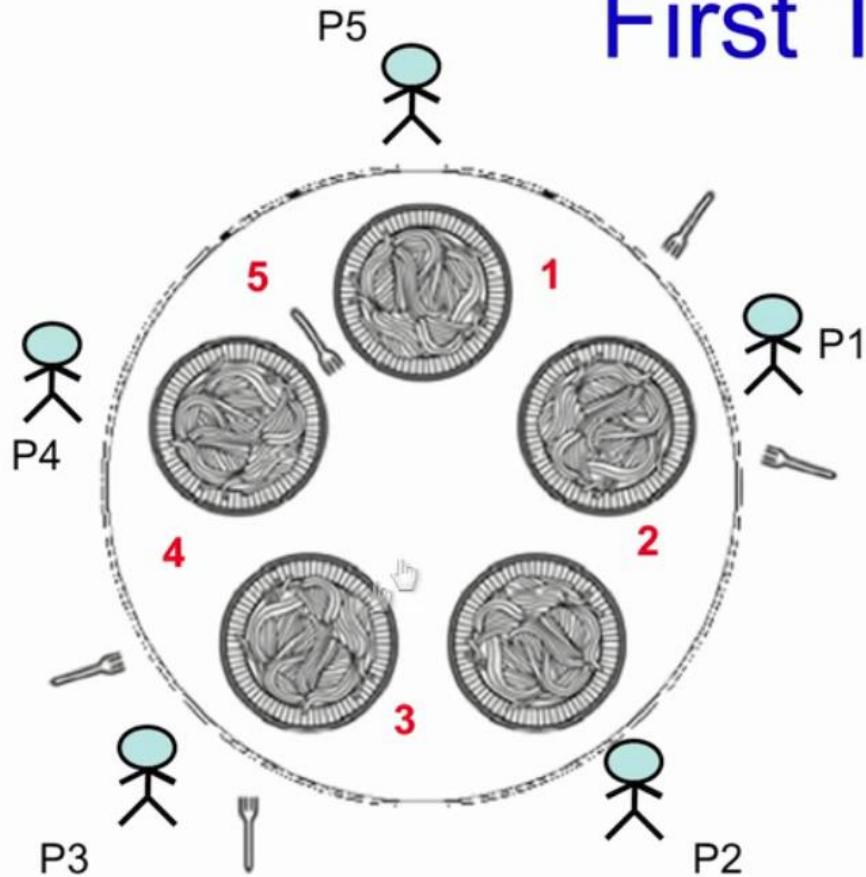


# First Try

```
#define N 5
```

```
void philosopher(int i){  
    while(TRUE){  
        think(); // for some_time  
        take_fork(Ri);  
        take_fork(Li);  
        eat();  
        put_fork(Li);  
        put_fork(Ri);  
    }  
}
```

## First Try

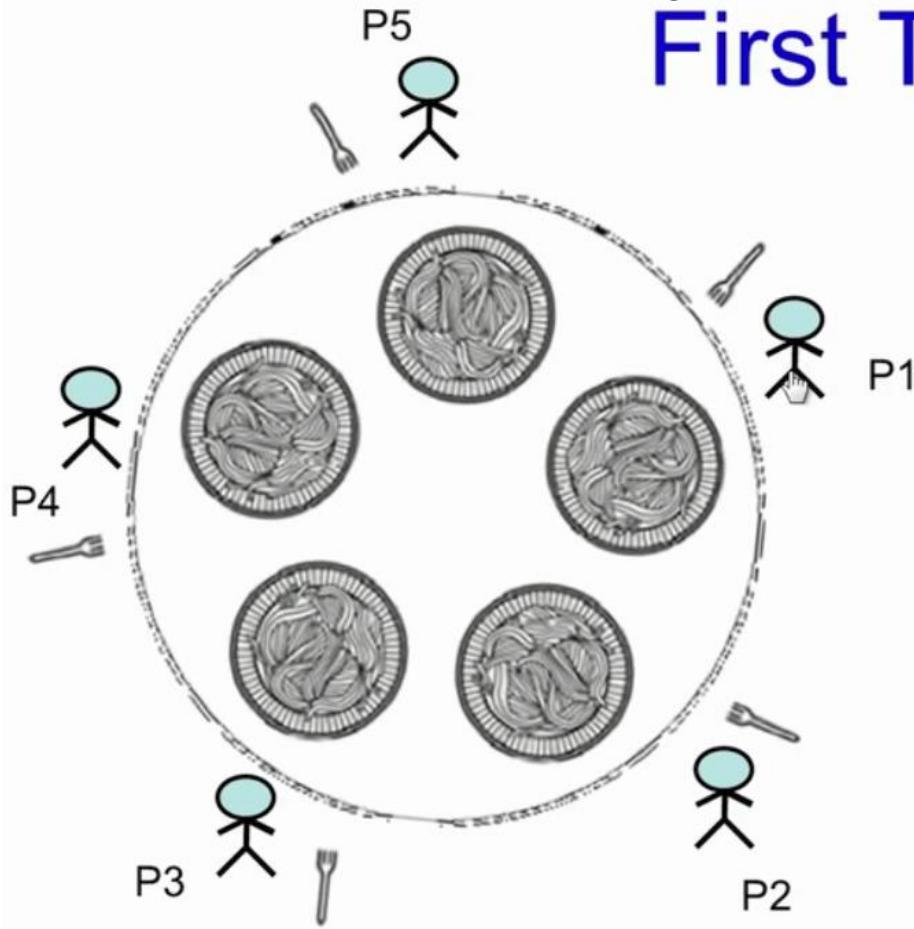


```
#define N 5
```

```
void philosopher(int i){  
    while(TRUE){  
        think(); // for some_time  
        take_fork(Ri);  
        take_fork(Li);  
        eat();  
        put_fork(Li);  
        put_fork(Ri);  
    }  
}
```

What happens if only philosophers P1 and P3 are always given the priority?  
P4, P5, and P2 starves... so scheme needs to be fair

# First Try



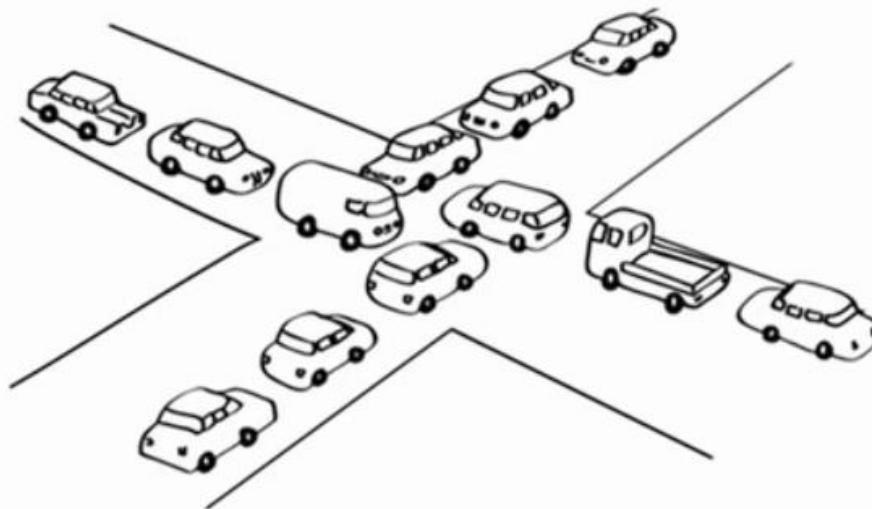
```
#define N 5
```

```
void philosopher(int i){  
    while(TRUE){  
        think(); // for some_time  
        take_fork(Ri);  
        take_fork(Li);  
        eat();  
        put_fork(Li);  
        put_fork(Ri);  
    }  
}
```

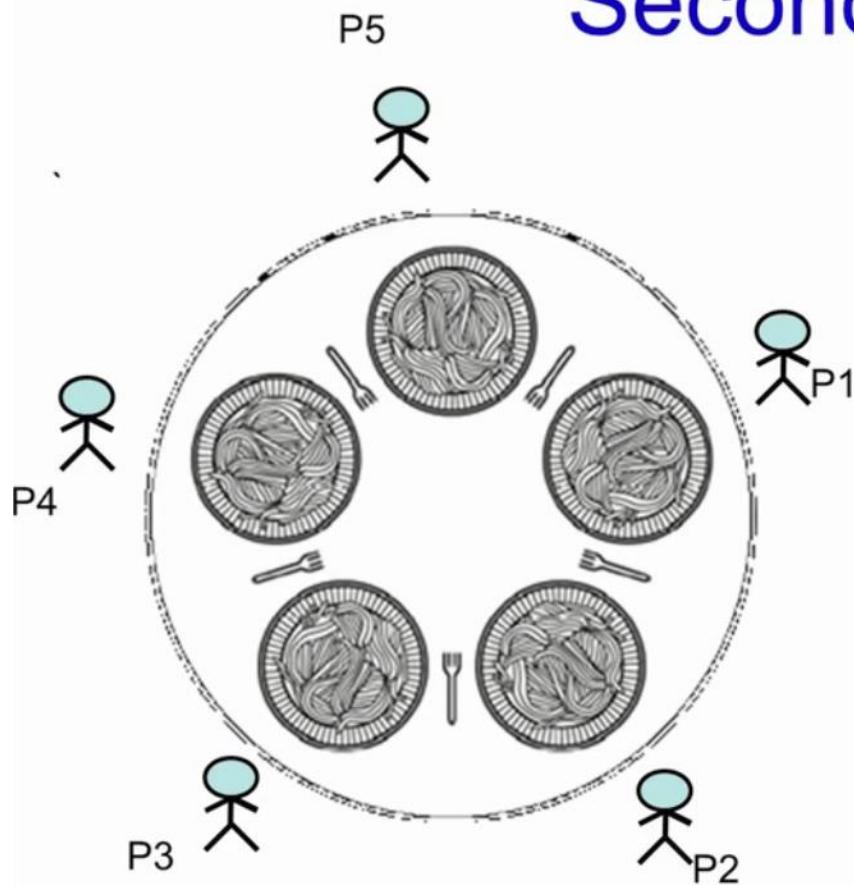
What happens if all philosophers decide to pick up their right forks at the same time?  
Possible starvation due to deadlock

# Deadlocks

- A situation where programs continue to run indefinitely without making any progress
- Each program is waiting for an event that another process can cause



## Second try



```
#define N 5
```

```
void philosopher(int i){  
    while(TRUE){  
        think();  
        take_fork(Ri);  
        if (available(Li)){  
            take_fork(Li);  
            eat();  
            put_fork(Ri);  
            put_fork(Li);  
        }else{  
            put_fork(Ri);  
            sleep(T)  
        }  
    }  
}
```

# Dining Philosopher Problem

Imagine,

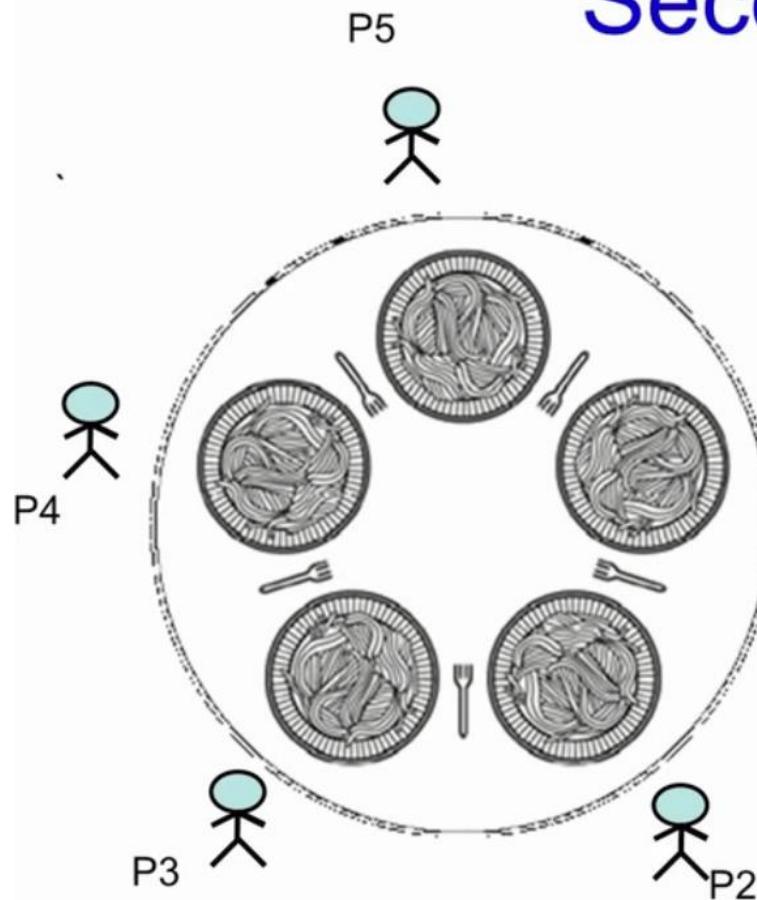
All philosophers start at the same time

Run simultaneously

And think for the same time

This could lead to philosophers taking fork and putting it down continuously. a deadlock.

## Second try



Imagine,

All philosophers start at the same time

Run simultaneously

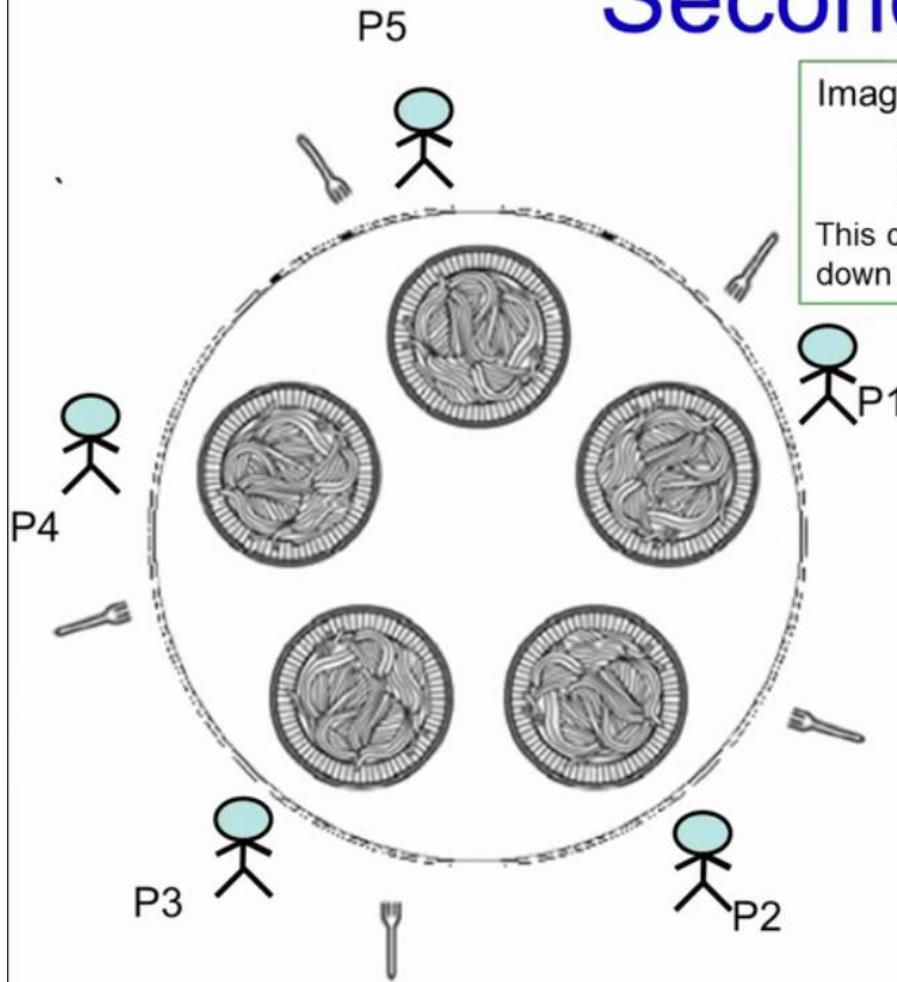
And think for the same time

This could lead to philosophers taking fork and putting it down continuously. a deadlock.

```
while(TRUE){  
    think();  
    take_fork(Ri);  
    if (available(Li)){  
        take_fork(Li);  
        eat();  
        put_fork(Ri);  
        put_fork(Li);  
    }else{  
        put_fork(Ri);  
        sleep(T)  
    }  
}
```

CR

## Second try



Imagine,

All philosophers start at the same time

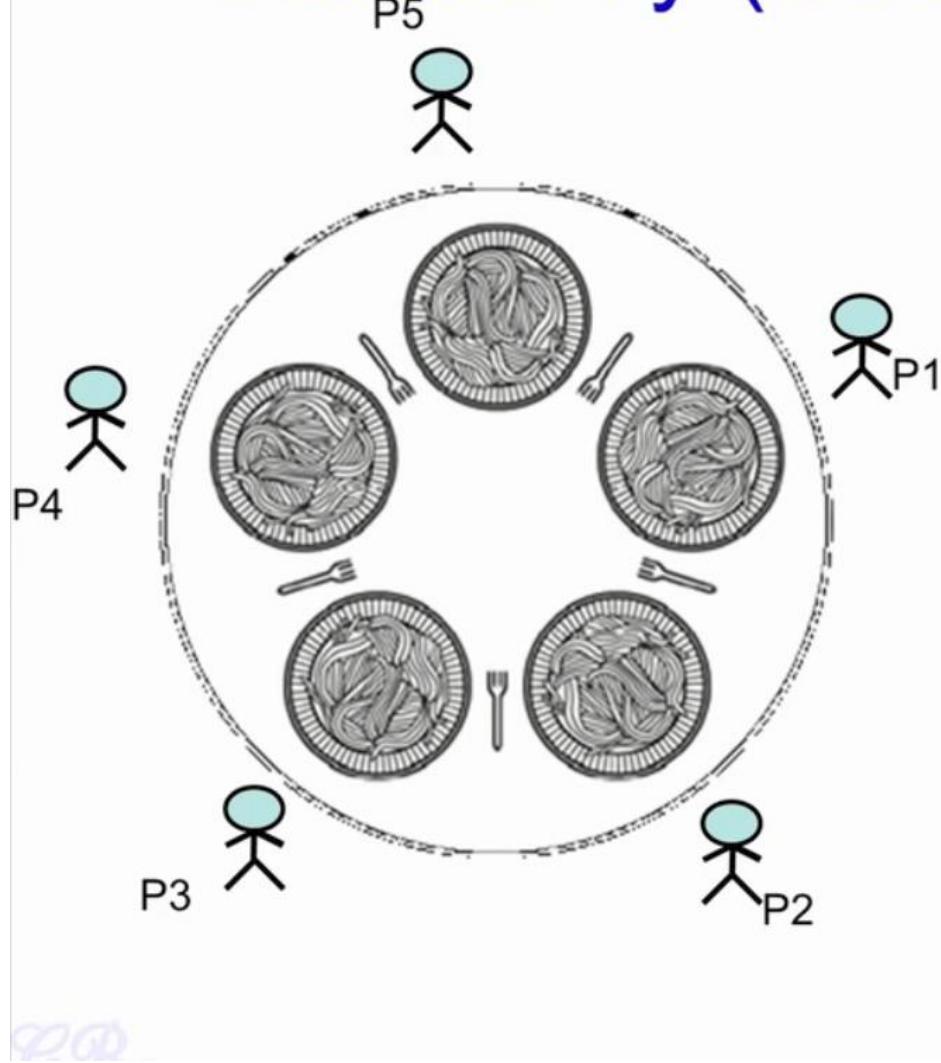
Run simultaneously

And think for the same time

This could lead to philosophers taking fork and putting it down continuously. a deadlock.

```
while(TRUE){  
    think();  
    take_fork(Ri);  
    if (available(Li){  
        take_fork(Li);  
        eat();  
        put_fork(Ri);  
        put_fork(Li);  
    }else{  
        put_fork(Ri);  
        sleep(T)  
    }  
}
```

# Second try (a better solution)



```
#define N 5
```

```
void philosopher(int i){  
    while(TRUE){  
        think();  
        take_fork(Ri);  
        if (available(Li)){  
            take_fork(Li);  
            eat();  
            put_fork(Li);  
            put_fork(Ri);  
        }else{  
            put_fork(Ri);  
            sleep(random_time);  
        }  
    }  
}
```

# Solution using Mutex

- Protect critical sections with a mutex
- Prevents deadlock
- But has performance issues
  - Only one philosopher can eat at a time

```
#define N 5

void philosopher(int i){
    while(TRUE){
        think(); // for some_time
        lock(mutex);
        take_fork(Ri);
        take_fork(Li);
        eat();
        put_fork(Li);
        put_fork(Ri);
        unlock(mutex);
    }
}
```

# Solution with Semaphores

Uses **N semaphores** ( $s[1], s[2], \dots, s[N]$ ) all initialized to 0, and a mutex  
Philosopher has 3 states: HUNGRY, EATING, THINKING

*A philosopher can only move to EATING state if neither neighbor is eating*

```
void philosopher(int i){  
    while(TRUE){  
        think();  
        take_forks(i);  
        eat();  
        put_forks();  
    }  
}
```

```
void take_forks(int i){  
    lock(mutex);  
    state[i] = HUNGRY;  
    test(i);  
    unlock(mutex);  
    down(s[i]);  
}
```

```
void put_forks(int i){  
    lock(mutex);  
    state[i] = THINKING;  
    test(LEFT);  
    test(RIGHT);  
    unlock(mutex);  
}
```

```
void test(int i){  
    if (state[i] = HUNGRY && state[LEFT] != EATING && state[RIGHT] != EATING){  
        state[i] = EATING;  
        up(s[i]);  
    }  
}
```

# Solution to Dining Philosophers

```
void philosopher(int i){  
    while(TRUE){  
        think();  
        take_forks(i);  
        eat();  
        put_forks();  
    }  
}
```

```
void take_forks(int i){  
    lock(mutex);  
    state[i] = HUNGRY;  
    test(i);  
    unlock(mutex);  
    down(s[i]);  
}
```

```
void put_forks(int i){  
    lock(mutex);  
    state[i] = THINKING;  
    test(LEFT);  
    test(RIGHT)  
    unlock(mutex);  
}
```

```
void test(int i){  
    if (state[i] = HUNGRY &&  
        state[LEFT] != EATING &&  
        state[RIGHT] != EATING){  
        state[i] = EATING;  
        up(s[i]);  
    }  
}
```

	P1	P2	P3	P4	P5
state	T	T	T	T	T
semaphore	0	0	0	0	0

# Solution to Dining Philosophers

```
void philosopher(int i){  
    while(TRUE){  
        think();  
        → take_forks(i);  
        eat();  
        put_forks();  
    }  
}
```

```
void take_forks(int i){  
    lock(mutex);  
    state[i] = HUNGRY;  
    test(i);  
    unlock(mutex);  
    down(s[i]);  
}
```

```
void put_forks(int i){  
    lock(mutex);  
    state[i] = THINKING;  
    test(LEFT);  
    test(RIGHT)  
    unlock(mutex);  
}
```

```
void test(int i){  
    if (state[i] = HUNGRY &&  
        state[LEFT] != EATING &&  
        state[RIGHT] != EATING){  
        state[i] = EATING;  
        up(s[i]);  
    }  
}
```

	P1	P2	P3	P4	P5
state	T	T	T	T	T
semaphore	0	0	0	0	0

# Solution to Dining Philosophers

```
void philosopher(int i){  
    while(TRUE){  
        think();  
        take_forks(i);  
        eat();  
        put_forks();  
    }  
}
```

```
void take_forks(int i){  
    lock(mutex);  
    → state[i] = HUNGRY;  
    test(i);  
    unlock(mutex);  
    down(s[i]);  
}
```

```
void put_forks(int i){  
    lock(mutex);  
    state[i] = THINKING;  
    test(LEFT);  
    test(RIGHT)  
    unlock(mutex);  
}
```

```
void test(int i){  
    if (state[i] = HUNGRY &&  
        state[LEFT] != EATING &&  
        state[RIGHT] != EATING){  
        state[i] = EATING;  
        up(s[i]);  
    }  
}
```

	P1	P2	P3	P4	P5
state	T	T	H	T	T
semaphore	0	0	0	0	0

# Solution to Dining Philosophers

```
void philosopher(int i){  
    while(TRUE){  
        think();  
        take_forks(i);  
        eat();  
        put_forks();  
    }  
}
```

```
void take_forks(int i){  
    lock(mutex);  
    state[i] = HUNGRY;  
    test(i);  
    unlock(mutex);  
    down(s[i]);  
}
```

```
void put_forks(int i){  
    lock(mutex);  
    state[i] = THINKING;  
    test(LEFT);  
    test(RIGHT);  
    unlock(mutex);  
}
```

```
void test(int i){  
    if (state[i] = HUNGRY &&  
        state[LEFT] != EATING &&  
        state[RIGHT] != EATING){  
        state[i] = EATING;  
        up(s[i]);  
    }  
}
```

	P1	P2	P3	P4	P5
state	T	T	E	T	T
semaphore	0	0	0	0	0

# Solution to Dining Philosophers

```
void philosopher(int i){  
    while(TRUE){  
        think();  
        take_forks(i);  
        eat();  
        put_forks();  
    }  
}
```

```
void take_forks(int i){  
    lock(mutex);  
    state[i] = HUNGRY;  
    test(i);  
    unlock(mutex);  
    down(s[i]);  
}
```

```
void put_forks(int i){  
    lock(mutex);  
    state[i] = THINKING;  
    test(LEFT);  
    test(RIGHT)  
    unlock(mutex);  
}
```

```
void test(int i){  
    if (state[i] = HUNGRY &&  
        state[LEFT] != EATING &&  
        state[RIGHT] != EATING){  
        state[i] = EATING;  
        up(s[i]);  
    }  
}
```

	P1	P2	P3	P4	P5
state	T	T	E	T	T
semaphore	0	0	1	0	0

# Solution to Dining Philosophers

```
void philosopher(int i){  
    while(TRUE){  
        think();  
        take_forks(i);  
        eat();  
        put_forks();  
    }  
}
```

```
void take_forks(int i){  
    lock(mutex);  
    state[i] = HUNGRY;  
    test(i);  
    → unlock(mutex);  
    down(s[i]);  
}
```

```
void put_forks(int i){  
    lock(mutex);  
    state[i] = THINKING;  
    test(LEFT);  
    test(RIGHT)  
    unlock(mutex);  
}
```

```
void test(int i){  
    if (state[i] = HUNGRY &&  
        state[LEFT] != EATING &&  
        state[RIGHT] != EATING){  
        state[i] = EATING;  
        up(s[i]);  
    }  
}
```

	P1	P2	P3	P4	P5
state	T	T	E	T	T
semaphore	0	0	1	0	0

# Solution to Dining Philosophers

```
void philosopher(int i){  
    while(TRUE){  
        think();  
        take_forks(i);  
        eat();  
        put_forks();  
    }  
}
```

```
void take_forks(int i){  
    lock(mutex);  
    state[i] = HUNGRY;  
    test(i);  
    unlock(mutex);  
    down(s[i]);  
}
```

```
void put_forks(int i){  
    lock(mutex);  
    state[i] = THINKING;  
    test(LEFT);  
    test(RIGHT)  
    unlock(mutex);  
}
```

```
void test(int i){  
    if (state[i] = HUNGRY &&  
        state[LEFT] != EATING &&  
        state[RIGHT] != EATING){  
        state[i] = EATING;  
        up(s[i]);  
    }  
}
```

s[i] is 1, so down will not block.  
The value of s[i] decrements by 1.

	P1	P2	P3	P4	P5
state	T	T	E	T	T
semaphore	0	0	0	0	0

# Solution to Dining Philosophers

```
void philosopher(int i){  
    while(TRUE){  
        think();  
        take_forks(i);  
        → eat();  
        put_forks();  
    }  
}
```

```
void take_forks(int i){  
    lock(mutex);  
    state[i] = HUNGRY;  
    test(i);  
    unlock(mutex);  
    down(s[i]);  
}
```

```
void put_forks(int i){  
    lock(mutex);  
    state[i] = THINKING;  
    test(LEFT);  
    test(RIGHT)  
    unlock(mutex);  
}
```

```
void test(int i){  
    if (state[i] = HUNGRY &&  
        state[LEFT] != EATING &&  
        state[RIGHT] != EATING){  
        state[i] = EATING;  
        up(s[i]);  
    }  
}
```

s[i] is 1, so down will not block.  
The value of s[i] decrements by 1.

	P1	P2	P3	P4	P5
state	T	T	E	T	T
semaphore	0	0	0	0	0

# Solution to Dining Philosophers

```
void philosopher(int i){  
    while(TRUE){  
        think();  
        → take_forks(i);  
        → eat();  
        put_forks();  
    }  
}
```

```
void take_forks(int i){  
    lock(mutex);  
    state[i] = HUNGRY;  
    test(i);  
    unlock(mutex);  
    down(s[i]);  
}
```

```
void put_forks(int i){  
    lock(mutex);  
    state[i] = THINKING;  
    test(LEFT);  
    test(RIGHT);  
    unlock(mutex);  
}
```

```
void test(int i){  
    if (state[i] = HUNGRY &&  
        state[LEFT] != EATING &&  
        state[RIGHT] != EATING){  
        state[i] = EATING;  
        up(s[i]);  
    }  
}
```

	P1	P2	P3	P4	P5
state	T	T	E	T	T
semaphore	0	0	0	0	0

# Solution to Dining Philosophers

```
void philosopher(int i){  
    while(TRUE){  
        think();  
        take_forks(i);  
        → eat();  
        put_forks();  
    }  
}
```

```
void take_forks(int i){  
    lock(mutex);  
    → state[i] = HUNGRY;  
    test(i);  
    unlock(mutex);  
    down(s[i]);  
}
```

```
void put_forks(int i){  
    lock(mutex);  
    state[i] = THINKING;  
    test(LEFT);  
    test(RIGHT)  
    unlock(mutex);  
}
```

```
void test(int i){  
    if (state[i] = HUNGRY &&  
        state[LEFT] != EATING &&  
        state[RIGHT] != EATING){  
        state[i] = EATING;  
        up(s[i]);  
    }  
}
```

	P1	P2	P3	P4	P5
state	T	T	E	H	T
semaphore	0	0	0	0	0

# Solution to Dining Philosophers

```
void philosopher(int i){  
    while(TRUE){  
        think();  
        take_forks(i);  
        → eat();  
        put_forks();  
    }  
}
```

```
void take_forks(int i){  
    lock(mutex);  
    state[i] = HUNGRY;  
    test(i);  
    unlock(mutex);  
    down(s[i]);  
}
```

```
void put_forks(int i){  
    lock(mutex);  
    state[i] = THINKING;  
    test(LEFT);  
    test(RIGHT)  
    unlock(mutex);  
}
```

```
void test(int i){  
    → if (state[i] = HUNGRY &&  
        state[LEFT] != EATING &&  
        state[RIGHT] != EATING){  
            state[i] = EATING;  
            up(s[i]);   
        }  
    }  
}
```

	P1	P2	P3	P4	P5
state	T	T	E	H	T
semaphore	0	0	0	0	0

# Solution to Dining Philosophers

```
void philosopher(int i){  
    while(TRUE){  
        think();  
        take_forks(i);  
        → eat();  
        put_forks();  
    }  
}
```

```
void take_forks(int i){  
    lock(mutex);  
    state[i] = HUNGRY;  
    test(i);  
    unlock(mutex);  
    down(s[i]);  
}
```

```
void put_forks(int i){  
    lock(mutex);  
    state[i] = THINKING;  
    test(LEFT);  
    test(RIGHT)  
    unlock(mutex);  
}
```

```
void test(int i){  
    if (state[i] = HUNGRY &&  
        state[LEFT] != EATING &&  
        state[RIGHT] != EATING){  
        state[i] = EATING;  
        up(s[i]);  
    }  
}
```

	P1	P2	P3	P4	P5
state	T	T	E	H	T
semaphore	0	0	0	0	0

# Solution to Dining Philosophers

```
void philosopher(int i){  
    while(TRUE){  
        think();  
        take_forks(i);  
        eat();  
        put_forks();  
    }  
}
```

```
void take_forks(int i){  
    lock(mutex);  
    state[i] = HUNGRY;  
    test(i);  
    unlock(mutex);  
    down(s[i]);  
}
```

```
void put_forks(int i){  
    lock(mutex);  
    state[i] = THINKING;  
    test(LEFT);  
    test(RIGHT);  
    unlock(mutex);  
}
```

```
void test(int i){  
    if (state[i] = HUNGRY &&  
        state[LEFT] != EATING &&  
        state[RIGHT] != EATING){  
        state[i] = EATING;  
        up(s[i]);  
    }  
}
```

	P1	P2	P3	P4	P5
state	T	T	E	H	T
semaphore	0	0	0	0	0

# Solution to Dining Philosophers

```
void philosopher(int i){  
    while(TRUE){  
        think();  
        take_forks(i);  
        eat();  
        put_forks();  
    }  
}
```

```
void take_forks(int i){  
    lock(mutex);  
    state[i] = HUNGRY;  
    test(i);  
    unlock(mutex);  
    down(s[i]);  
}
```

```
void put_forks(int i){  
    lock(mutex);  
    state[i] = THINKING;  
    test(LEFT);  
    test(RIGHT);  
    unlock(mutex);  
}
```

```
void test(int i){  
    if (state[i] = HUNGRY &&  
        state[LEFT] != EATING &&  
        state[RIGHT] != EATING){  
        state[i] = EATING;  
        up(s[i]);  
    }  
}
```

	P1	P2	P3	P4	P5
state	T	T	T	H	T
semaphore	0	0	0	0	0

# Solution to Dining Philosophers

```
void philosopher(int i){  
    while(TRUE){  
        think();  
        take_forks(i);  
        eat();  
        put_forks();  
    }  
}
```

```
void take_forks(int i){  
    lock(mutex);  
    state[i] = HUNGRY;  
    test(i);  
    unlock(mutex);  
    down(s[i]);  
}
```

```
void put_forks(int i){  
    lock(mutex);  
    state[i] = THINKING;  
    → test(LEFT);  
    test(RIGHT)  
    unlock(mutex);  
}
```

```
void test(int i){  
    if (state[i] = HUNGRY &&  
        state[LEFT] != EATING &&  
        state[RIGHT] != EATING){  
        state[i] = EATING;  
        up(s[i]);  
    }  
}
```

	P1	P2	P3	P4	P5
state	T	T	T	H	T
semaphore	0	0	0	0	0

# Solution to Dining Philosophers

```
void philosopher(int i){  
    while(TRUE){  
        think();  
        take_forks(i);  
        eat();  
        put_forks();  
    }  
}
```

```
void take_forks(int i){  
    lock(mutex);  
    state[i] = HUNGRY;  
    test(i);  
    unlock(mutex);  
    down(s[i]);  
}
```

```
void put_forks(int i){  
    lock(mutex);  
    state[i] = THINKING;  
    test(LEFT);  
    → test(RIGHT)  
    unlock(mutex);  
}
```

```
void test(int i){  
    if (state[i] = HUNGRY &&  
        state[LEFT] != EATING &&  
        state[RIGHT] != EATING){  
        state[i] = EATING;  
        up(s[i]);  
    }  
}
```

	P1	P2	P3	P4	P5
state	T	T	T	H	T
semaphore	0	0	0	0	0

# Solution to Dining Philosophers

```
void philosopher(int i){  
    while(TRUE){  
        think();  
        take_forks(i);  
        eat();  
        put_forks();  
    }  
}
```

```
void take_forks(int i){  
    lock(mutex);  
    state[i] = HUNGRY;  
    test(i);  
    unlock(mutex);  
    down(s[i]);  
}
```

```
void put_forks(int i){  
    lock(mutex);  
    state[i] = THINKING;  
    test(LEFT);  
    test(RIGHT);  
    unlock(mutex);  
}
```

```
void test(int i){  
    if (state[i] = HUNGRY &&  
        state[LEFT] != EATING &&  
        state[RIGHT] != EATING){  
        state[i] = EATING;  
        up(s[i]);  
    }  
}
```

	P1	P2	P3	P4	P5
state	T	T	T	E	T
semaphore	0	0	0	1	0

# Solution to Dining Philosophers

```
void philosopher(int i){  
    while(TRUE){  
        think();  
        take_forks(i);  
        eat();  
        put_forks();  
    }  
}
```

```
void take_forks(int i){  
    lock(mutex);  
    state[i] = HUNGRY;  
    test(i);  
    unlock(mutex);  
    down(s[i]);  
}  
→
```

```
void put_forks(int i){  
    lock(mutex);  
    state[i] = THINKING;  
    test(LEFT);  
    test(RIGHT)  
    unlock(mutex);  
}→
```

```
void test(int i){  
    if (state[i] = HUNGRY &&  
        state[LEFT] != EATING &&  
        state[RIGHT] != EATING){  
        state[i] = EATING;  
        up(s[i]);  
    }  
}
```

	P1	P2	P3	P4	P5
state	T	T	T	E	T
semaphore	0	0	0	1	0

# Solution to Dining Philosophers

```
void philosopher(int i){  
    while(TRUE){  
        think();  
        take_forks(i);  
        eat();  
        put_forks();  
    }  
}
```

```
void take_forks(int i){  
    lock(mutex);  
    state[i] = HUNGRY;  
    test(i);  
    unlock(mutex);  
    down(s[i]);  
}
```

```
void put_forks(int i){  
    lock(mutex);  
    state[i] = THINKING;  
    test(LEFT);  
    test(RIGHT);  
    unlock(mutex);  
}
```

```
void test(int i){  
    if (state[i] = HUNGRY &&  
        state[LEFT] != EATING &&  
        state[RIGHT] != EATING){  
        state[i] = EATING;  
        up(s[i]);  
    }  
}
```

P1	P2	P3	P4	P5
T	T	T	E	T
0	0	0	0	0

# Solution to Dining Philosophers

```
void philosopher(int i){  
    while(TRUE){  
        think();  
        take_forks(i);  
        → eat();  
        put_forks();  
    }  
}
```

```
void take_forks(int i){  
    lock(mutex);  
    state[i] = HUNGRY;  
    test(i);  
    unlock(mutex);  
    down(s[i]);  
}
```

```
void put_forks(int i){  
    lock(mutex);  
    state[i] = THINKING;  
    test(LEFT);  
    test(RIGHT)  
    unlock(mutex);  
}
```

```
void test(int i){  
    if (state[i] = HUNGRY &&  
        state[LEFT] != EATING &&  
        state[RIGHT] != EATING){  
        state[i] = EATING;  
        up(s[i]);  
    }  
}
```

P1	P2	P3	P4	P5
T	T	↑	E	T
0	0	0	0	0

# Dining Philosopher Problem

<https://genuineNotes.com>

```
#include <pthread.h>
#include <semaphore.h>
#include <stdio.h>
#define N 5
#define THINKING 2
#define HUNGRY 1
#define EATING 0
#define LEFT (phnum + 4) % N
#define RIGHT (phnum + 1) % N
int state[N];
int phil[N] = { 0, 1, 2, 3, 4 };
sem_t mutex;
sem_t S[N];
void test(int phnum)
{
    if (state[phnum] == HUNGRY && state[LEFT] != EATING && state[RIGHT] != EATING) {
        // state that eating
        state[phnum] = EATING;
        sleep(2);
        printf("Philosopher %d takes fork %d and %d\n",
               phnum + 1, LEFT + 1, phnum + 1);
        printf("Philosopher %d is Eating\n", phnum + 1);
        // sem_post(&S[phnum]) has no effect
        // during takefork
        // used to wake up hungry philosophers
        // during putfork
        sem_post(&S[phnum]);
    }
}
```

```
// take up chopsticks // put down chopsticks
void take_fork(int phnum) void put_fork(int phnum)
{
    sem_wait(&mutex);
    // state that hungry // state that thinking
    state[phnum] = HUNGRY;
    printf("Philosopher %d is Hungry\n", phnum + 1);
    // eat if neighbours are not eating
    test(phnum);
    sem_post(&mutex);
    // if unable to eat wait to be signalled
    sem_wait(&S[phnum]);
    sleep(1);
}

{
    sem_wait(&mutex);
    // state that thinking
    state[phnum] = THINKING;
    printf("Philosopher %d putting fork %d and %d down\n",
           phnum + 1, LEFT + 1, phnum + 1);
    printf("Philosopher %d is thinking\n", phnum + 1);

    test(LEFT);
    test(RIGHT);

    sem_post(&mutex);
}
```

```
void* philospher(void* num)
{
    while (1) {
        int* i = num;
        sleep(1);
        take_fork(*i);
        sleep(0);
        put_fork(*i);
    }
}

int main()
{
    int i;
    pthread_t thread_id[N];
    // initialize the semaphores
    sem_init(&mutex, 0, 1);
    for (i = 0; i < N; i++)
        sem_init(&S[i], 0, 0);
    for (i = 0; i < N; i++) {
        // create philosopher processes
        pthread_create(&thread_id[i], NULL, philospher,
                      &phil[i]);
        printf("Philosopher %d is thinking\n", i + 1);
    }

    for (i = 0; i < N; i++)
        pthread_join(thread_id[i], NULL);
}
```

# CPU Scheduling

## Introduction:

- In a multiprogramming system, frequently multiple process competes for the CPU at the same time.
- When two or more process are simultaneously in the ready state a choice has to be made which process is to run next.
- This part of the OS is called Scheduler and the algorithm is called scheduling algorithm.
- Process execution consists of cycles of CPU execution and I/O wait. Processes alternate between these two states.
- Process execution begins with a CPU burst that is followed by I/O burst, which is followed by another CPU burst then another i/o burst, and so on.
- Eventually, the final CPU burst ends with a system request to terminate execution.

# CPU Scheduling

## **The long-term scheduler:**

- selects processes from this process pool and loads selected processes into memory for execution.

## **The short-term scheduler:**

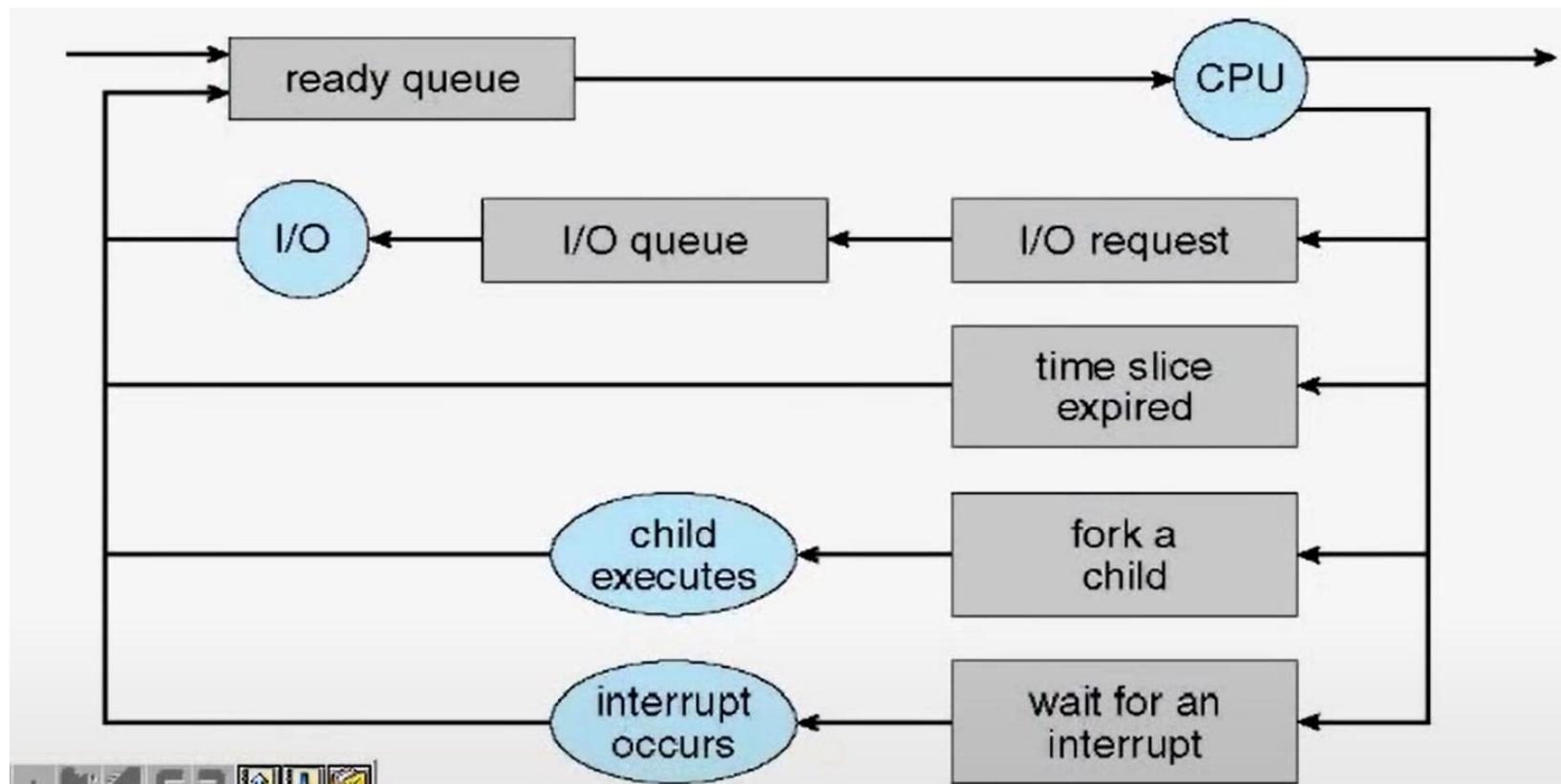
- selects the process to get the processor from among the processes which are already in memory.
- The short-time scheduler will be executing frequently (mostly at least once every 10 milliseconds).

So it has to be very fast in order to achieve a better processor utilization.

## **Medium term scheduler:**

- It can sometimes be good to reduce the degree of multiprogramming by removing processes from memory and storing them on disk.
- These processes can then be reintroduced into memory by the medium-term scheduler.
- This operation is also known as swapping. Swapping may be necessary to free memory.

# CPU Scheduling



# Scheduling Criteria:

Many criteria have been suggested for comparison of CPU scheduling algorithms.

## CPU utilization:

- we have to keep the CPU as busy as possible. It may range from 0 to 100%. In a real system it should range from 40 – 90 % for lightly and heavily loaded system.

## Throughput:

- It is the measure of work in terms of number of process completed per unit time. Eg: For long process this rate may be 1 process per hour, for short transaction, throughput may be 10 process per second.

# Scheduling Criteria:

## Turnaround Time:

- It is the sum of time periods spent in waiting to get into memory, waiting in ready queue, execution on the CPU and doing I/O.
- The interval from the time of submission of a process to the time of completion is the turnaround time.
- Waiting time plus the service time.
- **Turnaround time** = Time of completion of job - Time of submission of job. (waiting time + service time or burst time)

## Waiting time:

- it is the sum of periods waiting in the ready queue.

## Response time:

- in interactive system the turnaround time is not the best criteria.
- Response time is the amount of time it takes to start responding, not the time taken to output that response.

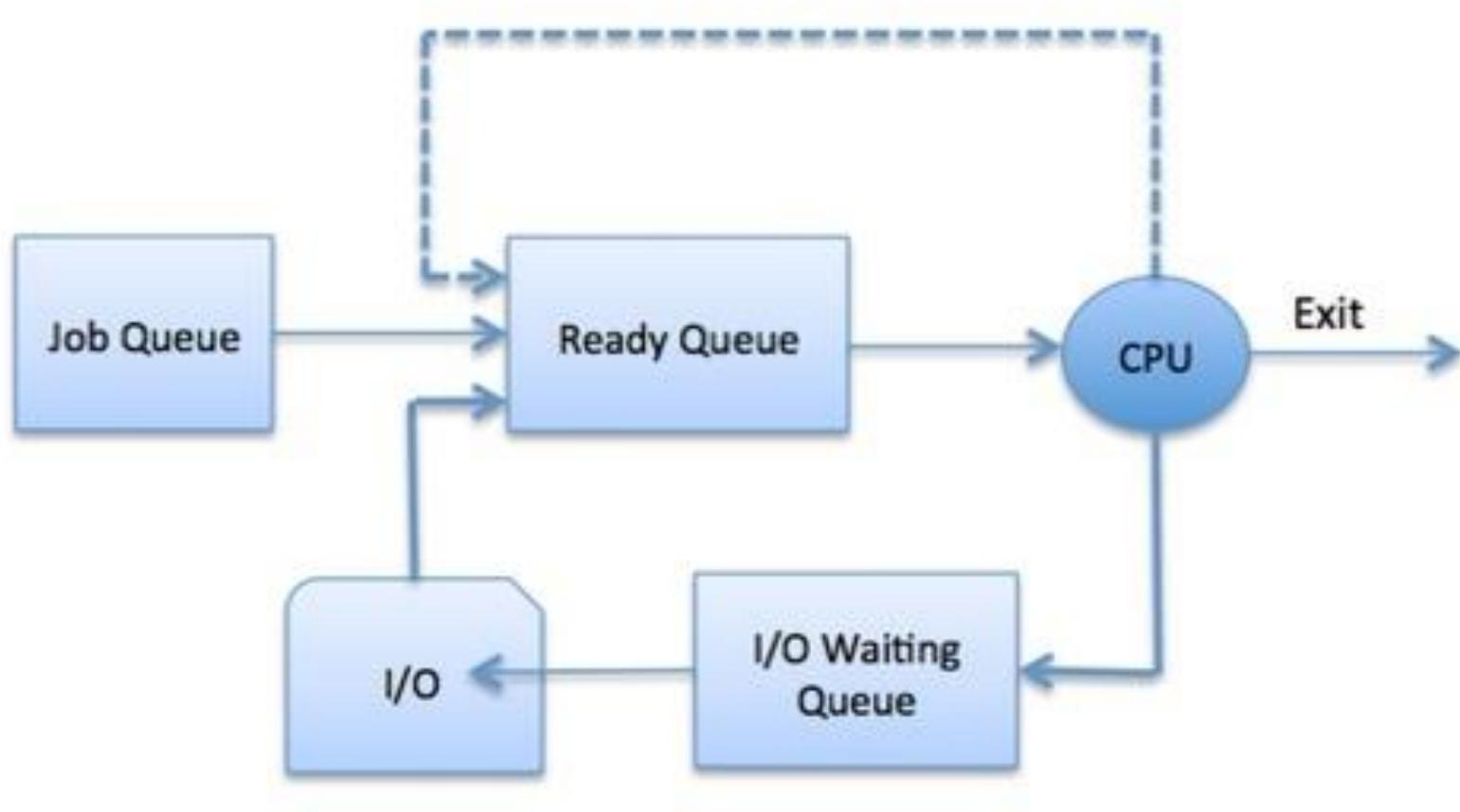
# Process Scheduling Queues

The OS maintains all PCBs in Process Scheduling Queues. The OS maintains a separate queue for each of the process states and PCBs of all processes in the same execution state are placed in the same queue. When the state of a process is changed, its PCB is unlinked from its current queue and moved to its new state queue.

The Operating System maintains the following important process scheduling queues –

- **Job queue** – This queue keeps all the processes in the system.
- **Ready queue** – This queue keeps a set of all processes residing in main memory, ready and waiting to execute. A new process is always put in this queue.
- **Device queues** – The processes which are blocked due to unavailability of an I/O device constitute this queue.

# Process Scheduling Queues



# Two-State Process Model

Two-state process model refers to running and non-running states which are described below –

- **Running**

When a new process is created, it enters into the system as in the running state.

- **Not Running**

Processes that are not running are kept in queue, waiting for their turn to execute. Each entry in the queue is a pointer to a particular process. Queue is implemented by using linked list. Use of dispatcher is as follows. When a process is interrupted, that process is transferred in the waiting queue. If the process has completed or aborted, the process is discarded. In either case, the dispatcher then selects a process from the queue to execute.

# Types of Scheduling:

## 1. Preemptive Scheduling

- **preemptive** scheduling algorithm picks a process and lets it run for a maximum of some fixed time.
- If it is still running at the end of the time interval, it is suspended and the scheduler picks another process to run (if one is available).
- Doing preemptive scheduling requires having a clock interrupt occur at the end of time interval to give control of the CPU back to the scheduler.

# Types of Scheduling:

## 2. Non preemptive Scheduling

- **Nonpreemptive** scheduling algorithm picks a process to run and then just lets it run until it blocks (either on I/O or waiting for another process) or until it voluntarily releases the CPU.
- Even it runs for hours, it will not be forcibly suspended.

# Preemptive Vs NonPreemptive

<https://genuineNotes.com>

PARAMENTER	PREEMPTIVE SCHEDULING	NON-PREEMPTIVE SCHEDULING
Basic	In this resources(CPU Cycle) are allocated to a process for a limited time.	Once resources(CPU Cycle) are allocated to a process, the process holds it till it completes its burst time or switches to waiting state.
Interrupt	Process can be interrupted in between.	Process can not be interrupted until it terminates itself or its time is up.
Starvation	If a process having high priority frequently arrives in the ready queue, low priority process may starve.	If a process with long burst time is running CPU, then later coming process with less CPU burst time may starve.
Overhead	It has overheads of scheduling the processes.	It does not have overheads.
Flexibility	flexible	rigid
Cost	cost associated	no cost associated

# 1. Preemptive Scheduling:

- Preemptive scheduling is used when a process switches from running state to ready state or from waiting state to ready state.
- The resources (mainly CPU cycles) are allocated to the process for the limited amount of time and then is taken away, and the process is again placed back in the ready queue if that process still has CPU burst time remaining. That process stays in ready queue till it gets next chance to execute.
- Algorithms based on preemptive scheduling are: Round Robin (RR), Shortest Job First (SJF basically non preemptive) and Priority (non preemptive version), etc.

# 1. Preemptive Scheduling:

Process	Arrival Time	CPU Burst Time (in millisec.)
P0	3	2
P1	2	4
P2	0	6
P3	1	4



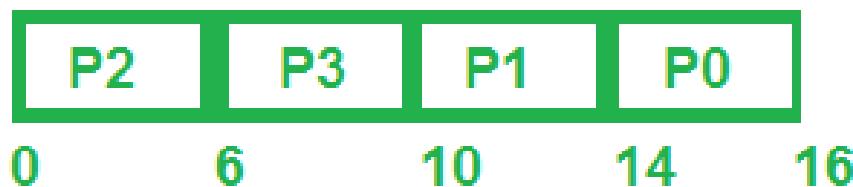
Preemptive Scheduling

## 2. Non-Preemptive Scheduling:

- Non-preemptive Scheduling is used when a process terminates, or a process switches from running to waiting state.
- In this scheduling, once the resources (CPU cycles) is allocated to a process, the process holds the CPU till it gets terminated or it reaches a waiting state. In case of non-preemptive scheduling does not interrupt a process running CPU in middle of the execution. Instead, it waits till the process complete its CPU burst time and then it can allocate the CPU to another process.
- Algorithms based on preemptive scheduling are: Shortest Remaining Time First (SRTF), Priority (preemptive version), etc.

## 2. Non-Preemptive Scheduling:

Process	Arrival Time	CPU Burst Time (in millisec.)
P0	3	2
P1	2	4
P2	0	6
P3	1	4



**Non-Preemptive Scheduling**

# Dispatcher

- A dispatcher is a special program which comes into play after the scheduler.
- When the scheduler completes its job of selecting a process, it is the dispatcher which takes that process to the desired state/queue.
- The dispatcher is the module that gives a process control over the CPU after it has been selected by the short-term scheduler. This function involves the following:
  - Switching context
  - Switching to user mode
  - Jumping to the proper location in the user program to restart that program

# Scheduler

**Schedulers** are special system software which handle process scheduling in various ways. Their main task is to select the jobs to be submitted into the system and to decide which process to run. There are three types of Scheduler:

- 1. Long term (job) scheduler**
- 2. Medium term scheduler**
- 3. Short term (CPU) scheduler**

# Scheduler

## 1. Long term (job) scheduler –

- Due to the smaller size of main memory initially all programs are stored in secondary memory.
- When they are stored or loaded in the main memory they are called process.
- This is the decision of long term scheduler that how many processes will stay in the ready queue.
- Hence, in simple words, long term scheduler decides the degree of multi-programming of system.

# Scheduler

## 2. Medium term scheduler –

- Most often, a running process needs I/O operation which doesn't require CPU.
- Hence during the execution of a process when a I/O operation is required then the operating system sends that process from running queue to blocked queue.
- When a process completes its I/O operation then it should again be shifted to ready queue.
- ALL these decisions are taken by the medium-term scheduler. Medium-term scheduling is a part of **swapping**.

# Scheduler

## 3. Short term (CPU) scheduler –

- When there are lots of processes in main memory initially all are present in the ready queue.
- Among all of the process, a single process is to be selected for execution.
- This decision is handled by short term scheduler.
- Let's have a look at the figure given below. It may make a more clear view for you.

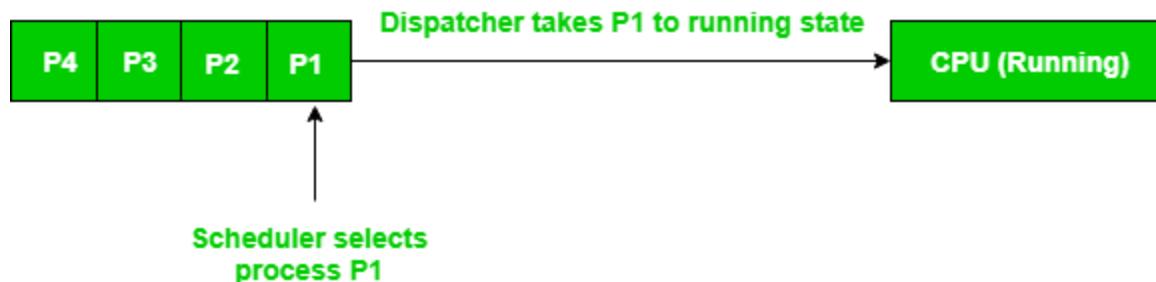
# The Difference between the Scheduler and Dispatcher

- Consider a situation, where various processes are residing in the ready queue waiting to be executed.
- The CPU cannot execute all of these processes simultaneously, so the operating system has to choose a particular process on the basis of the scheduling algorithm used. So, this procedure of selecting a process among various processes is done by **the scheduler**.
- Once the scheduler has selected a process from the queue, the **dispatcher** comes into the picture, and it is the dispatcher who takes that process from the ready queue and moves it into the running state.
- Therefore, the scheduler gives the dispatcher an ordered list of processes which the dispatcher moves to the CPU over time.

# The Difference between the Scheduler and Dispatcher

## Example

There are 4 processes in the ready queue, P1, P2, P3, P4; Their arrival times are t0, t1, t2, t3 respectively. A First in First out (FIFO) scheduling algorithm is used. Because P1 arrived first, the scheduler will decide it is the first process that should be executed, and the dispatcher will remove P1 from the ready queue and give it to the CPU. The scheduler will then determine P2 to be the next process that should be executed, so when the dispatcher returns to the queue for a new process, it will take P2 and give it to the CPU. This continues in the same way for P3, and then P4.



PROPERTIES	DISPATCHER	SCHEDULER
Definition:	Dispatcher is a module that gives control of CPU to the process selected by short term scheduler	Scheduler is something which selects a process among various processes
Types:	There are no different types in dispatcher. It is just a code segment.	There are 3 types of scheduler i.e. Long-term, Short-term, Medium-term
Dependency:	Working of dispatcher is dependednt on scheduler. Means dispatcher have to wait untill scheduler selects a process.	Scheduler works independently. It works immediately when needed
Algorithm:	Dispatcher has no specific algorithm for its implementation	Scheduler works on various algorithm such as FCFS, SJF, RR etc.
Time Taken:	The time taken by dispatcher is called dispatch latency.	Time taken by scheduler is usually negligible. Hence we neglect it.
Functions:	Dispatcher is also responsible for: Context Switching, Switch to user mode, Jumping to proper location when process again restarted	The only work of scheduler is selection of processes.

# Scheduling Criteria

There are several different criteria to consider when trying to select the "best" scheduling algorithm for a particular situation and environment, including:

1. **CPU utilization** - Ideally the CPU would be busy 100% of the time, so as to waste 0 CPU cycles. On a real system CPU usage should range from 40% ( lightly loaded ) to 90% ( heavily loaded. )
2. **Throughput** - Number of processes completed per unit time. May range from 10 / second to 1 / hour depending on the specific processes.
3. **Turnaround time** - Time required for a particular process to complete, from submission time to completion. ( Wall clock time. )
4. **Waiting time** - How much time processes spend in the ready queue waiting their turn to get on the CPU.
  - ( **Load average** - The average number of processes sitting in the ready queue waiting their turn to get into the CPU. Reported in 1-minute, 5-minute, and 15-minute averages by "uptime" and "who". )
5. **Response time** - The time taken in an interactive program from the issuance of a command to the *commence* of a response to that command.

# Scheduling Criteria

- In general one wants to optimize the average value of a criteria ( Maximize CPU utilization and throughput, and minimize all the others. ) However sometimes one wants to do something different, such as to minimize the maximum response time.
- Sometimes it is most desirable to minimize the *variance* of a criteria than the actual value. I.e. users are more accepting of a consistent predictable system than an inconsistent one, even if it is a little bit slower.

# The goals of CPU scheduling are:

- **Fairness:** Each process gets fair share of the CPU.
- **Efficiency:** When CPU is 100% busy then efficiency is increased.
- **Response Time:** Minimize the response time for interactive user.
- **Throughput:** Maximizes jobs per given time period.
- **Waiting Time:** Minimizes total time spent waiting in the ready queue.
- **Turn Around Time:** Minimizes the time between submission and termination.

# Scheduling

- Batch system scheduling
  - First come first served
  - Shortest job first
  - Shortest remaining time next
- Interactive System Scheduling
  - Round Robin scheduling
  - Priority scheduling
  - Multiple queues

# 1. First come first served

- FCFS is the simplest **non-preemptive** algorithm. Processes are assigned the CPU in the order they request it. That is the process that requests the CPU first is allocated the CPU first.
- The implementation of FCFS is policy is managed with a FIFO(First in first out) queue.
- When the first job enters the system from the outside in the morning, it is started immediately and allowed to run as long as it wants to.
- As other jobs come in, they are put onto the end of the queue.
- When the running process blocks, the first process on the queue is run next.
- When a blocked process becomes ready, like a newly arrived job, it is put on the end of the queue.

# 1. First come first served

## Advantages:

- Easy to understand and program.
- Equally fair.,
- Suitable specially for Batch Operating system.



## Disadvantages:

- FCFS is not suitable for time-sharing systems where it is important that each user should get the CPU for an equal amount of arrival time.

# 1. First come first served

- Calculate the average waiting time if the processes arrive in the order of:
  - P1, P2, P3
  - P2, P3, P1

Process	Burst Time
P1	24
P2	3
P3	3

# 1. First come first served

- a) The processes arrive in the order P1, P2, P3. Let us assume they arrive in the same time at 0 ms in the system.

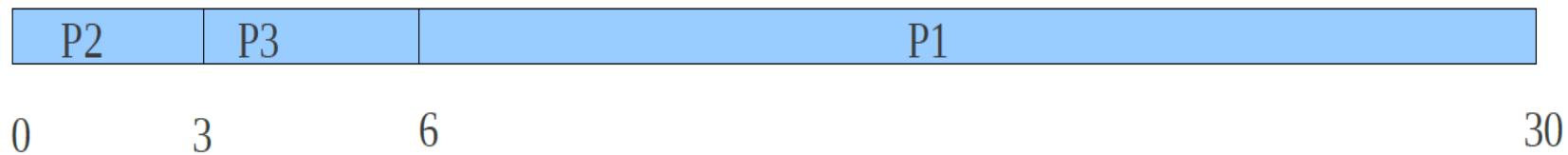
We get the following gantt chart.



- Waiting time for P1 = 0ms , for P2 = 24 ms for P3 = 27ms
- Avg waiting time:  $(0+24+27)/3= 17$

# 1. First come first served

b. ) If the process arrive in the order P2, P3, P1



- Average waiting time:  $(0+3+6)/3=3$ .
- Average waiting time vary substantially if the process CPU burst time vary greatly.

## 2. Shortest Job First:

- When several equally important jobs are sitting in the i/p queue waiting to be started, the scheduler picks the shortest jobs first.
- The disadvantages of this algorithm is the problem to know the length of time for which CPU is needed by a process.
- The SJF is optimal when all the jobs are available simultaneously.
- The SJF is either preemptive or non preemptive.
- Preemptive SJF scheduling is sometimes called **Shortest Remaining Time First** scheduling.
- With this scheduling algorithms the scheduler always chooses the process whose remaining run time is shortest.

## 2. Shortest Job First:

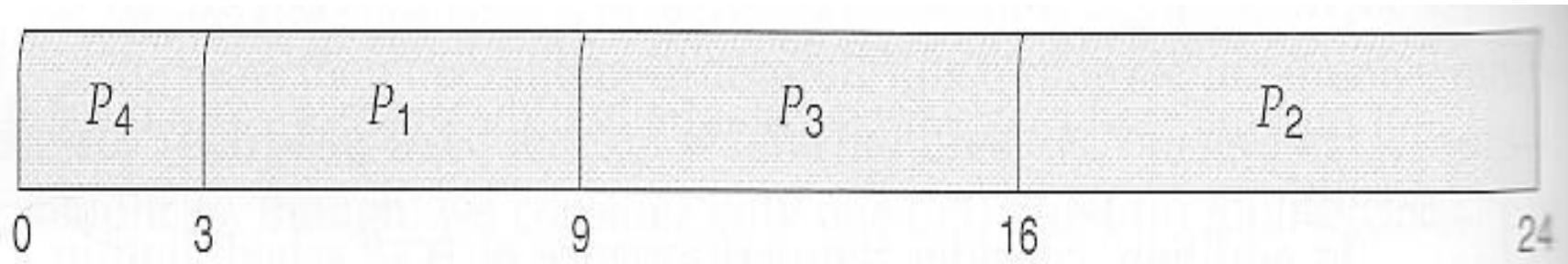
- When a new job arrives its total time is compared to the current process remaining time.
- If the new job needs less time to finish than the current process, the current process is suspended and the new job is started.
- This scheme allows new short jobs to get good service.

## 2. Shortest Job First:

- For example, the Gantt chart below is based upon the following CPU burst times, ( and the assumption that all jobs arrive at the same time. )

Process	Burst Time
P1	6
P2	8
P3	7
P4	3

## 2. Shortest Job First:



In the case above,

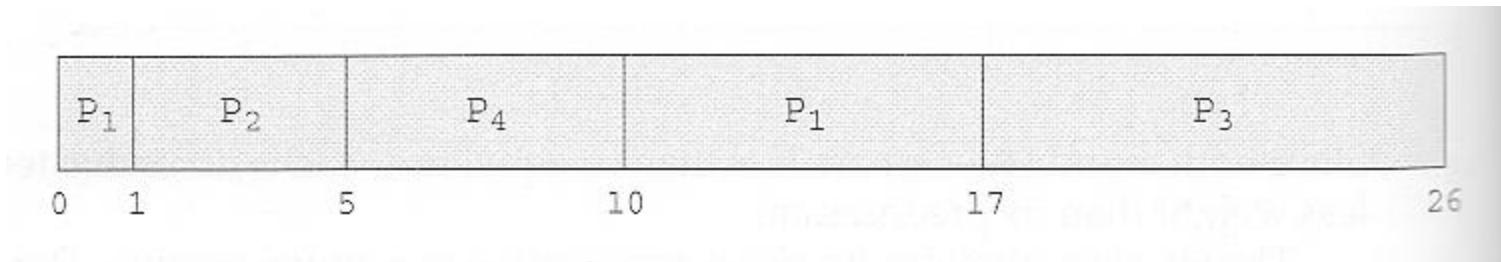
the average wait time is  $(0 + 3 + 9 + 16) / 4 = 7.0$  ms,  
( as opposed to 10.25 ms for FCFS for the same processes. )

## 2. Shortest Job First:

- For example, the following Gantt chart is based upon the following data:

Process	Arrival Time	Burst Time
P1	0	8
P2	1	4
P3	2	9
p4	3	5

## 2. Shortest Job First:



The average wait time in this case is  $( ( 5 - 3 ) + ( 10 - 1 ) + ( 17 - 2 ) ) / 4 = 26 / 4 = 6.5 \text{ ms}$ .

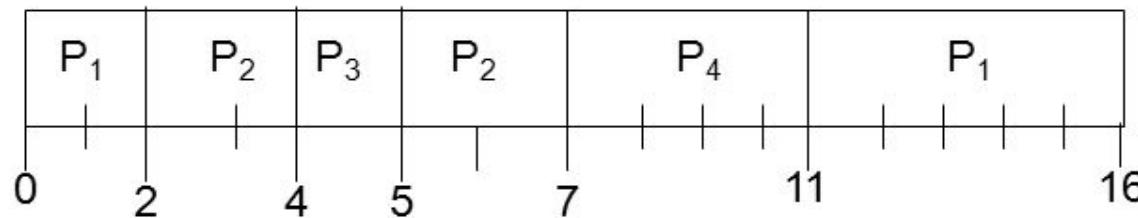
( As opposed to 7.75 ms for non-preemptive SJF or 8.75 for FCFS. )

### 3. Shortest remaining time next (SRTF)

- This Algorithm is the **preemptive version** of **SJF scheduling**.
- In SRTF, the execution of the process can be stopped after certain amount of time. At the arrival of every process, the short term scheduler schedules the process with the least remaining burst time among the list of available processes and the running process.
- Once all the processes are available in the **ready queue**, No preemption will be done and the algorithm will work as **SJF scheduling**.
- The context of the process is saved in the **Process Control Block** when the process is removed from the execution and the next process is scheduled. This PCB is accessed on the **next execution** of this process.

### 3. Shortest remaining time next

Process	Arrival Time	Burst Time
$P_1$	0.0	7
$P_2$	2.0	4
$P_3$	4.0	1
$P_4$	5.0	4



- **Throughput:** 4 processes/16 milliseconds =  $1/4$
- **Turnaround time** for  $P_1 = 16-0 = 16$ ;  $P_2 = 7-2 = 5$ ;  $P_3 = 5-4 = 1$ ;  $P_4 = 11-5 = 6$
- **Average turnaround time:**  $(16 + 5 + 1 + 6)/4 = 28/4 = 7$
- **Waiting time** for  $P_1 = 11-2 = 9$ ;  $P_2 = 5-4 = 1$ ;  $P_3 = 4-4 = 0$ ;  $P_4 = 7-5 = 2$
- **Average waiting time:**  $(9 + 1 + 0 + 2)/4 = 12/4 = 3$

## 4. Round-Robin Scheduling Algorithms:

- One of the oldest, simplest, fairest and most widely used algorithm is round robin (RR).
- In the round robin scheduling, processes are dispatched in a FIFO manner but are given a limited amount of CPU time called a time-slice or a quantum.
- If a process does not complete before its CPU-time expires, the CPU is preempted and given to the next process waiting in a queue.
- The preempted process is then placed at the back of the ready list.
- If the process has blocked or finished before the quantum has elapsed the CPU switching is done.
- Round Robin Scheduling is preemptive (at the end of time-slice) therefore it is effective in timesharing environments in which the system needs to guarantee reasonable response times for interactive users.

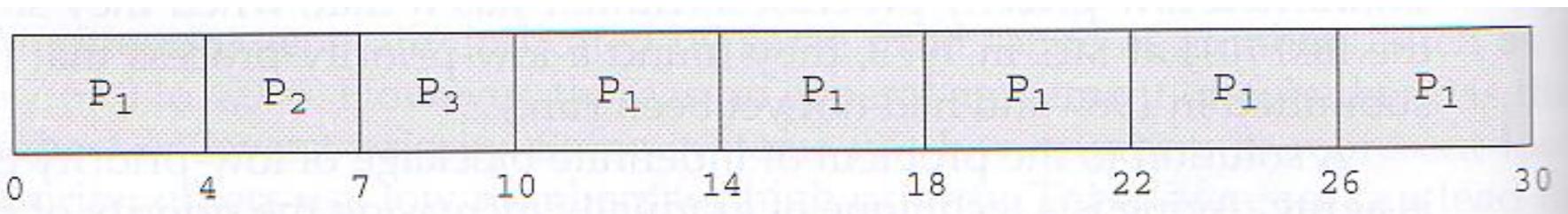
## 4. Round-Robin Scheduling Algorithms:

- The only interesting issue with round robin scheme is the length of the quantum.
- Setting the quantum too short causes too many context switches and lower the CPU efficiency.
- On the other hand, setting the quantum too long may cause poor response time and approximates FCFS.
- In any event, the average waiting time under round robin scheduling is quite long.

## 4. Round-Robin Scheduling Algorithms:

Process	Burst Time	Quantum = 4
P1	24	
P2	3	
P3	3	

Solution :



# 5. Priority Scheduling:

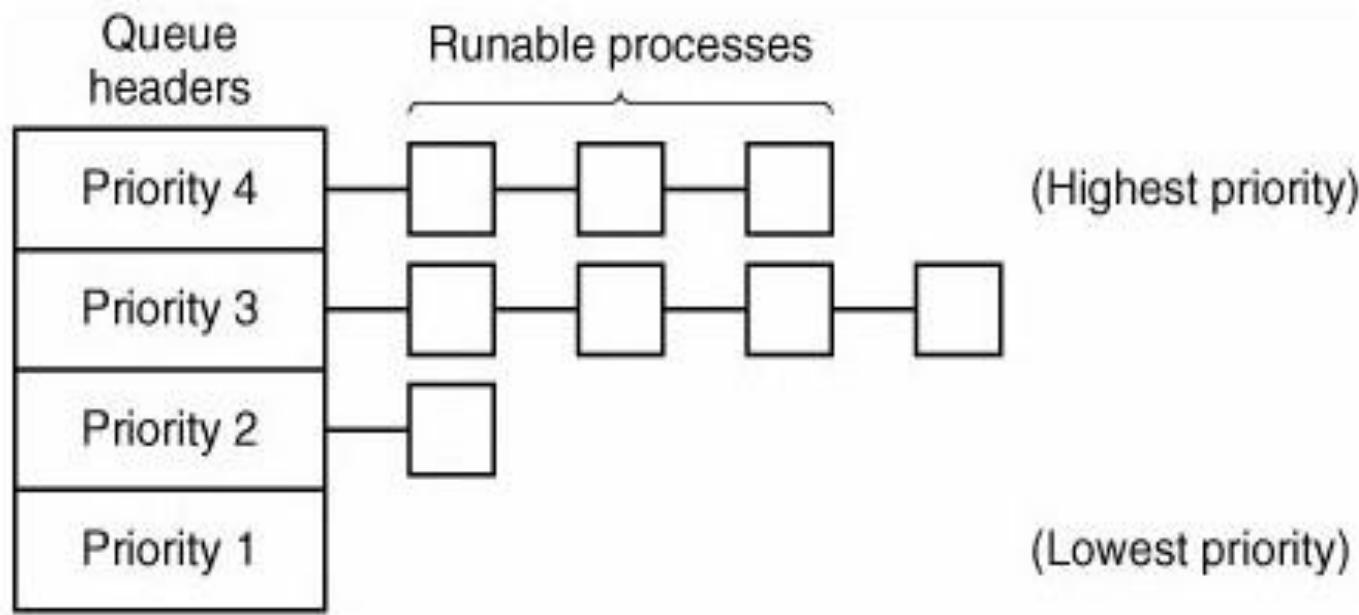
- A priority is associated with each process, and the CPU is allocated to the process with the highest priority.
- Equal priority processes are scheduled in the FCFS order.

## Assigning priority:

1. To prevent high priority process from running indefinitely the scheduler may decrease the priority of the currently running process at each clock interrupt. If this causes its priority to drop below that of the next highest process, a process switch occurs.
2. Each process may be assigned a maximum time quantum that is allowed to run. When this quantum is used up, the next highest priority process is given a chance to run.

## 5. Priority Scheduling:

- It is often convenient to group processes into priority classes and use priority scheduling among the classes but round-robin scheduling within each class.



# 5. Priority Scheduling:

## Problems in Priority Scheduling:

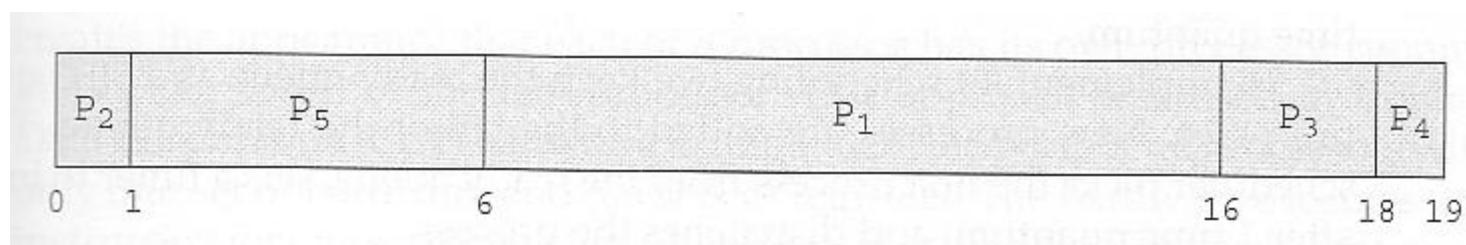
Starvation:

- Low priority process may never execute.
- Solution: **Aging**: As time progresses increase the priority of Process.

# 5. Priority Scheduling:

Process	Burst Time	Priority
P1	10	3
P2	1	1
P3	2	4
P4	1	5
P5	5	2

Solution :



## 6. Multilevel Queue Scheduling

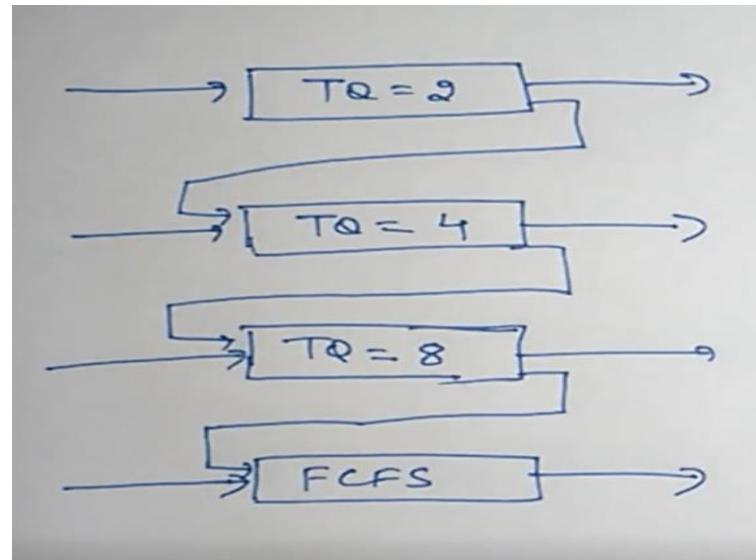
- Depending on the priority of the process, In which particular ready queue, the process has to be placed will be decided.
- The high priority process will be placed in the top level ready queue and low priority process will be placed in the bottom level queue.
- Only after completion of all the process from the top level ready queue, the further level ready queue process will be scheduled.
- The major drawback of this scheduling is, the process placed in the bottom level ready queue will suffer from starvation.
- It is also possible to use different scheduling algorithms in the different ready queue.
- To solve the problem of the starvation in this algorithm we use of the feedback in this algorithm which is called multi level feedback scheduling.

# 6. Multilevel feedback Queue Scheduling

- This algorithm avoids the problem of starvation and at the same time preference will be given to high priority process.

## example

Lets say a process arrives with the time quantum of 20 then,  
It will be executed in the 1<sup>st</sup> level ready queue for 2 time unit and it will be shifted to 2<sup>nd</sup> level ready queue In 2<sup>nd</sup> level queue it will execute for 4 time units and it will again be shifted to the 3<sup>rd</sup> level ready queue and so on until it completes the complete execution.



# 6. Multilevel Queue Scheduling

- Let us look at an example of a multilevel queue scheduling algorithm with five queues, listed below in the order of priority.

1. System processes

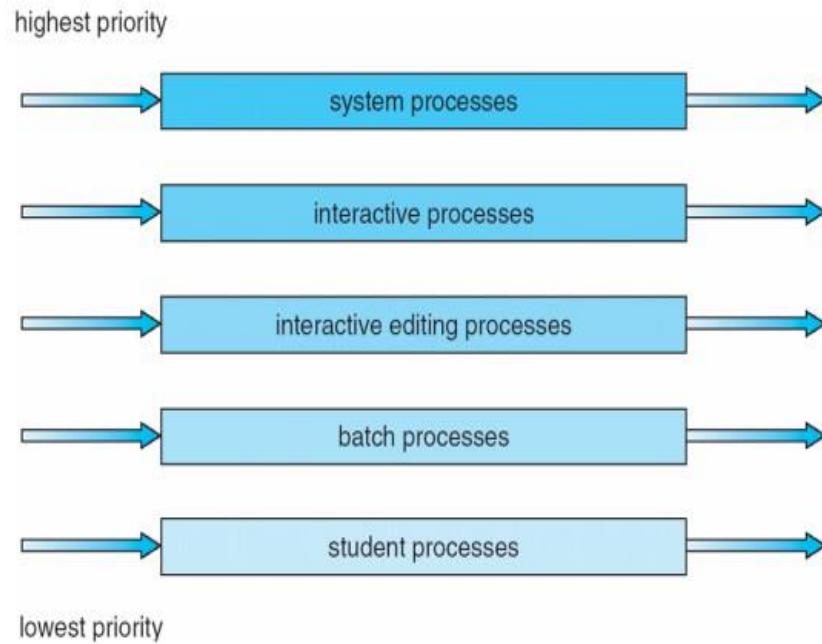
2. Interactive processes

3. Interactive editing processes

4. Batch processes

5. Student processes

- Each queue has absolute priority over lower priority queues.



# 6. Multilevel Queue Scheduling

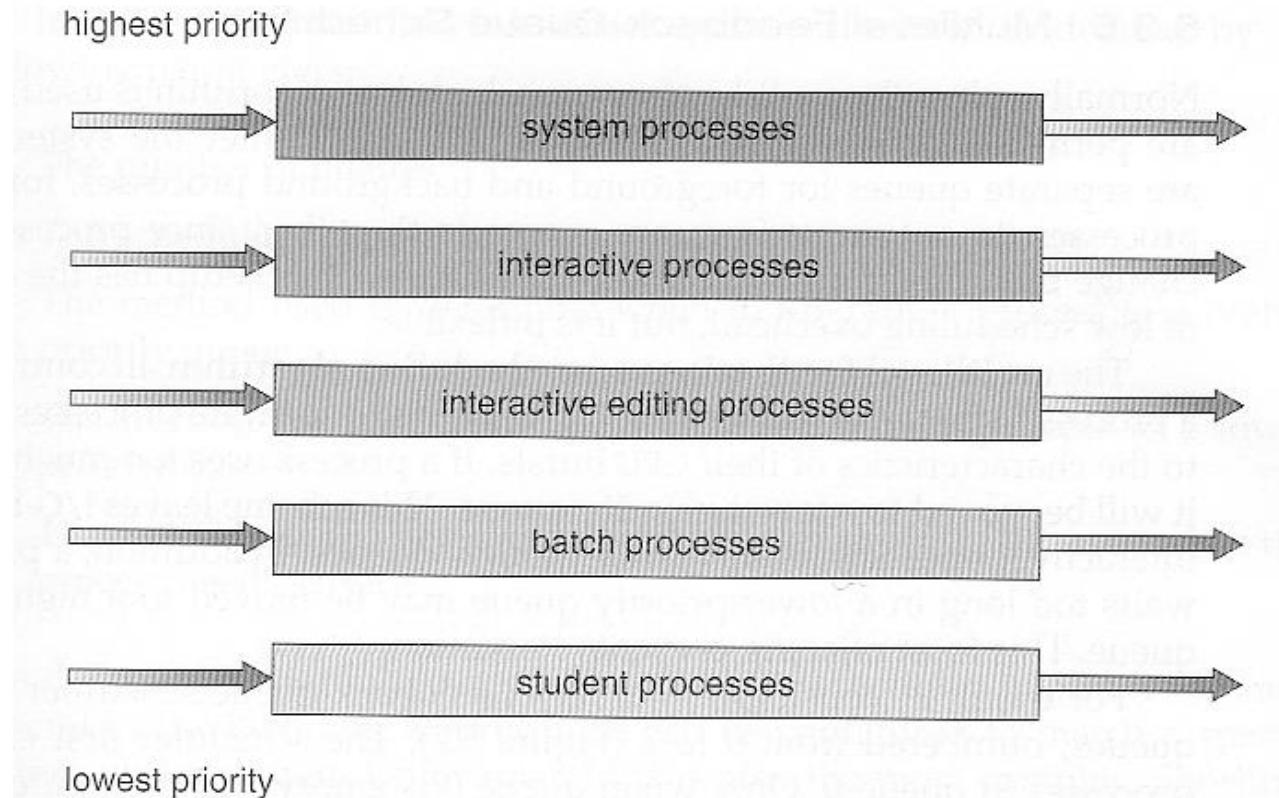


Figure 5.6 Multilevel queue scheduling.

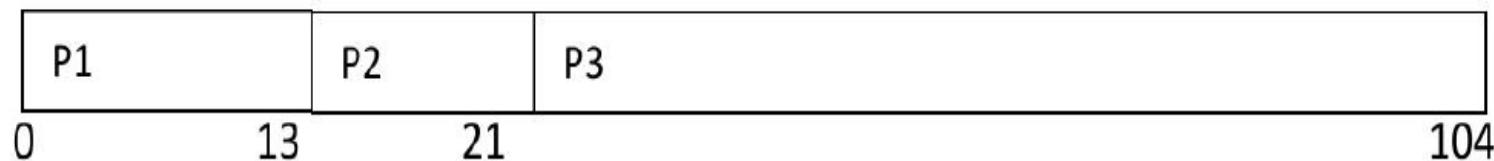
# Exercise:

<https://genuineNotes.com>

1. Draw the Gantt Chart for FCFS policy, considering the following set of processes that arrive at time 0, with the length of CPU burst given in msec, also calculate the average waiting time.

Processes	Burst Time
P1	13
P2	8
P3	83

Solution :



Waiting time for p1 = 0ms

Waiting time for p2 = 13 - 0 = 13ms

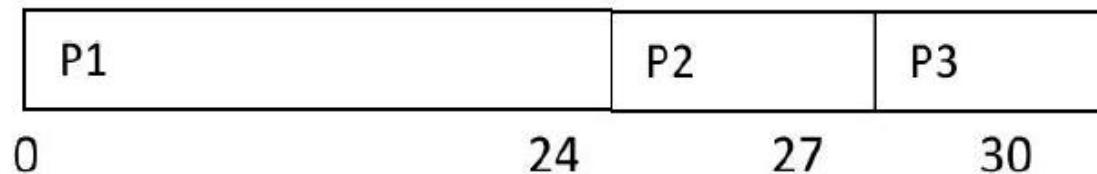
Waiting time for p3 = 21 - 0 = 21ms

Average waiting time=  $(0 + 13 + 21) / 3 = 11.33$  msec

2. Draw the Gantt chart for FCFS policy, considering the following set of processes that arrive at time as below, with the length of CPU burst given in msec, also calculate the average waiting time.

Processes	Burst Time	Arrival Time
P1	24	0
P2	3	1
P3	3	2

Solution :



Waiting time for P1 = 0ms

Waiting time for P2 =  $(24 - 0) - 1 = 23$ ms

Waiting time for P3 =  $(27 - 0) - 2 = 25$ ms

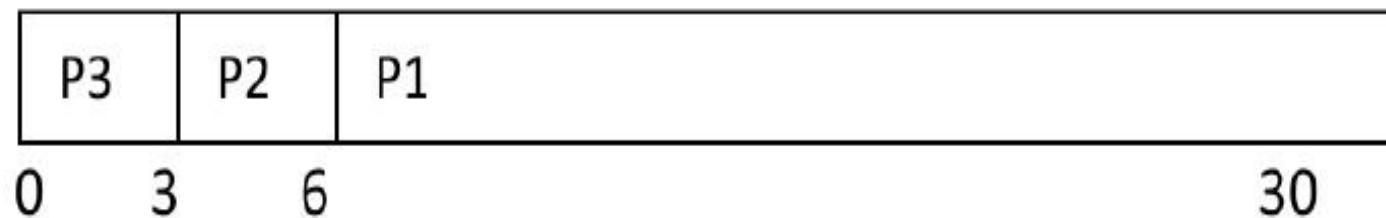
Average waiting time =  $(0 + 23 + 25) / 3 = 16$ ms

## Exercise:

3. Consider the following set of processes having their burst time mentioned in milliseconds. Calculate the average waiting time for FCFS policy, if the processes arrive in the order P2, P3, P1.

Processes	Burst Time
P1	24
P2	3
P3	3

## Solution :



$$\text{Average waiting time} = (0 + 3 + 6) / 3 = 3 \text{ ms}$$

# Exercise:

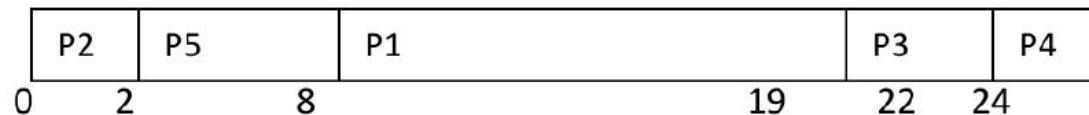
<https://genuineNotes.com>

4. Calculate the average waiting time for priority policy when all the processes arrived at time zero

Processes	Burst Time	priority
P1	11	3
P2	2	1
P3	3	4
P4	2	5
P5	6	2

Solution :

Preemptive:



Waiting time for P1= 8

Waiting time for P2= 0

Waiting time for P3= 19

Waiting time for P4= 22

Waiting time for P5= 2

Average waiting time= 10.2 ms

# Exercise:

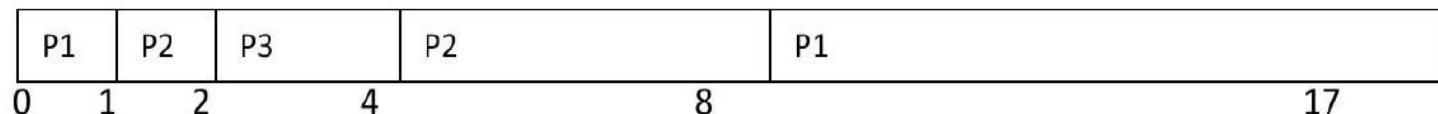
<https://genuineNotes.com>

5. Consider the following set of processes p1, p2, p3 having priorities ranging from 1 to 3. Using priority (preemptive and not-preemptive) scheduling, schedule the processes.

Processes	Burst Time	priority	Arrival Time
P1	10	3	0
P2	5	2	1
P3	2	1	2

Solution :

**Preemptive:**



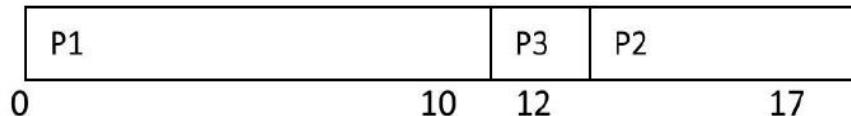
$$\text{Waiting time for P1} = 0 + (8 - 1) - 0 = 7$$

$$\text{Waiting time for P2} = (1 - 0) + (4 - 2) - 1 = 2$$

$$\text{Waiting time for P3} = (2 - 0) - 2 = 0$$

$$\text{Average waiting time} = (7 + 2 + 0) / 3 = 3 \text{ ms}$$

**Non-Preemptive:**



$$\text{Waiting time for P1} = 0$$

$$\text{Waiting time for P2} = (12 - 0) - 1 = 11$$

$$\text{Waiting time for P3} = (10 - 0) - 2 = 8$$

6. Calculate the average waiting time in
  - a. Preemptive SJF( Shortest Remaining First (SRT) )
  - b. Non-Preemptive SJF

Processes	Burst Time	Arrival time
P1	8	0
P2	4	1
P3	9	2
P4	5	3

Solution :

**Preemptive:**

P1	P2	P4		P1	P3
0	1	5	10	17	26

Waiting time for P1=  $0 + (10 - 1) - 0 = 9$  ms

Waiting time for P2=  $(1-0) - 1 = 0$  ms

Waiting time for P3=  $(17 - 0)-2 = 15$  ms

Waiting time for P4=  $(5 - 0) - 3= 2$  ms

Average waiting time=  $(9 + 0 + 15 + 2)/4 = 6.5$  ms

## Non-Preemptive SJF:

P1	P2	P4	P3	
0	8	12	17	26

Waiting time for P1= 0 ms

Waiting time for P2=  $8 - 1 = 7$  ms

Waiting time for P3=  $17 - 2 = 15$  ms

Waiting time for P4=  $12 - 3 = 9$  ms

Average waiting time=  $(0 + 7 + 15 + 9) / 4 = 7.75$  ms

7. Consider the following set of processes, with the length of CPU burst time given in millisecond.

Processes	Burst Time	Priority
P1	10	3
P2	1	1
P3	2	3
P4	1	4
P5	5	2

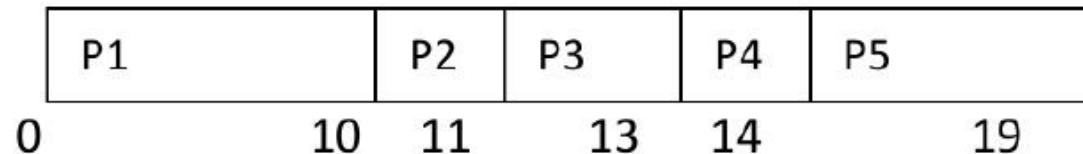
The processes are assumed to have arrived in the order p1, p2, p3, p4, p5 all at time zero.

- i. Draw the Gantt chart illustrating the execution of these processes using FCFS, SJF, priority (Non-Preemptive) and RR (quantum =1) scheduling.
- ii. What is the turnaround time of each process for each of the scheduling algorithm?
- iii. What is the waiting time for each process for each of scheduling algorithm?
- iv. Which of the scheduling algorithm has minimum waiting time?

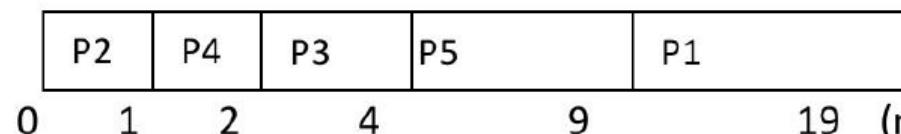
# Exercise:

<https://genuineNotes.com>

## FCFS:



## SJF:

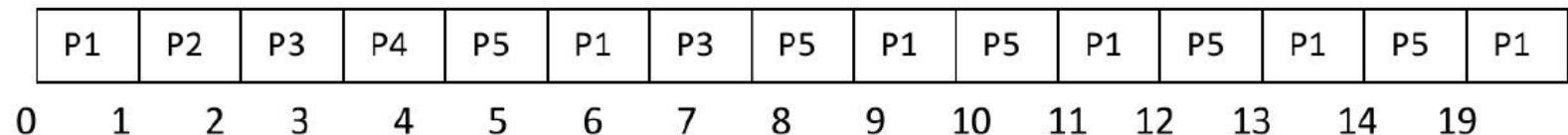


(note: P2 and P4 has same Burst time so use FCFS)

## Non-Preemptive Priority:



## Round Robin:



Average waiting time for FCFS =  $(0 + 10 + 13 + 11 + 14) / 5 = 9.6 \text{ ms}$

Average waiting time for SJF =  $(9 + 0 + 2 + 1 + 4) / 5 = 3.2 \text{ ms}$

Average waiting time for Priority (Non-Preemptive) =  $(6 + 0 + 16 + 18 + 1) / 5 = 8.2$

Average waiting time for RR =  $(9 + 1 + 5 + 3 + 9) / 5 = 5.8$

Therefore SJF scheduling algorithm has the minimum waiting time.

8. Consider a set of processes P1, P2, and P3 and their CPU burst time, priorities and arrived time being mentioned as below

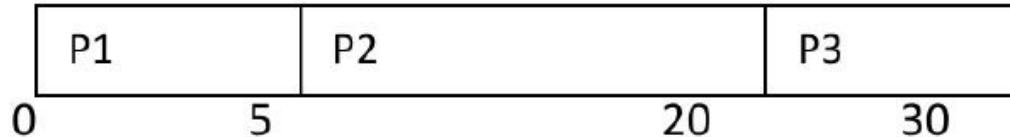
Process	CPU Burst Time	Arrival Time	Priority
P1	5	0	2
P2	15	1	3
P3	10	2	1

Assuming 1 to be the highest priority calculate

- i. Average waiting time using the FCFS, SJF (Preemptive and Non-Preemptive) and priority (Preemptive and Non Preemptive) and Preemptive FCFS (RR).
- ii. Average turnaround time using all the above scheduling mechanism.
- iii. Assume time quantum to be 2 unit of time.

Solution :

**FCFS:**



**Waiting Time:**

Waiting time for P1 = 0 ms

Waiting time for P2 =  $(5 - 0) - 1 = 4$  ms

Waiting time for P3 =  $(20 - 0) - 2 = 18$  ms

Average waiting time =  $(0 + 4 + 18) / 3 = 7.33$  ms

**Turn around Time:**

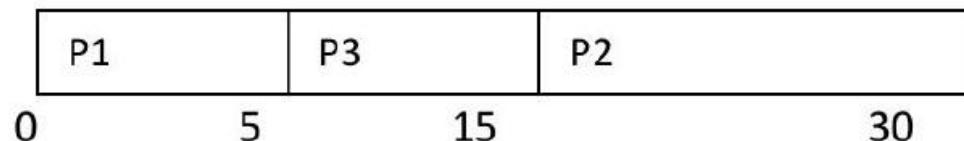
Turn around time for P1 =  $5 - 0 = 5$  ms

Turn around time for P2 =  $20 - 1 = 19$  ms

Turn around time for P3 =  $30 - 2 = 28$  ms

Average Turn around Time =  $(5 + 19 + 28) / 3 = 17.33$  ms

## SJF (Preemptive):



### Waiting Time:

Waiting time for P1 = 0 ms

Waiting time for P2 =  $(15 - 0) - 1 = 14$  ms

Waiting time for P3 =  $(5 - 0) - 2 = 3$  ms

Average waiting time =  $(0 + 14 + 3) / 3 = 5.66$  ms

### Turn around Time:

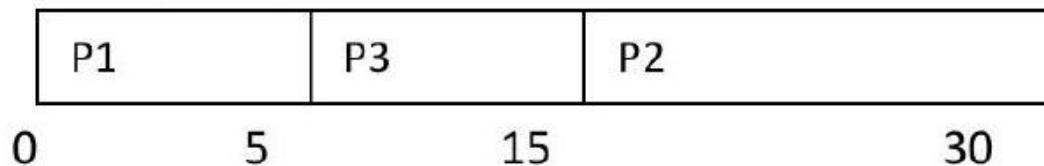
Turn around time for P1 =  $5 - 0 = 5$  ms

Turn around time for P2 =  $30 - 1 = 29$  ms

Turn around time for P3 =  $15 - 2 = 13$  ms

Average Turn around Time =  $(5 + 29 + 13) / 3 = 15.66$  ms

## SJF (Non-Preemptive):



### Waiting Time:

Waiting time for P1 = 0 ms

Waiting time for P2 =  $(15 - 0) - 1 = 14$  ms

Waiting time for P3 =  $(5 - 0) - 2 = 3$  ms

Average waiting time =  $(0 + 14 + 3) / 3 = 5.66$  ms

### Turn around Time:

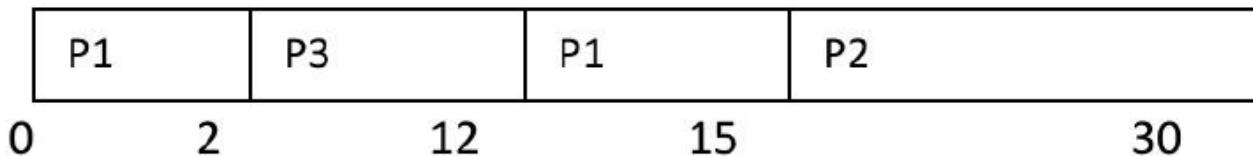
Turn around time for P1 =  $5 - 0 = 5$  ms

Turn around time for P2 =  $30 - 1 = 29$  ms

Turn around time for P3 =  $15 - 2 = 13$  ms

Average Turn around Time =  $(5 + 29 + 13) / 3 = 15.66$  ms

## Priority (Preemptive)



Waiting Time:

$$\text{Waiting time for P1} = 0 + (12 - 2) - 0 = 10 \text{ ms}$$

$$\text{Waiting time for P2} = (15 - 0) - 1 = 14 \text{ ms}$$

$$\text{Waiting time for P3} = (2 - 0) - 2 = 0 \text{ ms}$$

$$\text{Average waiting time} = (10 + 14 + 0) / 3 = 8 \text{ ms}$$

Turn around Time:

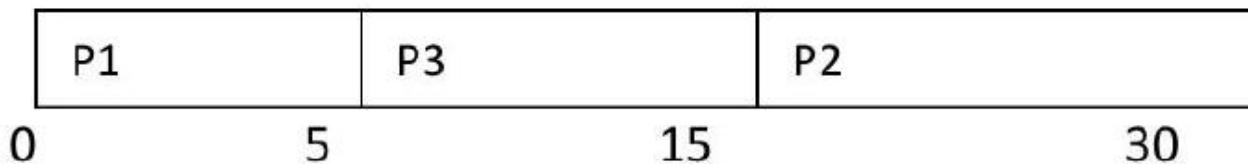
$$\text{Turn around time for P1} = 15 - 0 = 15 \text{ ms}$$

$$\text{Turn around time for P2} = 30 - 1 = 29 \text{ ms}$$

$$\text{Turn around time for P3} = 12 - 2 = 10 \text{ ms}$$

$$\text{Average Turn around Time} = (15 + 29 + 10) / 3 = 18 \text{ ms}$$

## Priority (Non-Preemptive)



### Waiting Time:

Waiting time for P1 = 0 ms

Waiting time for P2 =  $(15 - 0) - 1 = 14$  ms

Waiting time for P3 =  $(5 - 0) - 2 = 3$  ms

Average waiting time =  $(0 + 14 + 3) / 3 = 5.66$  ms

### Turn around Time:

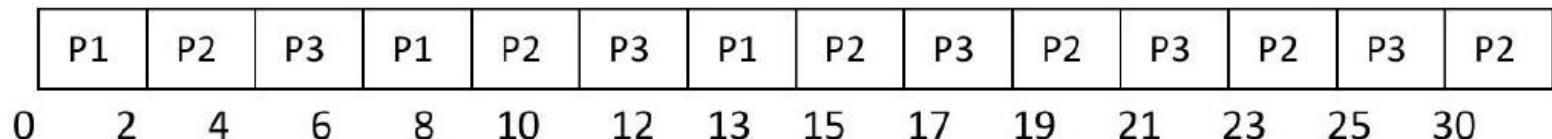
Turn around time for P1 =  $5 - 0 = 5$  ms

Turn around time for P2 =  $30 - 1 = 29$  ms

Turn around time for P3 =  $15 - 2 = 13$  ms

Average Turn around Time =  $(5 + 29 + 13) / 3 = 15.66$  ms

## Preemptive FCFS (Round Robin)



Waiting Time:

$$\text{Waiting time for P1} = 0 + (6 - 2) + (12 - 8) - 0 = 8 \text{ ms}$$

$$\text{Waiting time for P2} = (2 - 0) + (8 - 4) + (13 - 10) + (17 - 15) + (21 - 19) + (25 - 23) - 1 = 14$$

$$\text{Waiting time for P3} = (4 - 0) + (10 - 6) + (15 - 12) + (19 - 17) + (23 - 21) - 2 = 13$$

$$\text{Average waiting time} = (8 + 14 + 13) / 3 = 11.66 \text{ ms}$$

Turn around Time:

$$\text{Turn around time for P1} = 13 - 0 = 13 \text{ ms}$$

$$\text{Turn around time for P2} = 30 - 1 = 29 \text{ ms}$$

$$\text{Turn around time for P3} = 25 - 2 = 13 \text{ ms}$$

$$\text{Average Turn around Time} = (13 + 29 + 13) / 3 = 18.33 \text{ ms}$$

9

For the given process arriving , in the order given with the length of CPU burst time in millisecond

Process	Burst Time	Arrival Time	priority
P1	3	0	2
P2	6	2	4
P3	4	4	5
P4	5	6	3
P5	2	8	1

Consider the FCFS, SJF,SRT Priority (Preemptive and non preemptive), RR (Q=1) RR(Q=4) and HRN scheduling algorithm for this set of process. Which algorithm would give the maximum average waiting time? Also calculate turnaround time for each.

# Real-Time Scheduling Algorithms

- In the simplest real-time systems, where the tasks and their execution times are all known, there might not even be a scheduler. One task might simply call (or yield to) the next. This model makes a great deal of sense in a system where the tasks form a producer/consumer pipeline (e.g. MPEG frame receipt, protocol decoding, image decompression, display).
- In more complex real-time system, with a larger (but still fixed) number of tasks that do not function in a strictly pipeline fashion, it may be possible to do *static* scheduling. Based on the list of tasks to be run, and the expected completion time for each, we can define (at design or build time) a fixed schedule that will ensure timely execution of all tasks.

# Real-Time Scheduling Algorithms

But for many real-time systems, the work-load changes from moment to moment, based on external events. These require *dynamic* scheduling. For *dynamic* scheduling algorithms, there are two key questions:

1. how they choose the next (ready) task to run
  - shortest job first
  - static priority ... highest priority ready task
  - soonest start-time deadline first (ASAP)
  - soonest completion-time deadline first (slack time)
2. how they handle overload (infeasible requirements)
  - best effort
  - periodicity adjustments ... run lower priority tasks less often.
  - work shedding ... stop running lower priority tasks entirely.

# Real-Time Scheduling Algorithms

Preemption may also be a different issue in real-time systems. In ordinary time-sharing, preemption is a means of improving mean response time by breaking up the execution of long-running, compute-intensive tasks. A second advantage of preemptive scheduling, particularly important in a general purpose timesharing system, is that it prevents a buggy (infinite loop) program from taking over the CPU. The trade-off, between improved response time and increased overhead (for the added context switches), almost always favors preemptive scheduling. This may not be true for real-time systems:

- preempting a running task will almost surely cause it to miss its completion deadline.
- since we so often know what the expected execution time for a task will be, we can schedule accordingly and should have little need for preemption.
- embedded and real-time systems run fewer and simpler tasks than general purpose time systems, and the code is often much better tested ... so infinite loop bugs are extremely rare.

# Finished Unit 2