Operating System

BIM IV Semester

Credits: 3
Lecture Hours:48

Er. Santosh Bhandari,
(Master Computer Science)

Unit-2

Process and thread

Process and thread

-A process is a program in execution.
-The execution of a process must progress in a sequential manner.
-A process is defined as a sequence of instructions executed in a predefined order.
-A process is defined as an entity which represents the basic unit of work to be implemented in the system.
-Processes change their state as they execute and can be either new, ready, running, waiting or terminated.
-A process in OS is managed by the Process Control Block (PCB).
PCB is a data structure that contains information about the process, such as its state, priority, and memory usage.

Process and thread

What is thread?

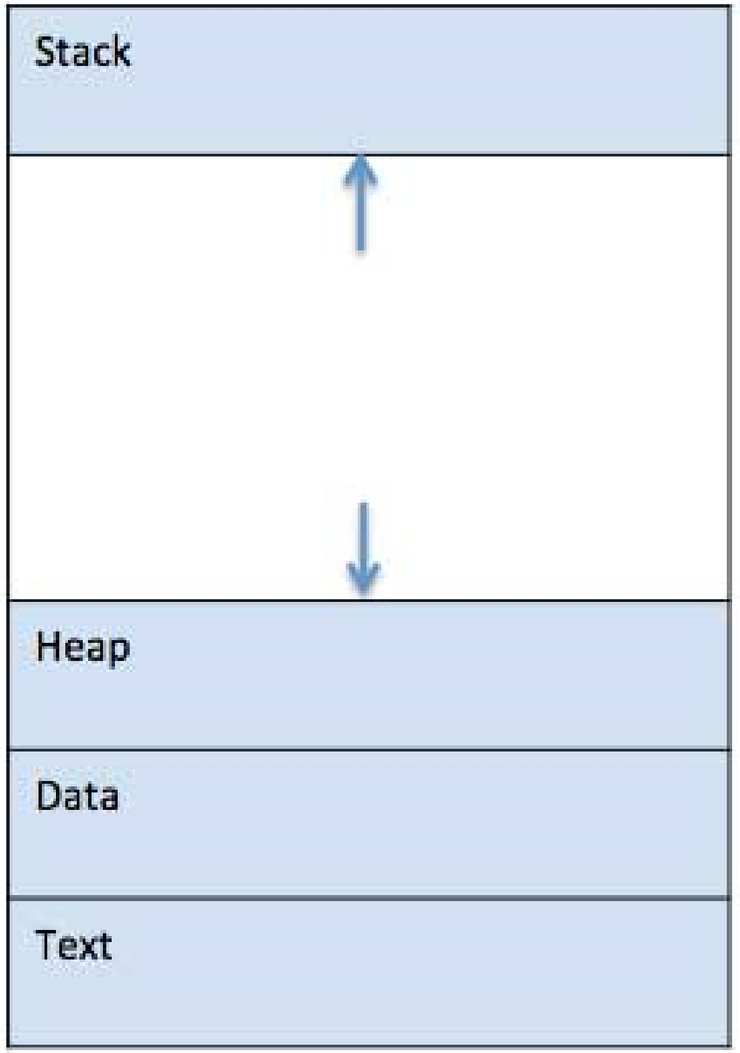
-Unit of execution within a process. A process can have one or many threads.

Process

-When a program is loaded into the memory and it becomes a process, it can be divided into four sections – stack, heap, text and data.

Stack: The process Stack contains the 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.

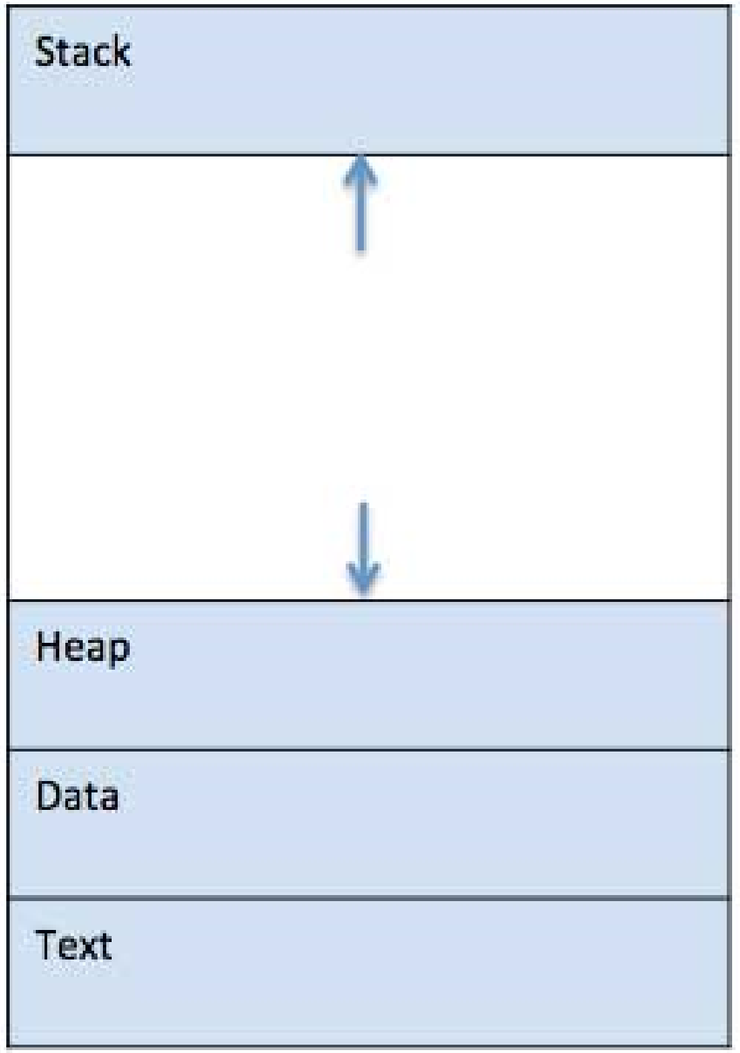


Process

Program code/Text: The instructions that the process will execute. This includes the current activity represented by the value of Program Counter and the contents of the processor's registers.

-The program counter provides the address of the next instruction to be executed by the current process.

Data: This section contains the global and static variables.



Program

-Program is a set of instructions.
-A computer program is written by a computer programmer.
-The programs are usually written in a Programming Language like C, C ++, Python, Java, R, C # (C sharp), etc.

-The step-by-step process of constructing a program is an **algorithm.**

```
#include <stdio.h>

int main() {
   printf("Hello, World! \n");
   return 0;
}
```

Process vs. Program

Program	Process
Program contains a set of instructions designed to complete a specific task.	Process is an instance of an executing program.
Program is a passive entity as it resides in the secondary memory.	Process is a active entity as it is created during execution and loaded into the main memory.
Program is a static entity.	Process is a dynamic entity.

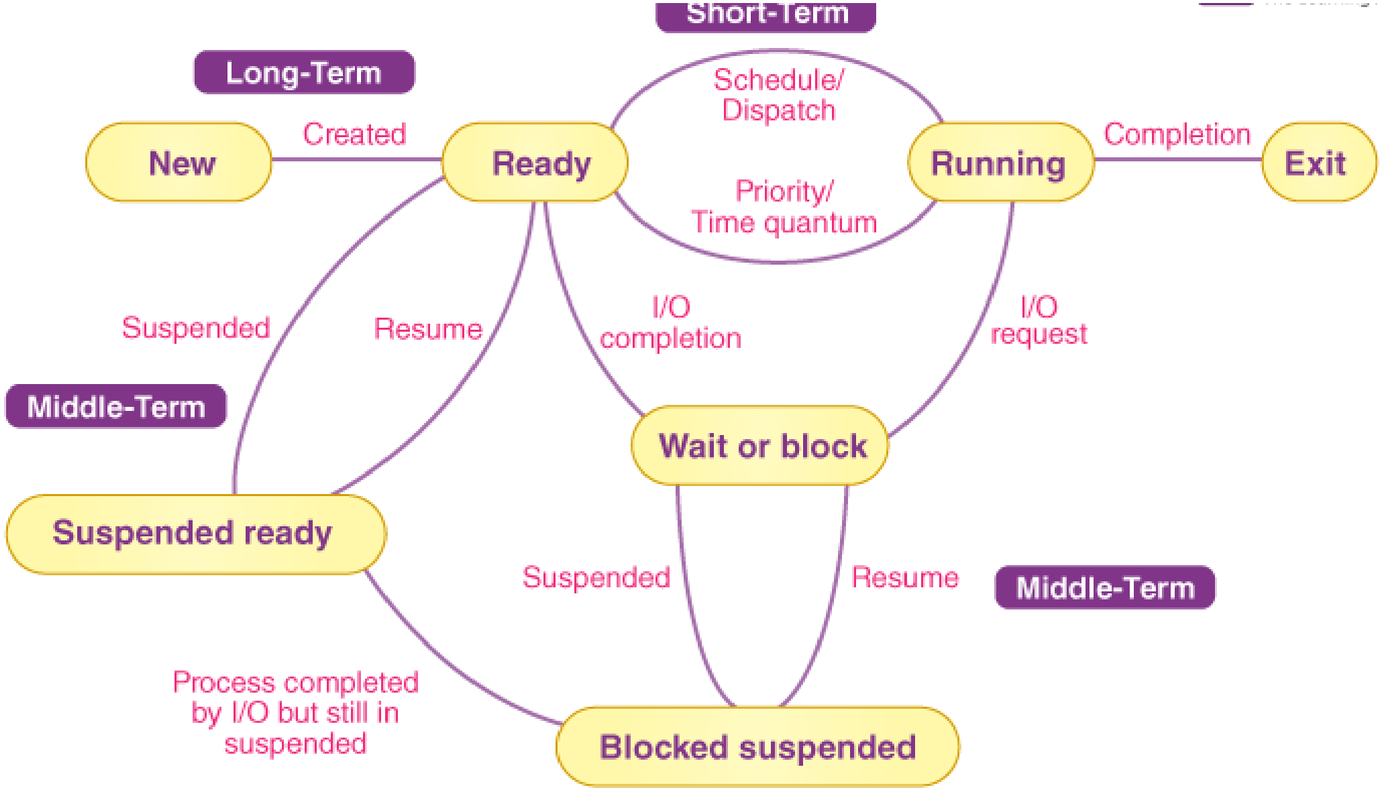
Process vs. Program

Program	Process
Program does not have any resource requirement, it only requires memory space for storing the instructions.	Process has a high resource requirement, it needs resources like CPU, memory address, I/O during its lifetime.
Program contains instructions	Process is a sequence of instruction execution.

Process Life Cycle/Process Model

When a process executes, it passes through different states.

1. Start/new: This is the initial state when a process is first started/created.



Process Life Cycle

2. Ready: The process is waiting to be assigned to a processor. Ready processes are waiting to have the processor allocated to them by the operating system so that they can run. Long-term scheduler works here.

3. Running: Once the process has been assigned to a processor by the OS scheduler, the process state is set to running and the processor executes its instructions.

4. Waiting: The process moves into the waiting state if it needs to wait for a resource, such as waiting for user input, or waiting for a file to become available.

Process Life Cycle

5. Terminated or Exit: Once the process finishes its execution, or it is terminated by the operating system, it is moved to the terminated state where it waits to be removed from main memory.

Additional states:

1. Suspend wait/block: if the wait state is full then, the process is sent to secondary memory for waiting.

2. Suspend ready: when ready is full, it sends to suspend ready state.

Process Life Cycle

We have many processes ready to run. There are two types of multiprogramming:

Preemption – Process is forcefully removed from CPU. Pre-emotion is also called time sharing or multitasking.
Non-preemption – Processes are not removed until they complete the execution. Once control is given to the CPU for process execution, till the CPU releases the control by itself, control cannot be taken back forcibly from the CPU.

Process Scheduling

Process scheduling is the activity of the process manager that handles the removal of the running process from the CPU and the selection of another process based on a particular strategy.

CPU Scheduling takes place when the process:

1. Switches from running to waiting state (preemptive)
2. Switches from running to ready state (preemptive)
3. Switches from waiting to ready state (non-preemptive)
4. Terminates (Non-Preemptive)

1 and 2 are preemptive and others are non preemptive

Schedulers

Process Scheduling is an integral part of Multi-programming applications. Such operating systems allow more than one process to be loaded into usable memory at a time and the loaded shared CPU process uses repetition time.

There are three types of process schedulers:

1. Long-term or Job Scheduler
2. Short-term or CPU Scheduler
3. Medium-term Scheduler

Schedulers

Types of Schedulers

Long-term scheduler or Job scheduler or High-level Scheduler or Admission Scheduler – performance: Decides how many processes should be made to stay in the ready state. (Created).

-It brings the new process to the ‘Ready State’.

-It controls the Degree of Multi-programming.

-Provides both I/O and CPU-bound processes.

-If the job scheduler selects more I/O bound processes, all of the jobs may become stuck, the CPU will be idle for long time, and multiprogramming will be reduced as a result.

CPU-bound: If the execution of a task or program is highly dependent on the CPU.

I/O bound: if its execution is dependent on the input-output system and its resources, such as disk drives and peripheral devices.

Schedulers

Short-term scheduler or CPU scheduler – Context switching time:

-It chooses one job from the ready queue and then sends it to the CPU for processing.

The short-term scheduler will decide which process is to be executed next and then it will call the dispatcher.

-In other words, it is context switching.

-If it chooses a job with a long CPU burst time, all subsequent jobs will have to wait in a ready queue for a long period. This is known as hunger/starvation.

Schedulers

Medium-term – Swapping time:

-Suspension decision is taken by the medium-term scheduler.

-It is used for swapping (which is moving the process from main memory to secondary and vice versa.)

-It handles the switched-out processes.

-If the running state processes require some IO time to complete, the state must be changed from running to waiting. This is accomplished using a Medium-Term scheduler.

Other Schedulers

I/O schedulers: I/O schedulers manage the execution of I/O operations such as reading and writing to discs or networks.
-Algorithms to determine the order in which I/O operations are executed: FCFS (First-Come, First-Served) or RR (Round Robin).
Real-time schedulers: Real-time schedulers ensure that critical tasks are completed within a specified time frame.
-Used in a real-time system.
They can prioritize and schedule tasks using various algorithms such as EDF (Earliest Deadline First) or RM (Rate Monotonic).

Short term. Vs. Long-Term vs. Medium-term Schedulers

Long Term Scheduler	Short term schedular	Medium Term Scheduler
It is a job scheduler	It is a CPU scheduler	It is a process-swapping scheduler.
Speed is lesser than short-term scheduler	Speed is the fastest among all of them.	Speed lies in between both short and long-term
It controls the degree of multiprogramming	It gives less control over how much multiprogramming is done.	It reduces the degree of multiprogramming.
It is barely present or nonexistent in the time-sharing system.	It is a minimal time-sharing system.	It is a component of systems for time sharing.
Selects processes from the pool and then loads them into the memory for execution.	It selects those processes which are ready to execute	It can re-introduce the process into memory and execution can be continued.

Possible State Transitions in process model

1. Null → New: A new process is created for execution.

2. New → Ready: The newly created process which is ready to execute on the CPU and is waiting for it to get free is moved here in the ready state. There can be multiple processes in these states at a time.

3. Ready → Running: The Selected process is moved into the Running state where it will get the CPU for executing its tasks. There can be at most one process in the Running state at a time.

Possible State Transitions in process model

4. Running → Exit: The process whose execution gets over will be terminated and moved into the Exit state.

5. Running → Ready: When the process has reached its maximum time limit of execution or when a high priority process came for execution, the process currently present in the running state will be moved into the ready state.

6. Running → Blocked: When the process is waiting for some event to occur the process will be moved to the Blocked state from the Running state.

Possible State Transitions in process model

7. Blocked → Ready: A process will be moved back into the ready state when the event that it has been waiting for occurs.

8. Ready → Exit: This will only happen when the parent process of the current process gets terminated or it requests explicitly to terminate the child process.

9. Blocked → Suspend: When the Blocked queue gets filled with the processes then some of the processes from the blocked queue will be moved into the Suspend state. Suspend state exists in the secondary memory by using the concept of virtual memory.

Possible State Transitions in process model

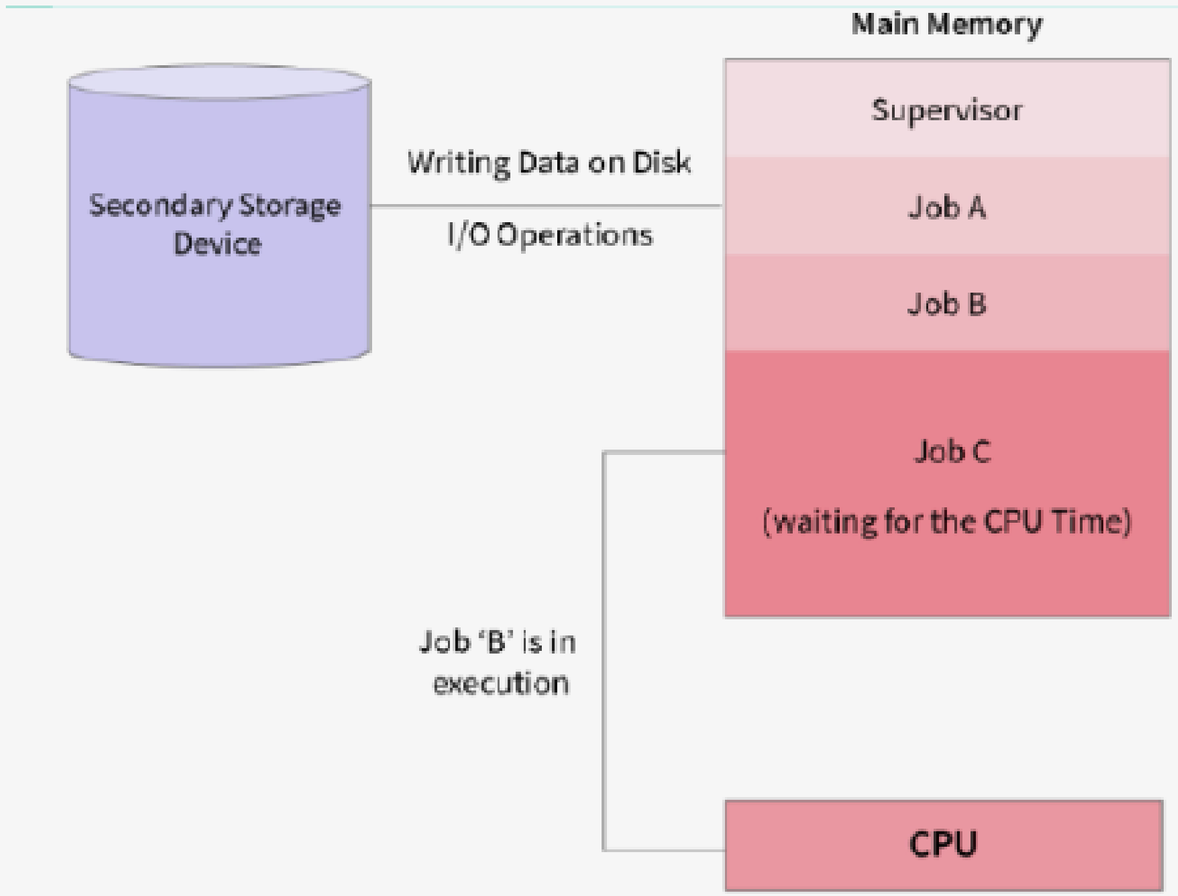
10. Suspend → Blocked: When the Space gets available into the Blocked state then the process that has been put into the Suspend state will be moved back into the Blocked State.

11. Suspend → Ready: When the event has occurred for the process that has been waiting into the suspend state in the secondary memory then it will be directly moved into the Ready state.

Multiprogramming

-A multiprogramming operating system may run many programs on a single processor computer.
-If one program must wait for an input/output transfer the other programs are ready to use the CPU.
-Various jobs may share CPU time.
-Execution of their jobs is not defined to be at the same time period.
-When a program is being performed, it is known as a **"Task", "Process", and "Job".**
-The CPU won't be idle in a multiprogramming OS.

Multiprogramming



Types of Multiprogramming

Two types:

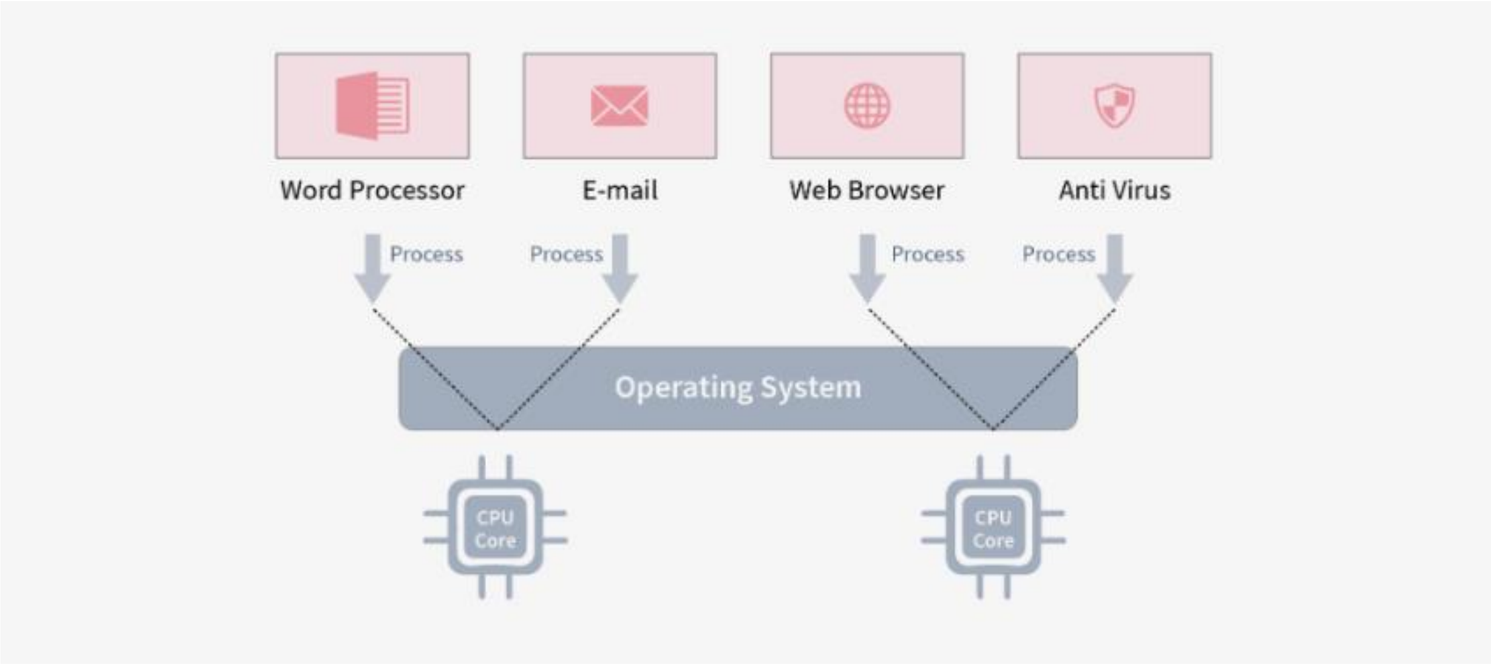
1. Multitasking Operating System
2. Multiuser Operating System

Types of Multiprogramming

Two types:

1. **Multitasking OS**

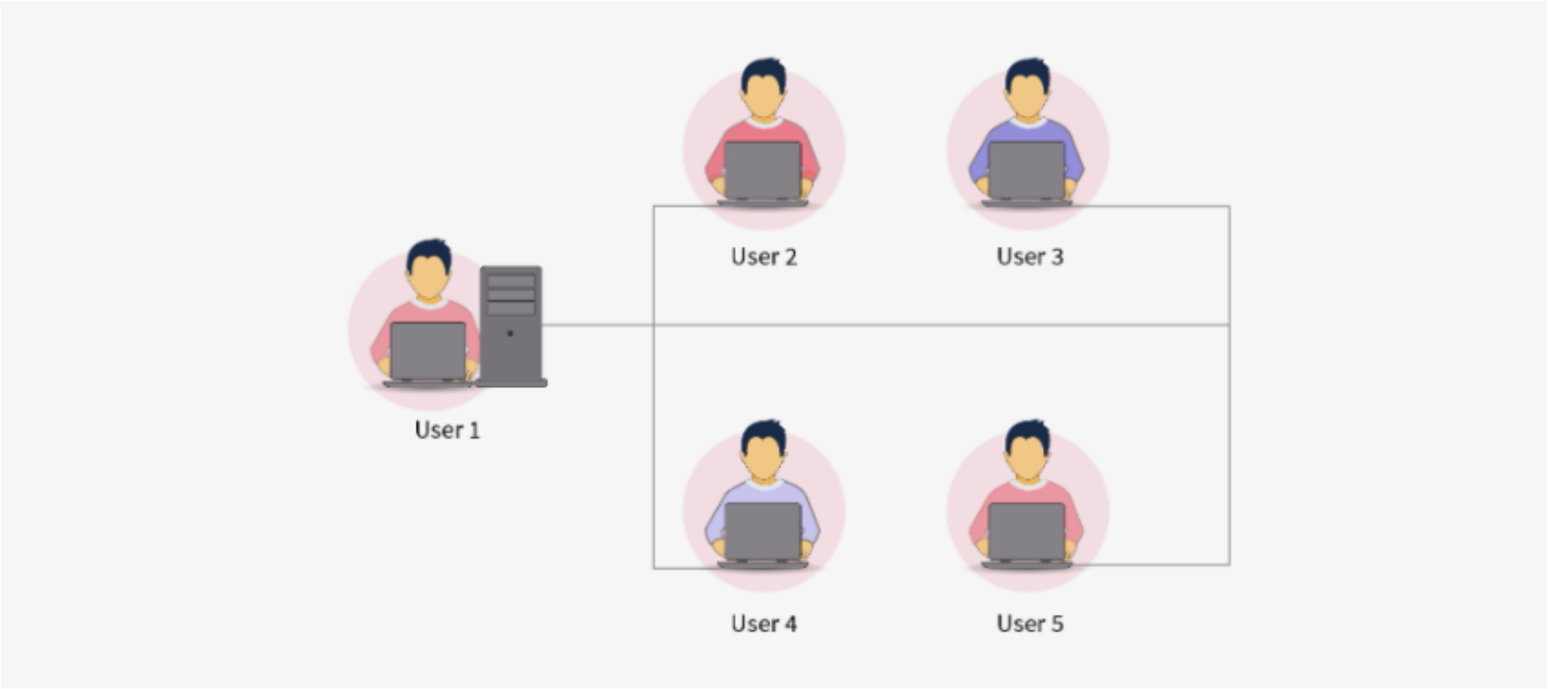
A multitasking operating system enables the execution of two or more programs at the same time.



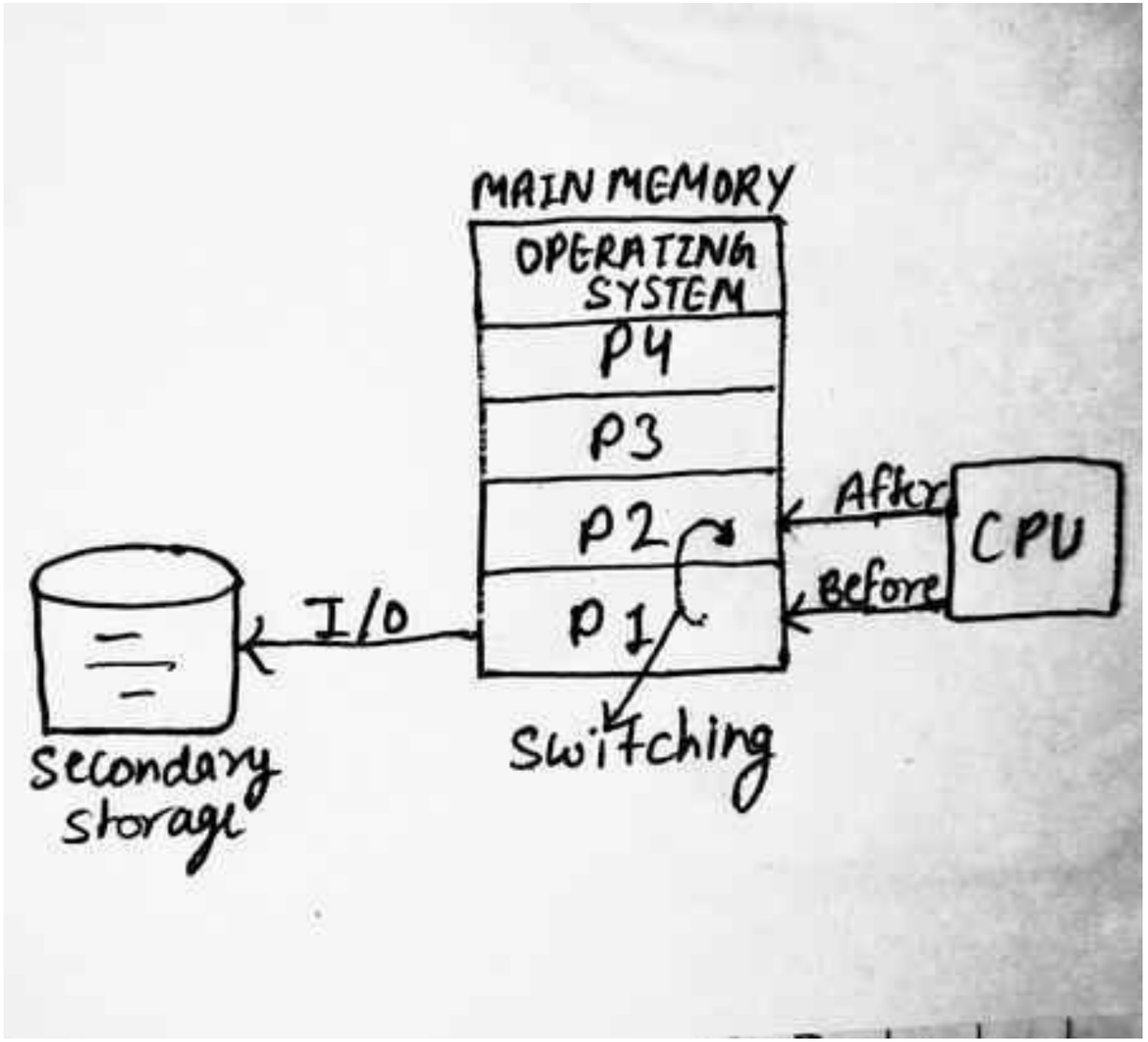
Types of Multiprogramming

2. Multiuser OS

A multiuser OS allows many users to share processing time on a central computer from different terminals.



How Multiprogramming works?



Advantages of Multiprogramming

1. CPU utilization is high because the CPU is never goes to idle state.

2. Memory utilization is efficient.

3. CPU throughput is high and also supports multiple interactive user terminals.

Disadvantages of Multiprogramming

1. CPU scheduling is compulsory because lots of jobs are ready to run on CPU simultaneously.

2. If it has a large number of jobs, then long-term jobs will require a long wait.

3. Programmers also cannot modify a program that is being executed.

Process Creating

A process can create several new processes through creating process system calls during the process execution.

-In Linux Process creation is achieved through the **fork() system call**.

-In Windows, the **CreateProcess()** function is commonly used to create a new process

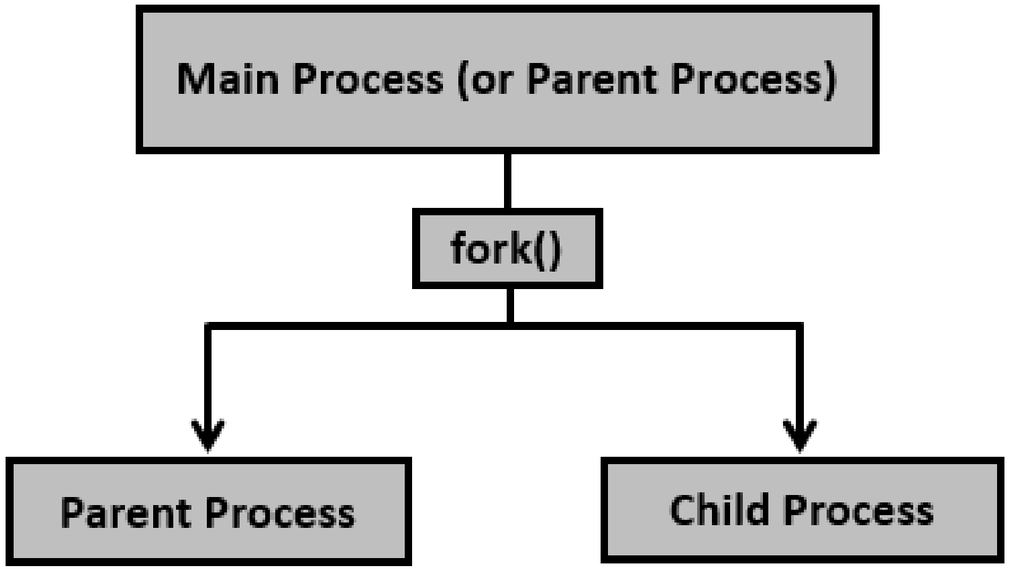
-The newly created process is called the **child process** and the process that initiated it (or the process when execution is started) is called the **parent process**.

-Every new process creates another process forming a tree-like structure.

Process Creating

-A unique process ID is allocated to each process when it is created

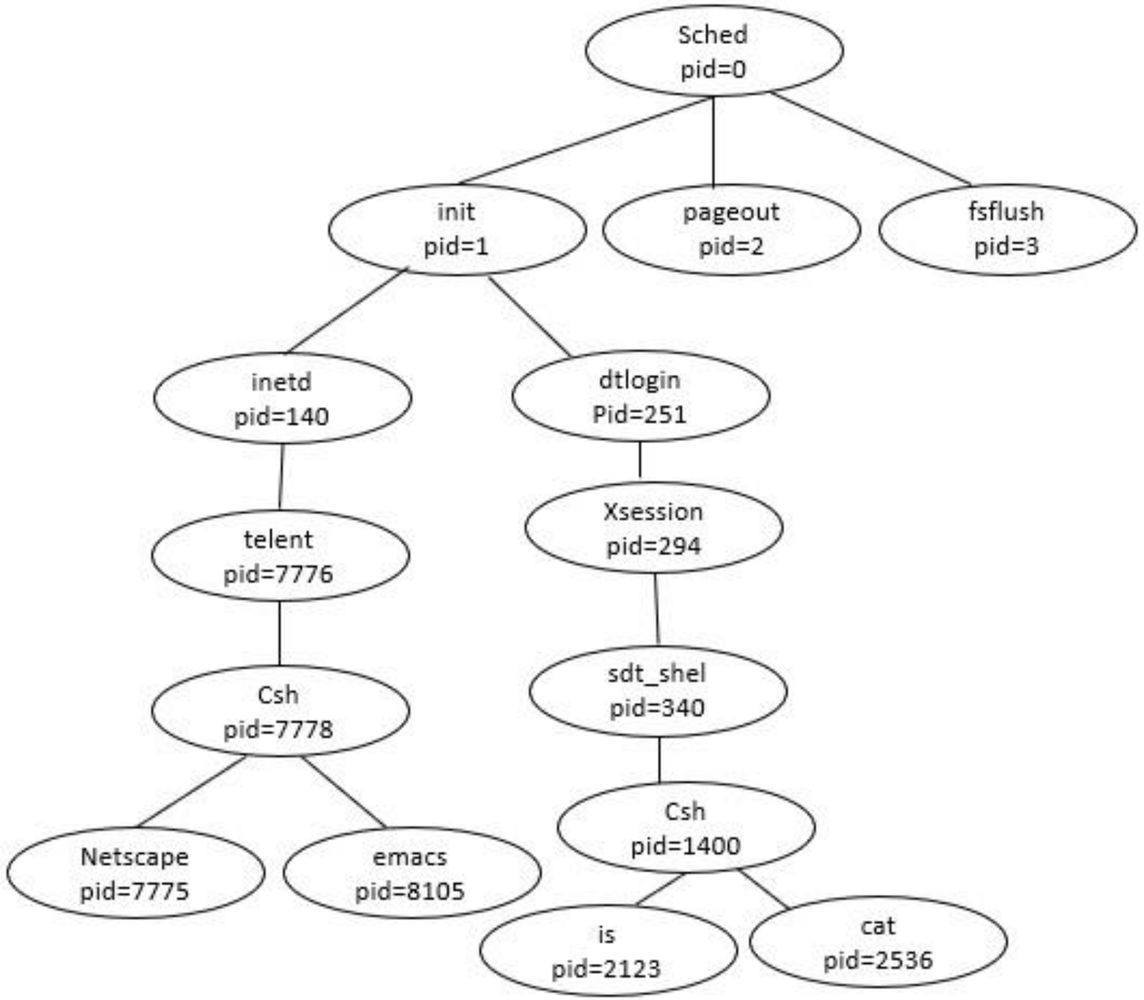
-The child process created by fork is a copy of the original parent process, except that it has its own process ID.



Tree of process on a Solaris system

-On UNIX systems the process scheduler is termed sched, and is given PID 0.

-The first thing it does at system startup time is to launch init, which gives that process PID 1. Init then launches all system daemons and becomes the ultimate parent of all other processes.



Tree of process on a Solaris system

What is Daemon?

a **daemon** is a type of background process that runs continuously, providing specific services or performing tasks without direct user interaction.

Process Creating

A process that is waiting for its parent to accept its return code is called a **zombie process**.

If a parent dies before its child, the child (orphan process) is automatically adopted by the original “init” process whose PID is 1.

Process Creating

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 Creating

The fork() system call returns either of the three values

1. fork() returns Negative value to indicate an error, i.e., unsuccessful in creating the child process.
2. fork() returns a zero for child process, i.e. successful creation of child process.
3. fork() returns a positive value for the parent process. This value is the process ID of the newly created child process.

Process Creating

```
#include<stdio.h>
#include<unistd.h>
int main()
{
        pid_t pid;
        pid = fork(); /* fork another process */
        if (pid < 0) { /* error occurred */
                printf(“child process creation is Failed");
                }
        else if (pid == 0)     { /* child process */
        printf(“child process”);
        printf("%d", pid);
                }
        else { /* parent process */
        printf (“parent process");
        printf("%d", pid);
                }
}
```

Process Creating

```
#include <stdio.h>
#include <sys/types.h> : Header file for data types pid_t (process ID type).
#include <unistd.h>  : header file for the fork() system call.
int main() {
   fork();
   printf("Called fork() system call\n");
   return 0;
}
```

Output

Called fork() system call

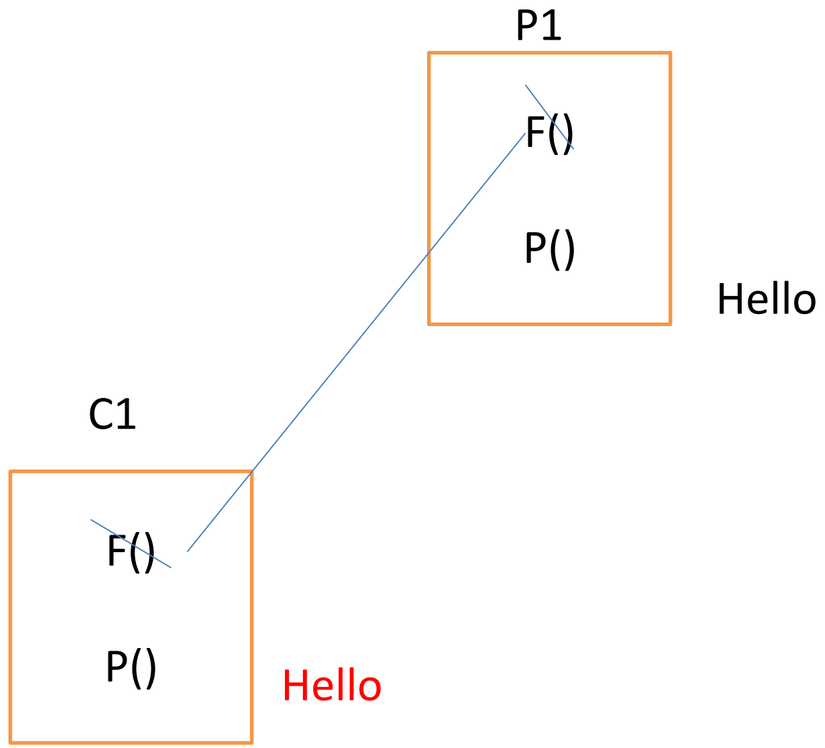
Called fork() system call

Output explanation:

In the above case, fork() is called once, hence the output is printed twice (2 power 1). If fork() is called, 3 times, then the output would be printed 8 times (2 power 3).

Process Creation

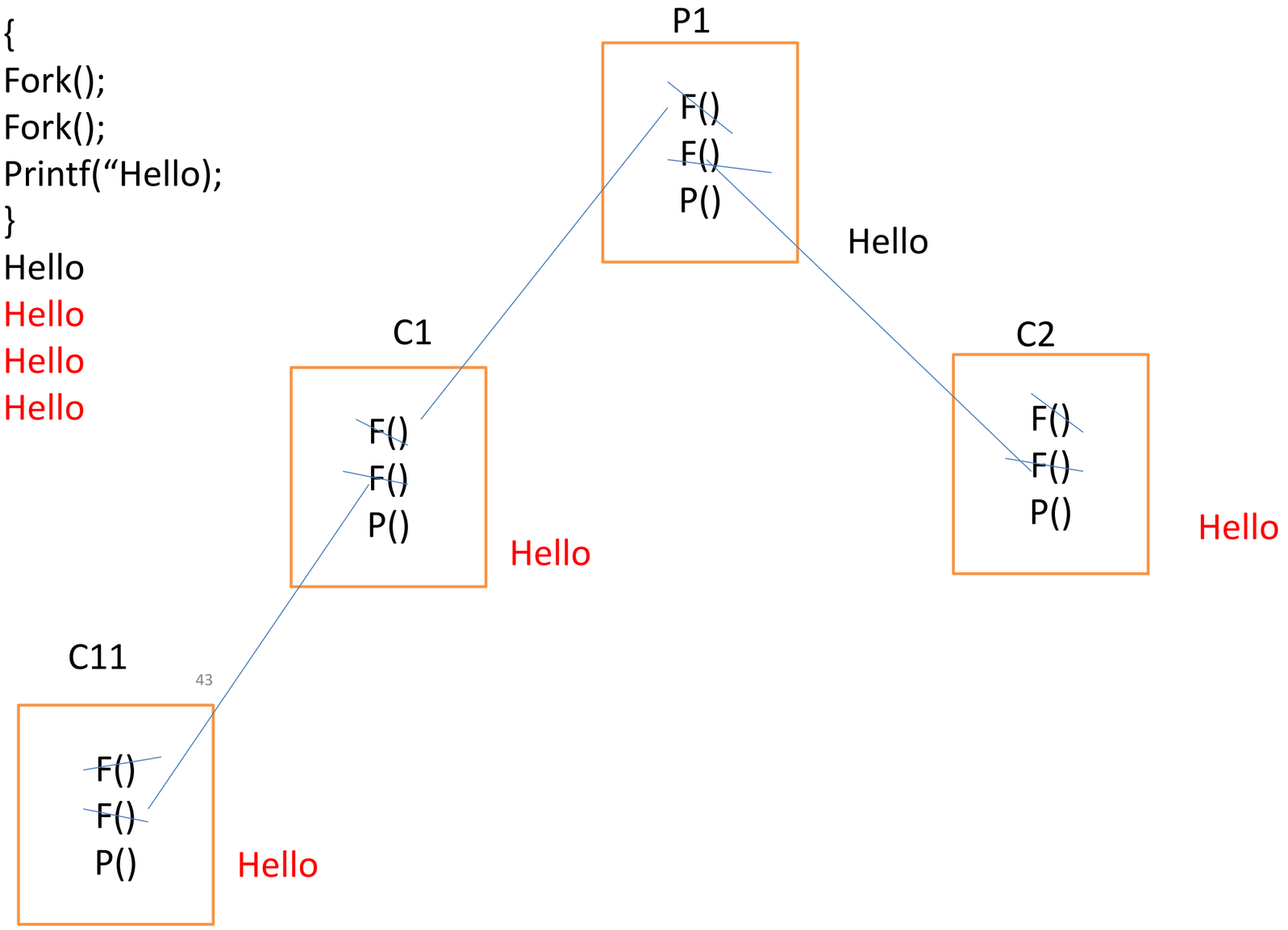
```
{
Fork();
Printf(“Hello);
}
Hello
Hello
```



Process Creation

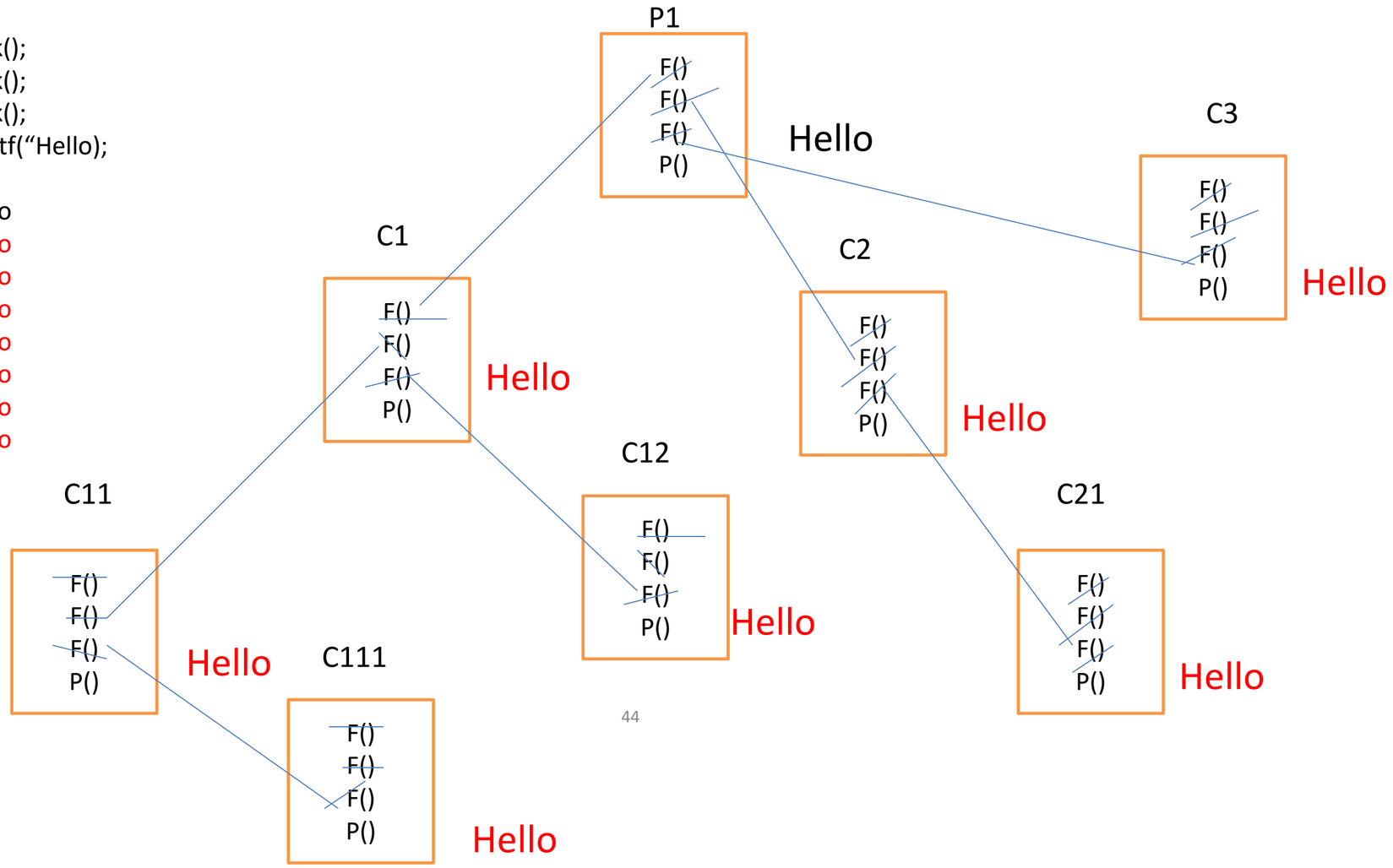
```
{
Fork();
Fork();
Printf(“Hello);
}
```

Hello
Hello
Hello
Hello



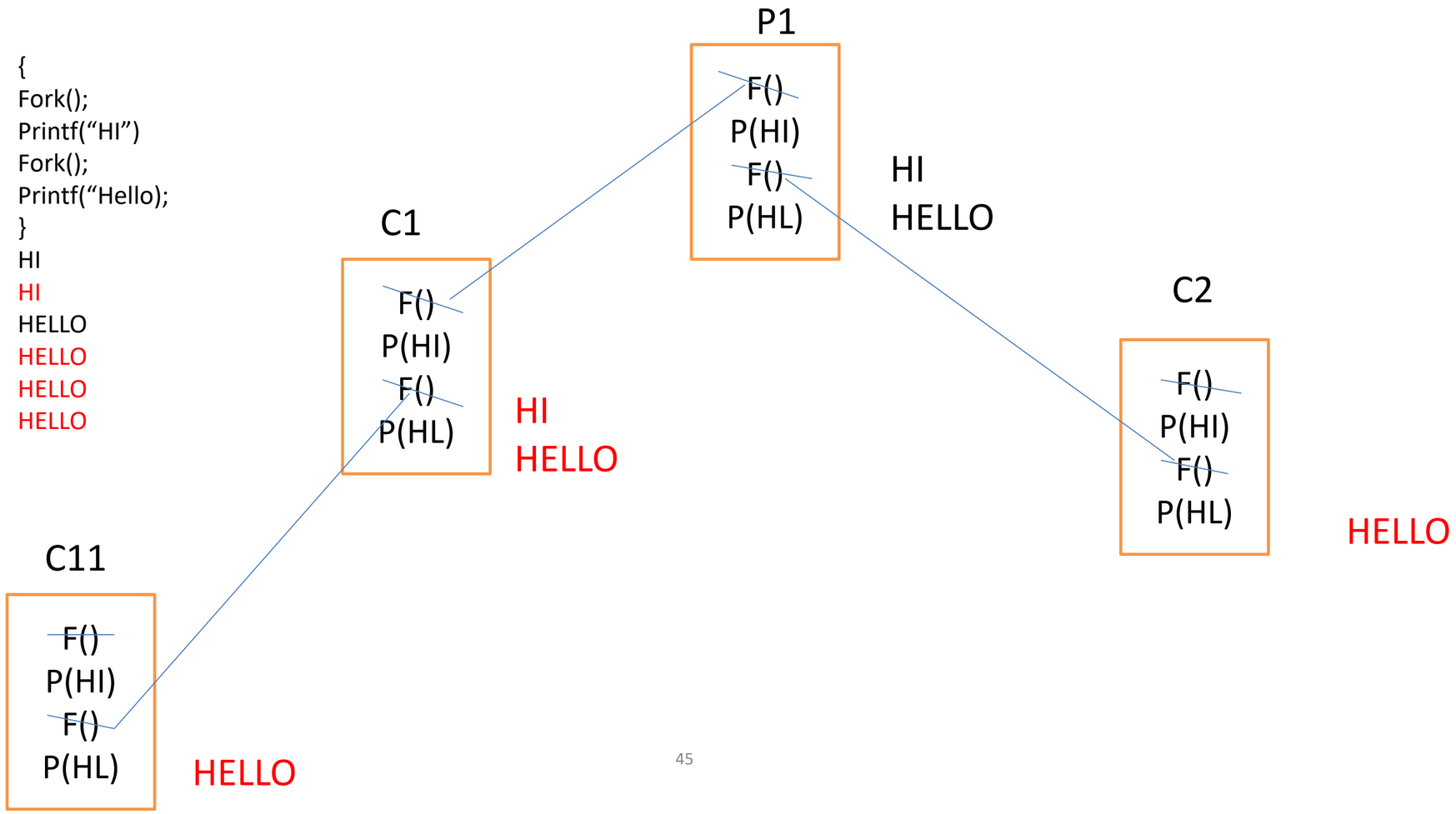
```
{
Fork();
Fork();
Fork();
Printf(“Hello);
}
```

Hello
Hello
Hello
Hello
Hello
Hello
Hello
Hello



```
{
Fork();
Printf(“HI”)
Fork();
Printf(“Hello);
}
```

HI
HI
HELLO
HELLO
HELLO
HELLO



Process Creation..

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

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

Process Control Block (PCB)

-A process in OS is managed by the Process Control Block (PCB).
-A Process Control Block is a data structure maintained by the Operating System for every process. The PCB is identified by an integer process ID (PID).
Following information exists in PCB:
Process State (PS): The current state of the process i.e., New, ready, running, waiting, terminated.
Process privileges: This is required to allow/disallow access to system resources.

Process Control Block (PCB)

Process ID (PID): Unique identification for each of the process in the operating system.

Pointer: A pointer to parent process.

Program Counter (PC): Program Counter is a pointer to the address of the next instruction to be executed for this process.

CPU registers: Various CPU registers where process need to be stored for execution for running state.

Process Control Block (PCB)

Memory management information: This includes the information of page table, memory limits, Segment table depending on memory used by the operating system.

Accounting information: This includes the amount of CPU used for process execution, time limits, execution ID etc.

IO status information: This includes a list of I/O devices allocated to the process.

Process Control Block (PCB): Example

-Use the top or ps command in Linux to check process state.

PID	USER	PR	NI	VIRT	RES	SHR	S	%CPU	%MEM	TIME+	COMMAND
1465	sb	20	0	4749384	335100	133120	S	3.6	14.3	0:12.48	gnome-s+
2003	sb	20	0	553472	51792	39580	S	1.0	2.2	0:00.95	gnome-t+
2116	**sb**	**20**	**0**	**13544**	**4224**	**3456**	**R**	**0.3**	**0.2**	**0:00.56**	**top**
1	root	20	0	166704	11348	8148	S	0.0	0.5	0:01.98	systemd
2	root	20	0	0	0	0	S	0.0	0.0	0:00.00	kthreadd
3	root	0	-20	0	0	0	I	0.0	0.0	0:00.00	rcu_gp
4	root	0	-20	0	0	0	I	0.0	0.0	0:00.00	rcu_par+
5	root	0	-20	0	0	0	I	0.0	0.0	0:00.00	slub_fl+
6	root	0	-20	0	0	0	I	0.0	0.0	0:00.00	netns
8	root	0	-20	0	0	0	I	0.0	0.0	0:00.00	kworker+
9	root	20	0	0	0	0	I	0.0	0.0	0:01.49	kworker+
10	root	0	-20	0	0	0	I	0.0	0.0	0:00.00	mm_perc+
11	root	20	0	0	0	0	I	0.0	0.0	0:00.00	rcu_tas+
12	root	20	0	0	0	0	I	0.0	0.0	0:00.00	rcu_tas+
13	root	20	0	0	0	0	I	0.0	0.0	0:00.00	rcu_tas+
14	root	20	0	0	0	0	S	0.0	0.0	0:00.05	ksoftir+
15	root	20	0	0	0	0	I	0.0	0.0	0:00.21	rcu_pre+

Process Control Block (PCB)

-Use the top command in Linux.

PID (Process ID): The unique identification number assigned to each running process. Processes are listed in order of their PID.

USER: The username of the owner of the process.

PR (Priority): The priority of the process. Higher numerical values indicate lower priority. The priority may be influenced by factors such as nice values and process scheduling.

Process Control Block (PCB)

NI (Nice Value): The "niceness" value of the process, which determines its priority. Higher nice values mean lower priority. Users can adjust the nice value to influence the scheduling priority of a process.

VIRT (Virtual Memory): The total virtual memory used by the process. This includes the process's code, data, and shared libraries, as well as memory that has been swapped out.

RES (Resident Memory): It represents the portion of the process's memory that is held in RAM.

Process Control Block (PCB)

SHR (Shared Memory): The amount of shared memory used by the process. Shared memory is memory that may be used by multiple processes.

S (%CPU): The percentage of CPU time used by the process since the last update. This value is calculated as a percentage of total CPU time.

MEM (%MEM): The percentage of physical RAM used by the process. It is calculated as the ratio of the process's resident set size (RES) to the total physical memory.

Process Control Block (PCB)

TIME+: The total accumulated CPU time used by the process since it started.

COMMAND: The name of the command or executable associated with the process.

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.**

Thread

- A thread is also called a lightweight process.
- Each thread belongs to exactly one process and no thread can exist outside a process.
- The process can be split down into so many threads.

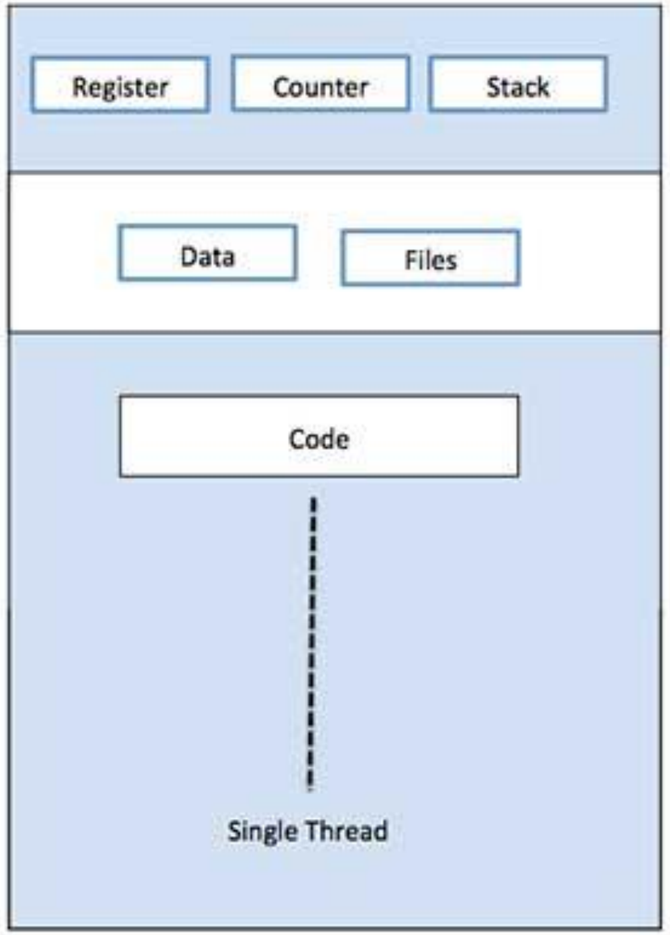
For example, in a browser, many tabs can be viewed as threads. MS Word uses many threads - formatting text from one thread, processing input from another thread, etc.

Thread

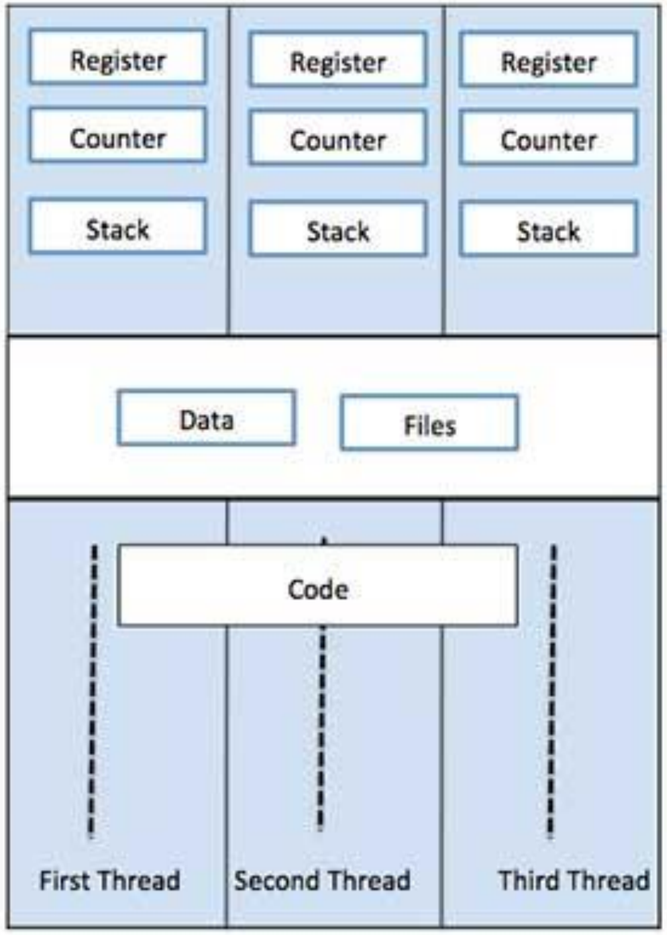
What is shared among threads and what is unique for each thread?

Shared Among Threads	Unique For Each Thread
Code Section	Thread Id
Data Section	Register Set
OS Resources	Stack
Open Files & Signals	Program Counter

Single Thread and Multi thread

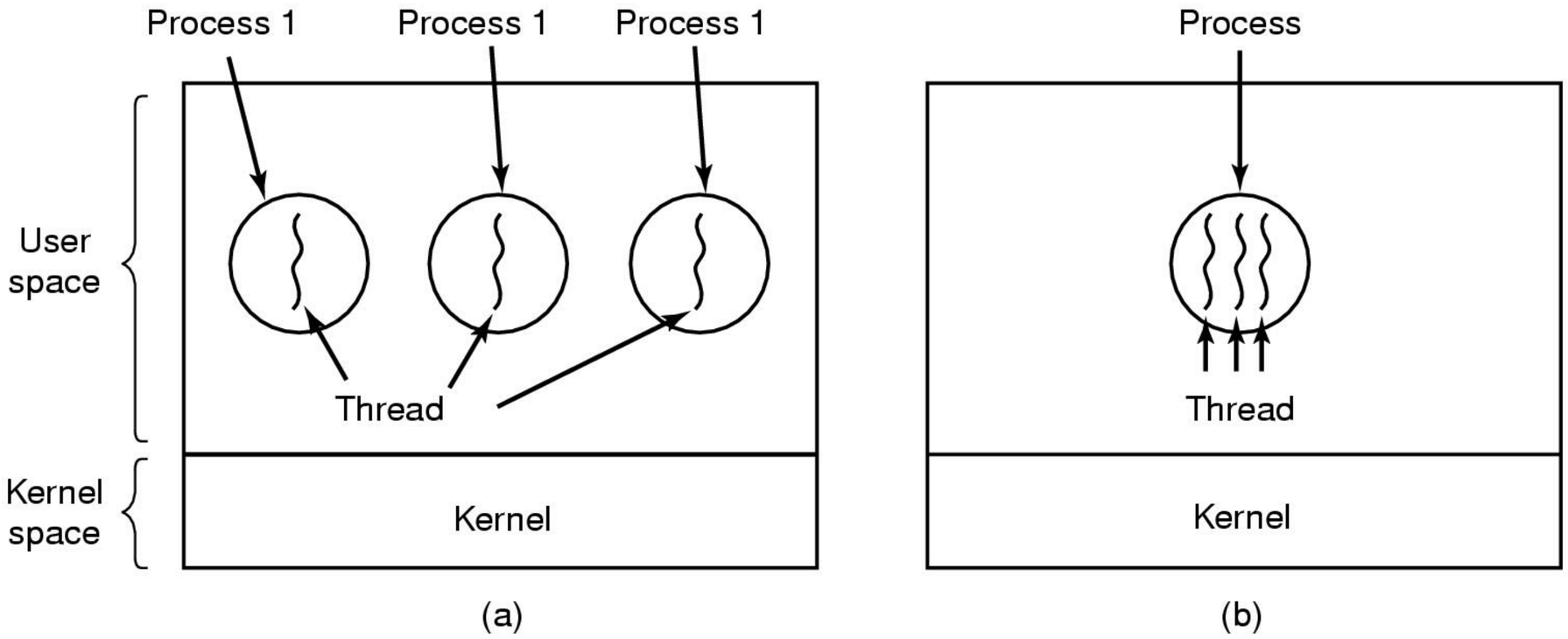


Single Process P with single thread



Single Process P with three threads

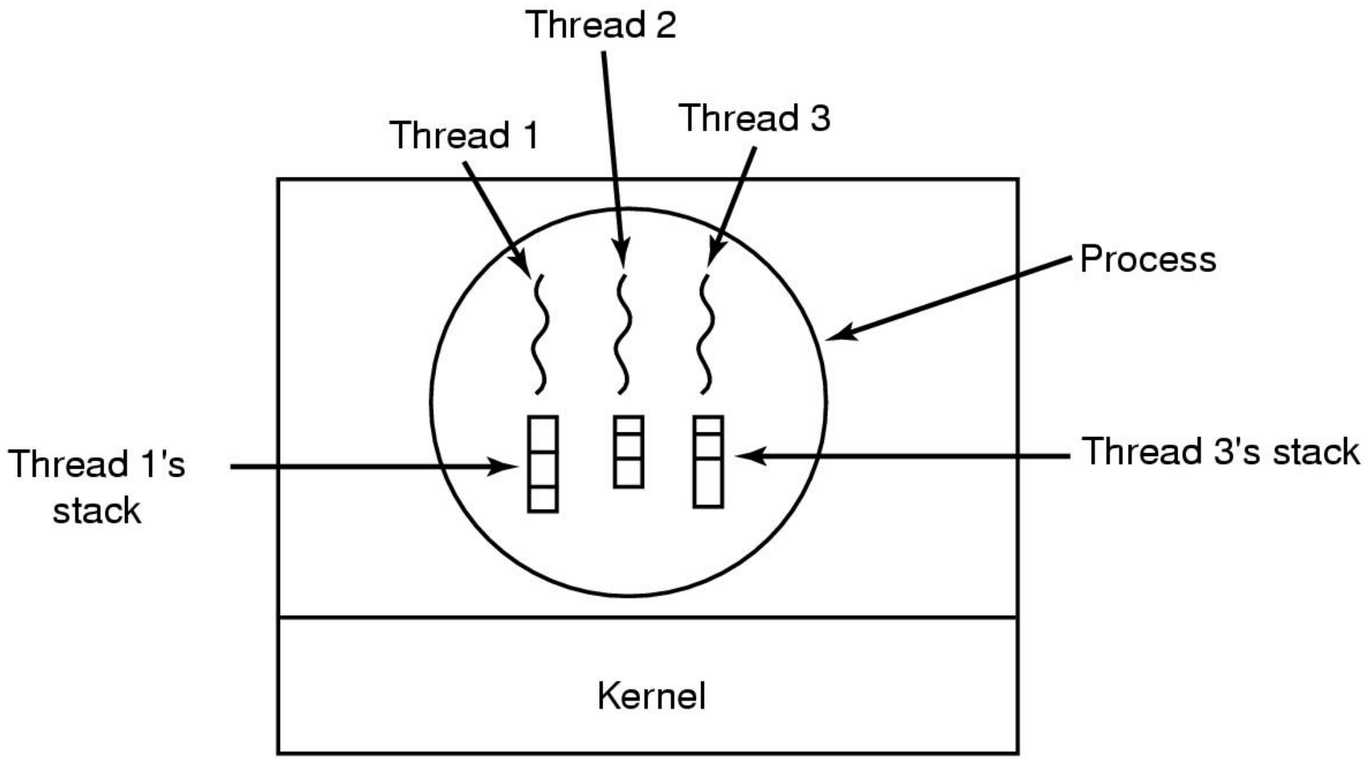
Single Thread and Multi thread



(a) Three processes each with one thread

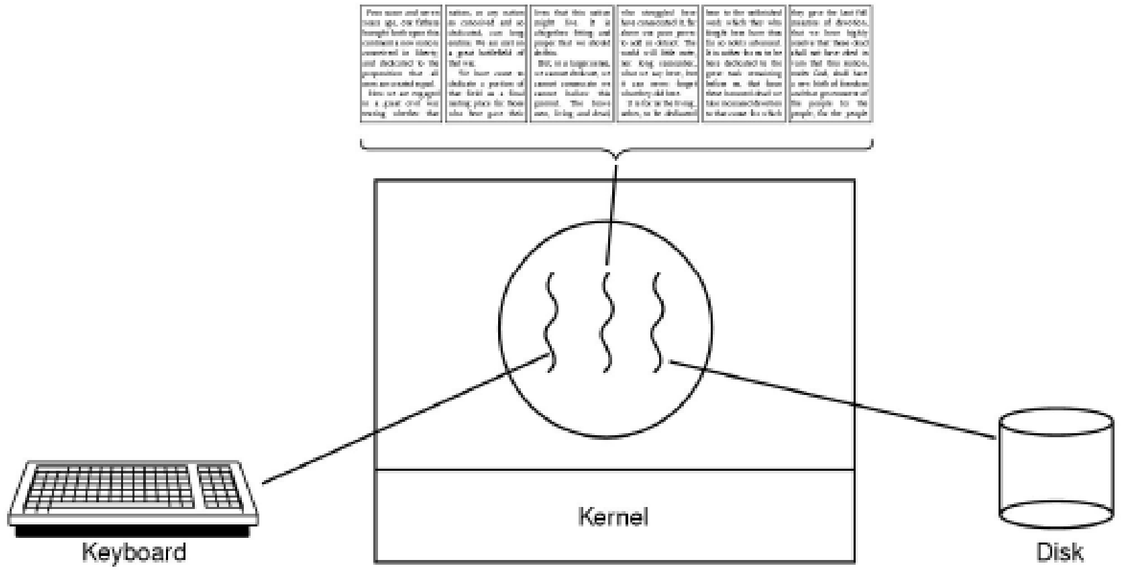
(b) One process with three threads

Single Thread and Multi thread



Each thread has its own stack

Thread usage example



A word processor with three threads

Each thread has its own stack

Process vs. Thread

Process	Thread
Process is heavy weight or resource intensive.	Thread is light weight, taking lesser resources than a process.
Process switching needs interaction with operating system.	Thread switching does not need to interact with operating system.
In multiple processing environments, each process executes the same code but has its own memory and file resources.	All threads can share same set of open files, child processes.
If one process is blocked, then no other process can execute until the first process is unblocked.	While one thread is blocked and waiting, a second thread in the same task can run.
Multiple processes without using threads use more resources.	Multiple threaded processes use fewer resources.

Properties of 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

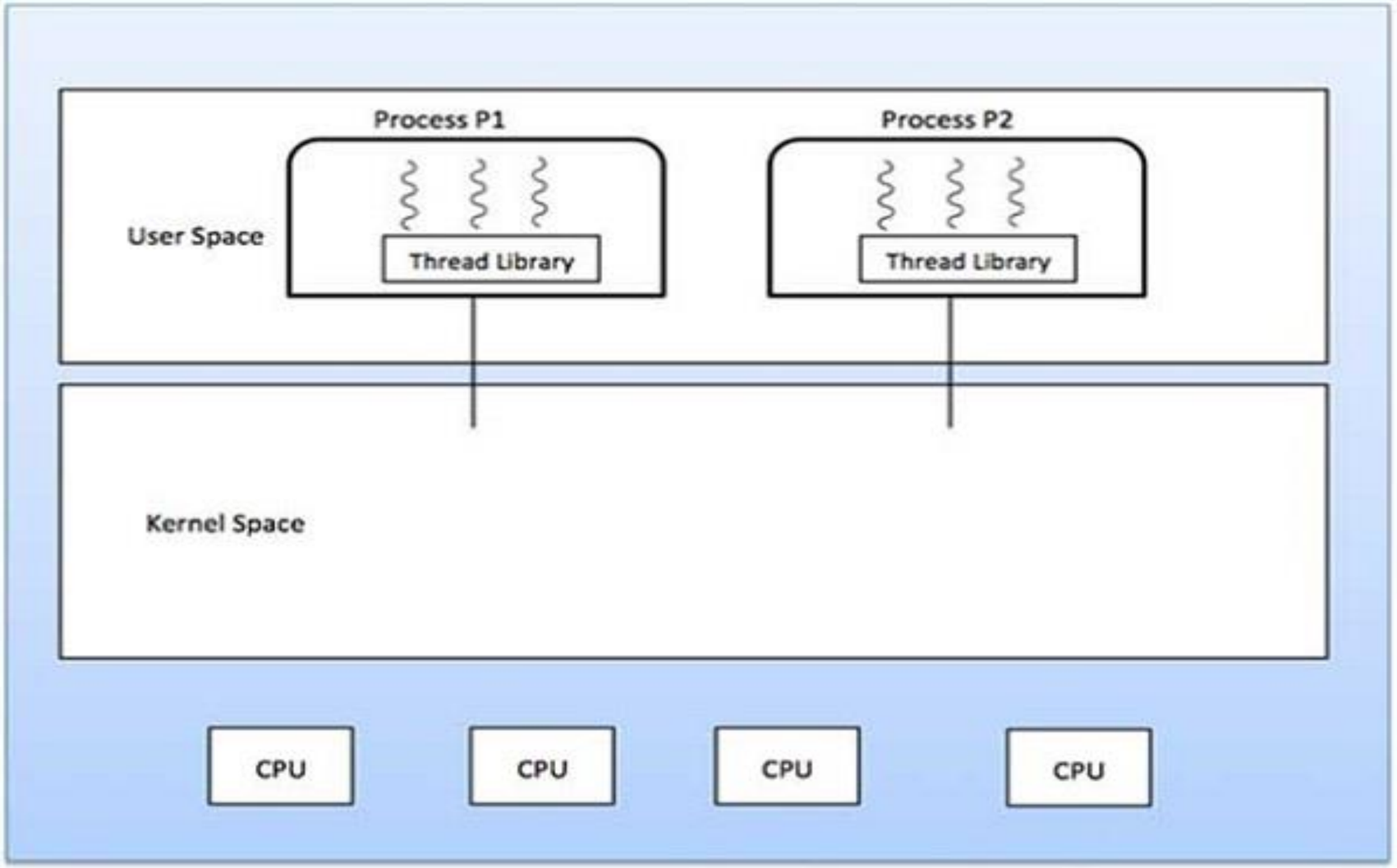
There are two types of threads:

1. User level Threads
2. Kernel level Threads

User level Threads

-Thread that can occurs in user level called user-level thread.
-The operating system does not recognize the user-level thread.
-User threads can be easily implemented and it is implemented by the user.
-The kernel level thread does not know nothing about the user level thread.
examples: Java thread, POSIX threads, etc.

User level Threads



User level Threads

Advantages of User-level threads

- The user threads can be easily implemented than the kernel thread.
- User-level threads can be applied to such types of operating systems that do not support threads at the kernel-level.
- It is faster and efficient.
- Context switch time is shorter than the kernel-level threads.
- It does not require modifications of the operating system.

User level Threads

Disadvantages of User-level threads

- User-level threads lack coordination between the thread and the kernel.
- If a thread causes a page fault, the entire process is blocked.

Kernel level Threads

-The kernel thread recognizes the operating system.
-There is a thread control block and process control block for each thread and process in the kernel-level thread.
-The kernel-level thread is implemented by the operating system.
-The kernel-level thread offers a system call to create and manage the threads from user-space.

Kernel level Threads

Advantages of Kernel-level threads

- The kernel-level thread is fully aware of all threads.
- The scheduler may decide to spend more CPU time in the process of threads being large numerical.
- The kernel-level thread is good for those applications that block the frequency.

Kernel level Threads

Disadvantages of Kernel-level threads

- The kernel thread manages and schedules all threads.
- The implementation of kernel threads is difficult than the user thread.
- The kernel-level thread is slower than user-level threads.

User thread vs. Kernel level Threads

User Threads	Kernel Thread
Multithreading in user process	Multithreading in kernel process
Created without kernel intervention	Kernel itself is multithreaded
Context switch is very fast	Context switch is slow
If one thread is blocked, OS blocks entire process	Individual thread can be blocked
Generic and can run on any OS	Specific to OS
Faster to create and manage	Slower to create and manage

Multithreading

-Multithreading allows the application to divide its task into individual threads.
-In multi-threads, the same process or task can be done by the number of threads, or we can say that there is more than one thread to perform the task in multithreading.
-With the use of multithreading, multitasking can be achieved.

Multithreading

- Many software package that run on modern desktop pcs are multithreaded.
- A web browser might have one thread to display images or text while other thread retrieves data from the network.
- A word-processor may have a thread for displaying graphics, another thread for reading the character 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.

Benefits of Multithreading

Responsiveness:

The multithreaded interactive application continues to run even if part of it is blocked or performing a lengthy operation.

Resource Sharing:

- It allows an application to have several different threads of activity within the same address space.

Benefits of Multithreading

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.

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.

Benefits of Multithreading

Modularity:
Multithreading can facilitate better code organization and modularity by dividing complex tasks into smaller, manageable units of execution.

Multithreading Models

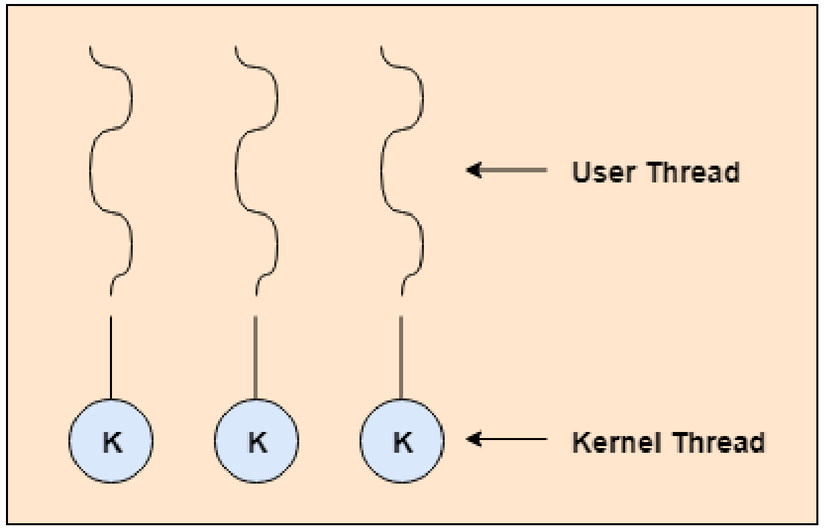
1. Many to many relationship.
2. Many to one relationship.
3. One to one relationship.

One to one Model

-The one to one model creates a separate kernel thread to handle each and every user thread.
-Linux and Windows from 95 to XP implement the one-to-one model for threads.
-As each user thread is connected to different kernel , if any user thread makes a blocking system call, the other user threads won’t be blocked.

One to one Model

A disadvantage of this model is that the creation of a user thread requires a corresponding kernel thread.



One to One Model

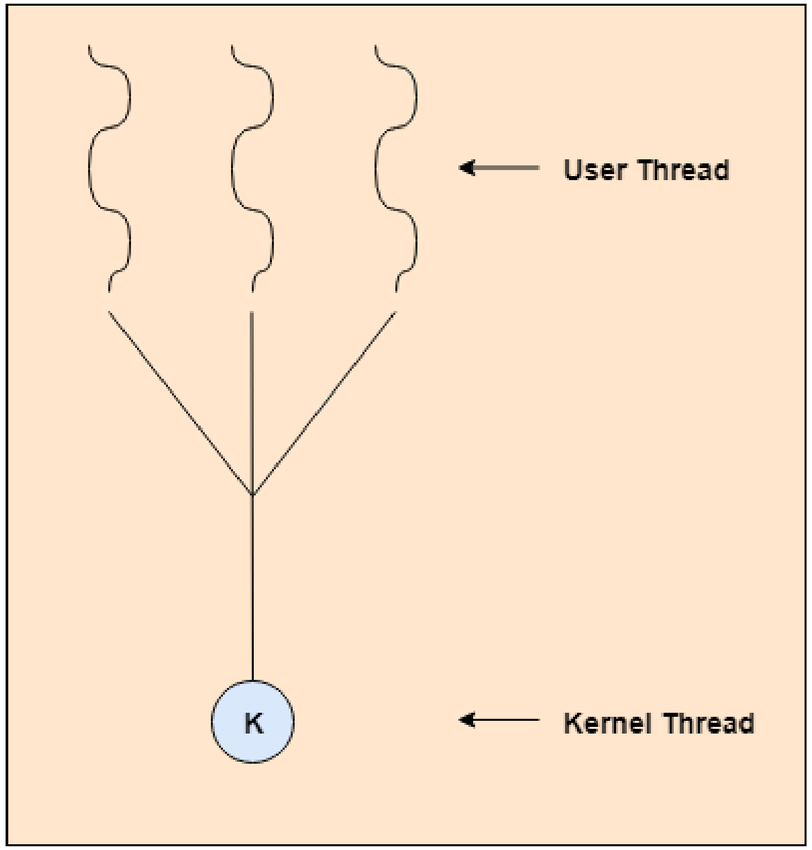
Many to One Model

-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.

Many to One Model

A disadvantage of this model is that a thread blocking system call blocks the entire process. Also, multiple threads cannot run in parallel as only one thread can access the kernel at a time.



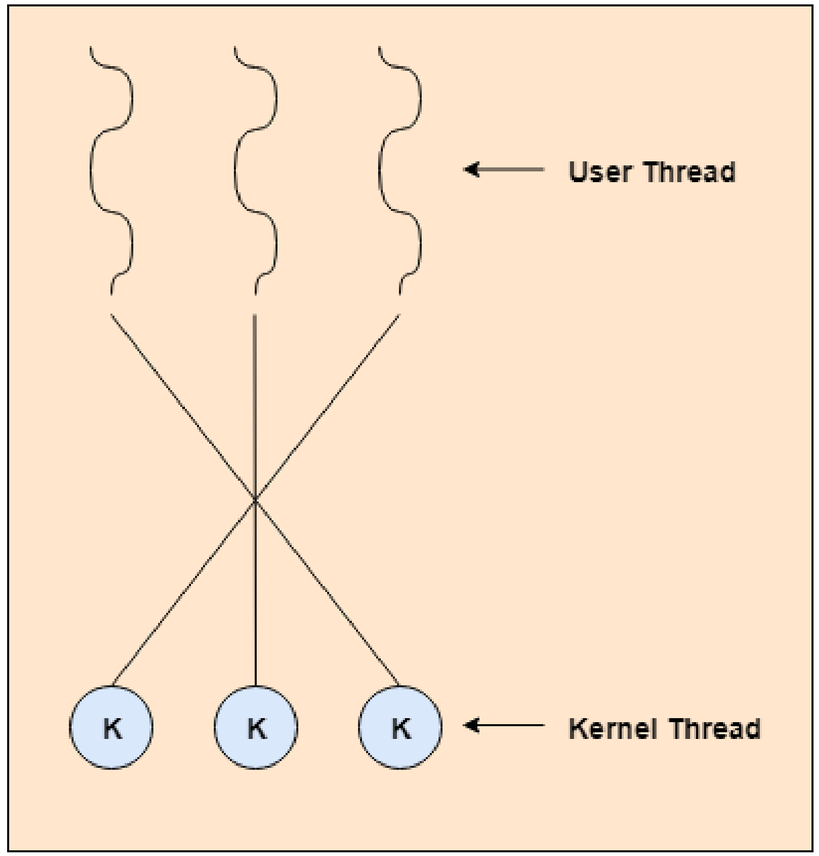
Many to One Model

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.

Many to Many Model

Advantage of this model is if a user thread is blocked we can schedule others user thread to other kernel thread. Thus, System doesn’t block if a particular thread is blocked.



Many to Many Model

Hyperthreading or simultaneous multithreading

Hyperthreaded system allow their processor cores resources to become multiple logical processors for performance.

It enables processors to execute two threads at a time.

To check if your system supports hyperthreading:

CMD>wmic

(window management Instrumentation)

>CPU Get NumberOfCores,NumberOfLogicalProcessors

Example Thread creation in C

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
void *print_message_function( void *ptr );  //Function to be executed by the threads
main()
{
      pthread_t thread1, thread2; // Thread identifiers
      char *message1 = " thread1 ";  //message for thread 1
      char *message2 = " thread2";
      int a, b;                                   //variables to store created thread
// Create two threads, each running the print_message_function
a = pthread_create( &thread1, NULL, print_message_function, (void*) message1);
b = pthread_create( &thread2, NULL, print_message_function, (void*) message2);
```

Example Thread creation inC

```
// Wait for both threads to complete
pthread_join( thread1, NULL);
pthread_join(thread2, NULL);
// Print the return status of each thread
      printf(" thread1 returns: %d\n",a);
      printf(" thread2 returns: %d\n",b);
      exit(0);
}
void *print_message_function( void *ptr ) //// Function to be executed by threads
{
      char *message;
      message = (char *) ptr;
      printf("%s \n", message);
}
```

Output:

Thread1

Thread 2

Thread 1 returns: 0

Thread 2 returns: 0

Inter-Process Communication

- IPC is a mechanism that allows the exchange of data between processes.
- Processes frequently needs 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.
- 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

Process are divided into two categories

1. Independent process:

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

2. 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.

Why do we provide process cooperation?

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.

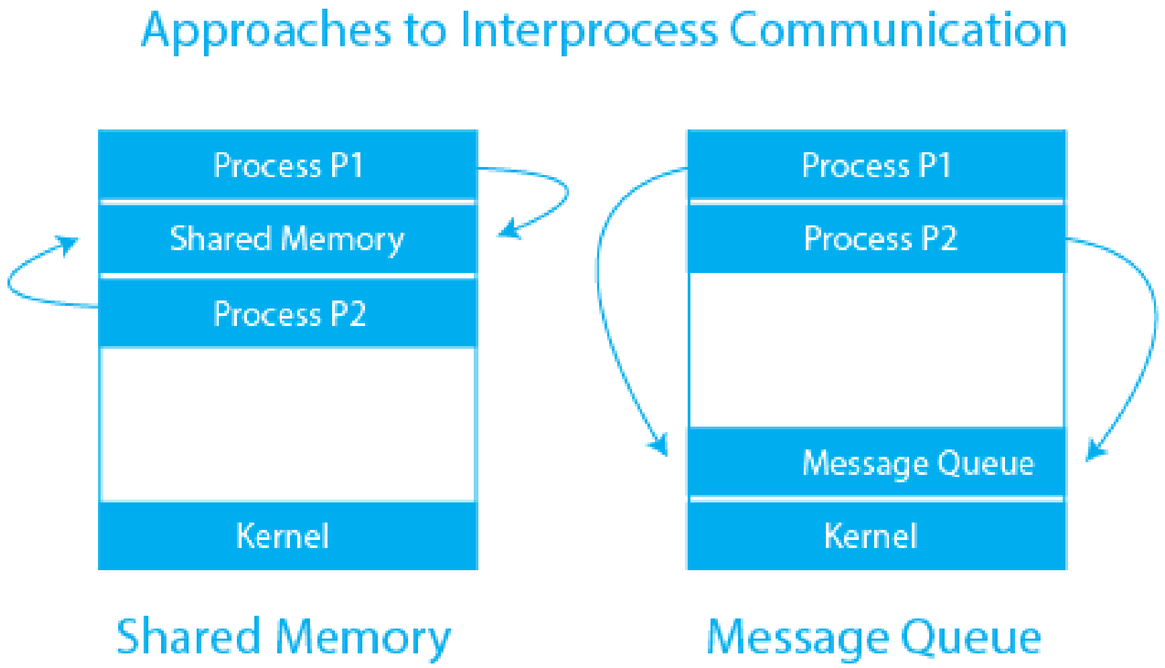
4.convenience:

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.

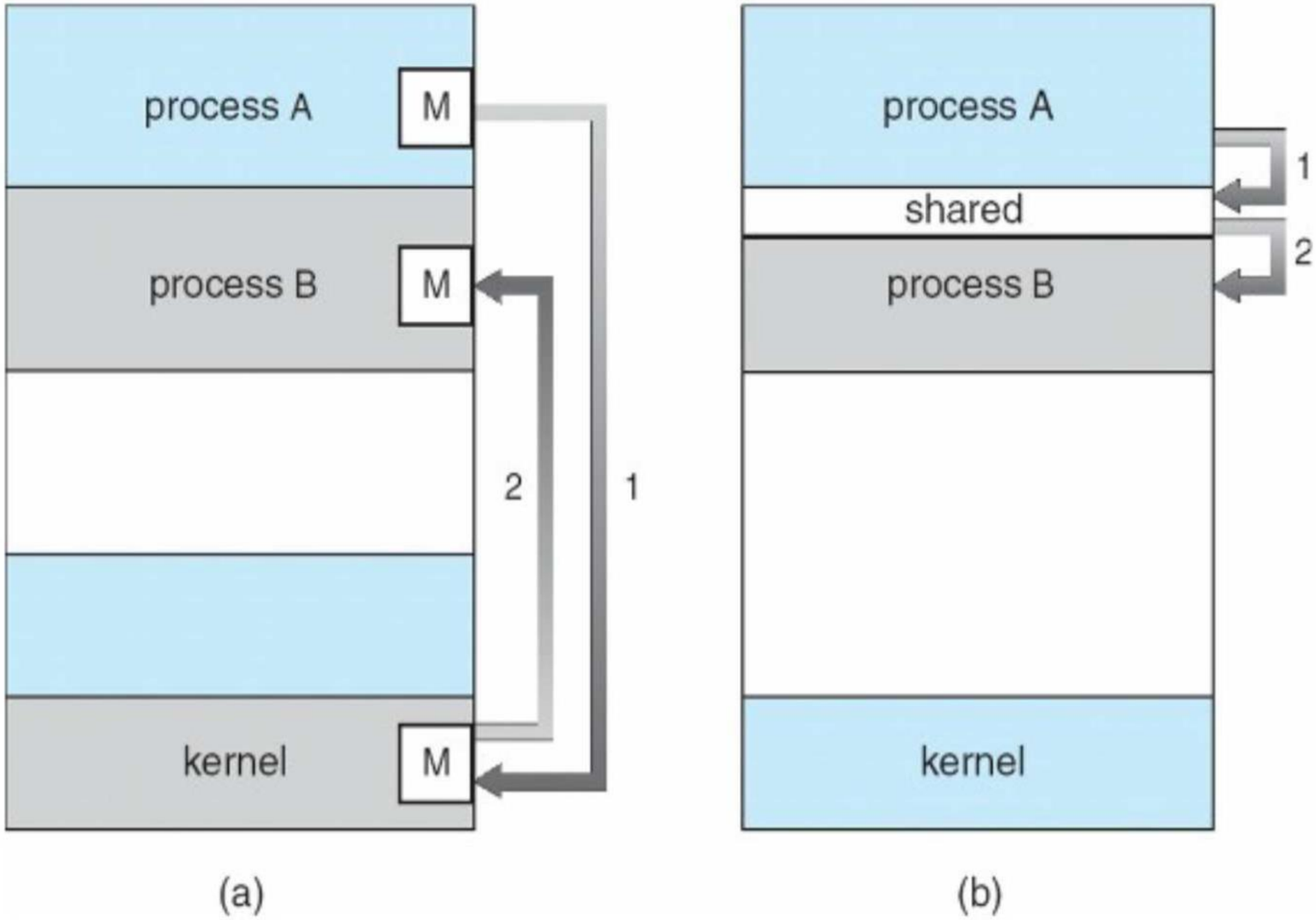
Ways of IPC

The communication between processes can be seen as a method of co-operation between them. Processes can communicate with each other in two ways.

1. Shared Memory
2. Message Passing



Ways of IPC


process A
M
process B
M
2
1
kernel
M
(a)
process A
1
shared
2
process B
kernel
(b)

Ways of IPC

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.

Ways of IPC

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.

Synchronization of concurrent processes

If multiple processes run in our system there would be two modes.

1. Serial mode:

-executes one after another (e.g. ATM)

-one process does not affect another

- Non-preemptive approach

2. Parallel mode

-Multiple process runs at a time simultaneously

-One process may affect another

-Preemptive approach

Processes Types

The process is divided into two categories

1. Independent process:

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

2. Co-operating Process:

- A process is cooperating if it may affect others or be affected by the other process.
- They share something (Memory, Resources, Variables, Code) with other processes.
- They have something in common

Process synchronization

Example:

Initially Shared = 5

Process 1	Process 2
int X = shared	int Y = shared
X++	Y--
sleep(1)	sleep(1)
shared = X	shared = Y

We are assuming the final value of a shared after execution of P1 and P2 is 5 (as P1 increment by 1 and P2 decrement by 1). But we are getting undesired value 4.

This is called RACE condition

Example how one process can affect another

P1 and P2 share a common variable (shared=5),

Suppose, Process P1 first executes:

Shared = 5

X=5

increment it by 1(X=6),

sleep(1), it switches from P1 to P2 and P1 goes waiting for 1 sec.

Now CPU executes P2:

Shared = 5

Y=5

decrement Y by 1(Y=4),

sleep(1), the P2 goes in waiting.

after 1 second CPU takes the P1 and executes the remaining:

shared=6 (shared value is updated), process 1 terminated.

after 1 second CPU starts executing the remaining line of P2

(store the local variable (Y=4) in shared.

Shared = 4 (shard value updated). Process terminated.

Process synchronization

-Process Synchronization is the coordination of execution of multiple processes in a multi-process system to ensure that they access shared resources.

-It is crucial for managing multiple concurrent processes or threads effectively.

-It aims to resolve conflict, problem of race conditions and other synchronization issues in a concurrent system.

-The primary goal of Process Synchronization is to maintain data integrity, manage shared resources, and prevent concurrency-related issues like data corruption, deadlock, and contention.

'resource contention’: conflict over a shared resource between several components.

Process synchronization

Process synchronization is crucial in scenarios where:

Shared Resources: Multiple processes need to access and modify shared data, such as files, memory locations, or hardware devices.

Concurrency: Processes run concurrently, and their execution order cannot be predetermined.

Critical Sections: Critical sections are parts of code where data integrity must be maintained. Only one process should access a critical section at a time.

Preventing Race Conditions: Race conditions occur when multiple processes attempt to modify shared data simultaneously.

Synchronization Mechanism

Several synchronization mechanisms are commonly used in OS:

1. Mutex (Mutual Exclusion):

- Mutexes are binary semaphores that allow only one process to access a shared resource at a time.
- They are often used to protect critical sections of code.

2. Semaphore:

- Semaphores are more general synchronization primitives that can be used to control access to a resource by multiple processes.

Synchronization Mechanism

3. Condition Variables:

- Condition variables are used to control the flow of execution in a multi-threaded program.
- They allow threads to wait for a specific condition to be met before proceeding.

4. Spinlock:

- Spinlocks are used in multi-core systems, where a thread repeatedly checks the lock until it becomes available.

Challenges in Process Synchronization

Synchronization is critical, it comes with its set of challenges:

Deadlocks: Inefficient synchronization can lead to deadlocks, where processes are stuck, waiting for resources indefinitely.

Starvation: Poorly designed synchronization mechanisms can result in. some processes getting more access to resources and others are waiting

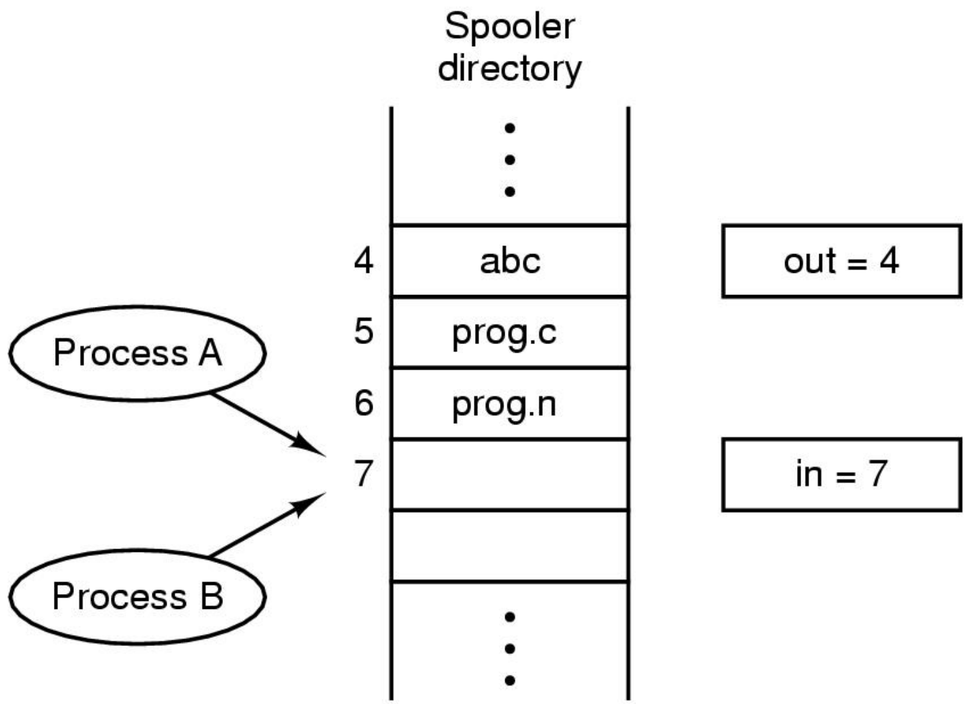
Performance Overhead: Excessive use of synchronization can introduce performance overhead, as processes spend time waiting for locks and resources.

Complexity: Implementing synchronization mechanisms correctly can be challenging, especially in large-scale systems.

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.

The final results depend on who runs precisely (accurately).



Two processes want to access shared memory at same time

Race condition

Q.1: What is a race condition?

A race condition occurs when multiple processes or threads access shared data concurrently, and the final outcome of the program depends on the relative timing of their execution. This can lead to unexpected and undesired results, such as data corruption, incorrect calculations, or program crashes.

Critical section problem

What is critical section?

-The critical section is a part of the program or code segment where the shared variables can be accessed by various processes.

-Only one process can execute in its critical section at a time.

-All the other processes have to wait to execute in their critical sections.

Scenario:

Concurrent process: Multiple processes are running

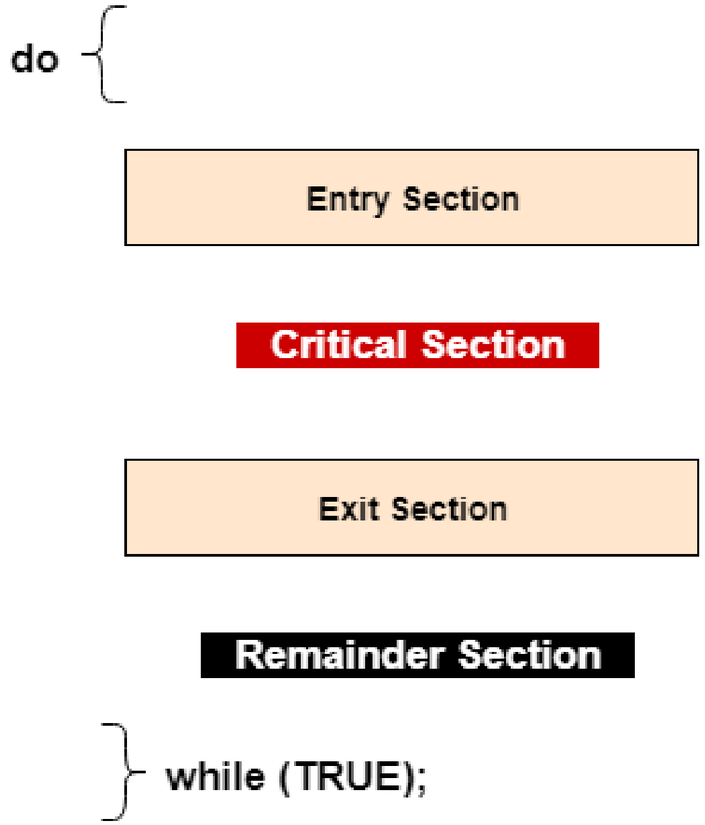
Cooperative process: which shares something.

Critical section problem

There are four section in program:

1. Entry section: The entry section handles the entry into the critical section. That is to execute the code written in the critical section, it is necessary to execute the code in the entry section. Entry section is door to critical section.

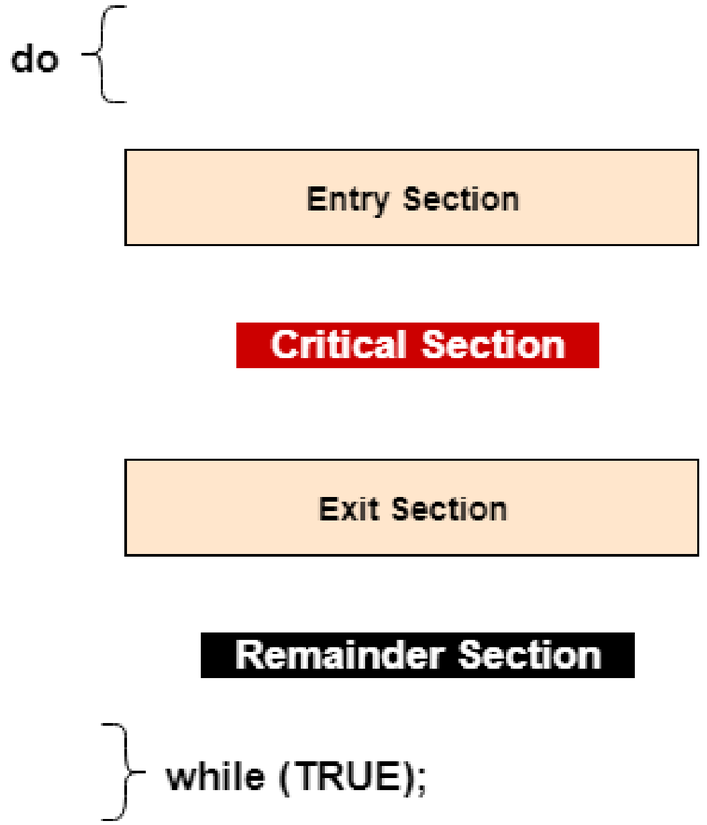
2. Critical Section: This part allows one process to enter and modify the shared variable. (common section)



Critical section problem

3. Exit Section: Exit section allows the other process that are waiting in the Entry Section, to enter into the Critical Sections. It also checks that a process that finished its execution should be removed through this Section.

4. Remainder Section: All other parts of the Code, which is not in Critical, Entry, and Exit Section, are known as the Remainder Section.



Critical section problem

When two process tries to execute code written in critical section, then problem may occurs (race condition).

The critical section problem is used to design a set of protocols which can ensure that the Race condition among the processes will never arise.

Synchronization mechanism

There are four requirements or rules or conditions for any solution to the critical section problem or to Achieve synchronization.

Primary rules: mandatory to follow

1. Mutual Exclusion: only one process can be inside the critical section at any time.

2. Progress: Any process wants to access the critical section but another process protects it even if the critical section is free. In the progress, there should not be such a case. Progress means that if one process doesn't need to execute into critical section then it should not stop other processes to get into the critical section.

Secondary rules: Optional to follow

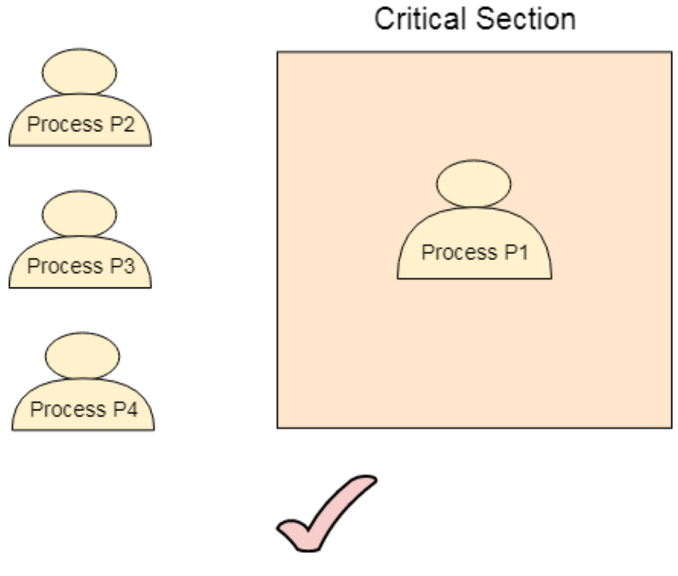
3. Bounded Waiting: Each process must have a limited waiting time. It should not wait endlessly to access the critical section.

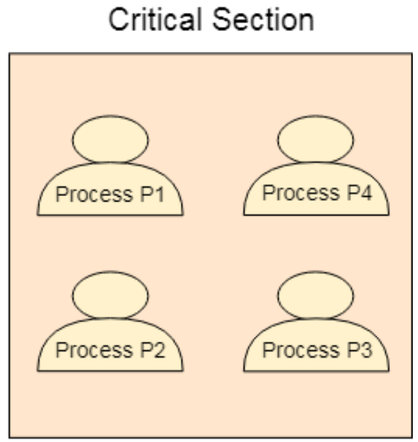
4. No assumption related to hardware, or speed: It must be portable or compatible with hardware and software.

Critical section and solution: Mutual Exclusion

-Mutual Exclusion is a property of process synchronization that states that “**no two processes can exist in the critical section at any given point of time**”. only one process can be inside the critical section at any time.

-If any other processes require the critical section, they must wait until it is free.





Mutual Exclusion: use cases

Use cases of Mutual Exclusion in Synchronization:

- Printer Spooling
- Bank Account Transactions
- Traffic Signal Control

Ways to manage race condition

Mutual Exclusion with Busy Waiting

- Disabling Interrupts
- Lock Variables
- The Test and set lock (TSL) Instruction
- Strict Alternation
- Peterson’s Solution

Mutual Exclusion without Busy Waiting

- Sleep and wakeup
- Semaphore
- Message Passing
- Monitor

Busy waiting: It is process synchronization technique where the process waits and continuously keeps on checking for the condition to be satisfied before going ahead with its execution.

-Busy Waiting is also known as busy looping or spinning.

Disabling Interrupts

-An interrupt is a signal emitted by hardware or software when a process or an event needs immediate attention.
-It is hardware mechanism provided by OS
-OS should disable interrupt so that another process can enter to the critical section.
-The simplest solution is to have each process disable all interrupts just after entering its critical region and re-enable them just before leaving it.

<interrupt on>
Critical section
<interrupt off>

Disabling Interrupts

Limitation:

-This approach is generally unattractive because it is unwise to give user processes the power to turn off interrupts.

-Suppose that one of them did it and never turned them on or off again? That could be the end of the system.

Lock variable

A lock variable provides the simplest synchronization mechanism for processes.

Some important points about lock variables:

1. Its a software mechanism implemented in user mode, i.e. no support required from the Operating System.
2. Its a busy waiting solution (keeps the CPU busy even when its technically waiting).
3. It can be used for more than two processes.

Lock variable working

Lock = 0 implies critical section is vacant
Lock = 1 implies critical section occupied.

```
do{
    acquire lock
    critical section
     release lock
}
```

Pseudo code:
At starting lock = 0 or false

```
while(lock == 1);   // Entry section
 Lock = 1;
//critical section
Lock = 0;              // Exit section
```

Case1: if no preemption, Mutual Exclusion is possible
Case 2: if preemption, Mutual Exclusion is not possible

Lock variable

Case 2: if preemption, Mutual Exclusion is not possible

Explanation:

Let us consider that we have two processes P1 and P2. The process P1 wants to execute its critical section. P1 gets into the entry section. Since the value of lock is 0 hence P1 changes its value from 0 to 1 and enters into the critical section.

Meanwhile, P1 is preempted by the CPU and P2 gets scheduled. Now there is no other process in the critical section and the value of lock variable is 0. P2 also wants to execute its critical section. It enters into the critical section by setting the lock variable to 1.

Now, CPU changes P1's state from waiting to running. P1 is yet to finish its critical section. P1 has already checked the value of lock variable and remembers that its value was 0 when it previously checked it. Hence, it also enters into the critical section without checking the updated value of lock variable.

Now, we got two processes in the critical section.

Lock variable

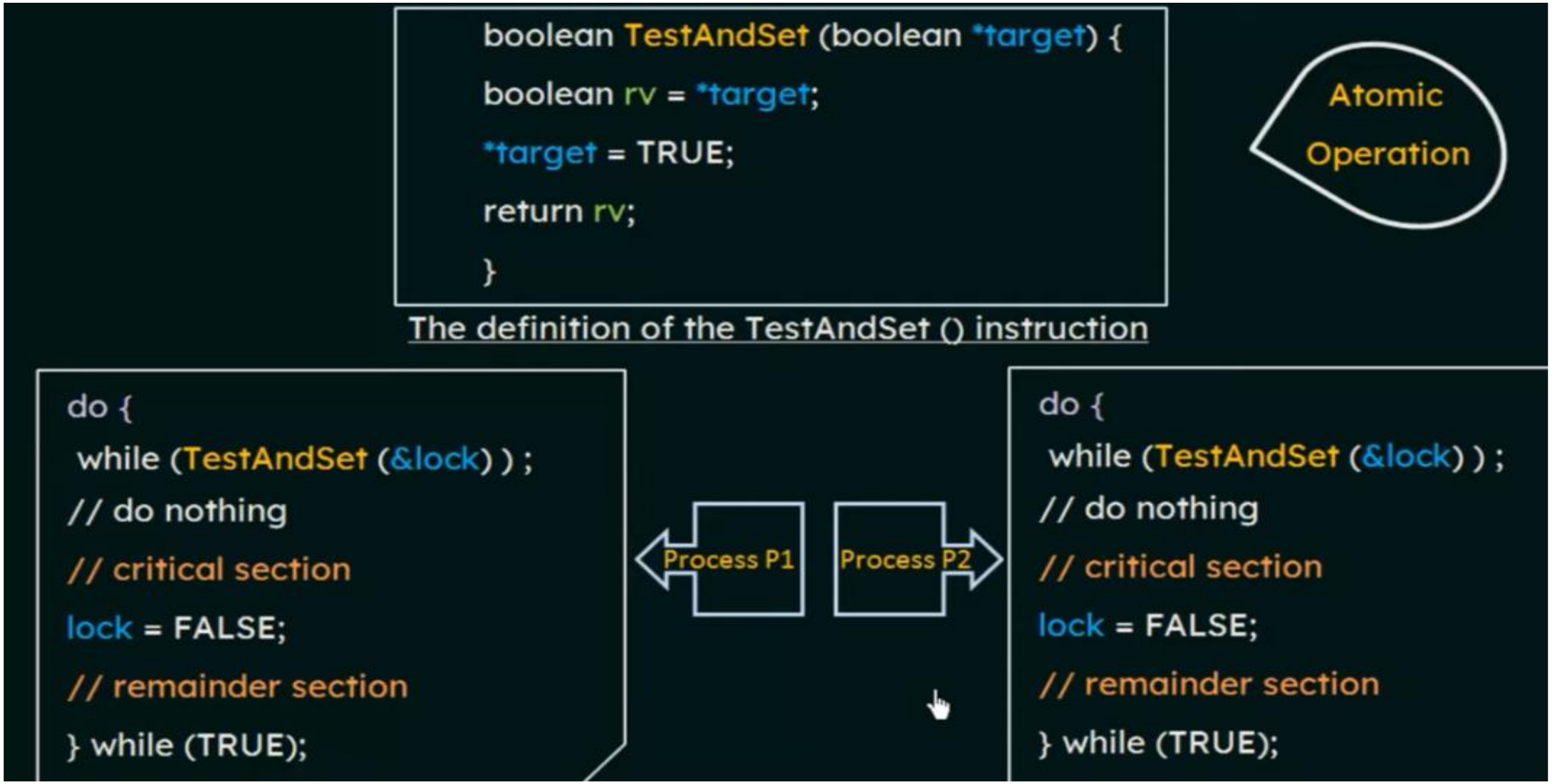
- In the lock variable approach, the process is able to enter into the critical section only when the lock variable is set to 1.
- In the lock variable approach, more than one process has a lock variable value of 1 at the same time.
- Due to such conditions, the lock variable is not able to guarantee mutual execution.

To overcome this mutual exclusion problem, there is another method called Test and Set Lock

Test and Set Lock (TSL) method

-It is an atomic (combined) method, which means all the operations in the process (i.e. test lock, check vacant, set lock 1) can execute in a single operation, which can not be interrupted by another process.
-It is a hardware solution to the synchronization problem
-there is a shared lock variable which can have a value of either 0 (vacant) or 1 (occupied).
-Before entering into the critical section, the process enquires about a lock.
-if it is locked, it waits until it becomes vacant
-if not locked, it sets the lock and enters into the Critical section.

Test and Set Lock (TSL) method



Test and Set Lock (TSL) method

-It satisfied mutual exclusion
-It satisfied progress
-It does not satisfy bounded-waiting because, there is no queue maintained, so any new process that finds the lock to be false again can enter. So bounded waiting is not ensured.

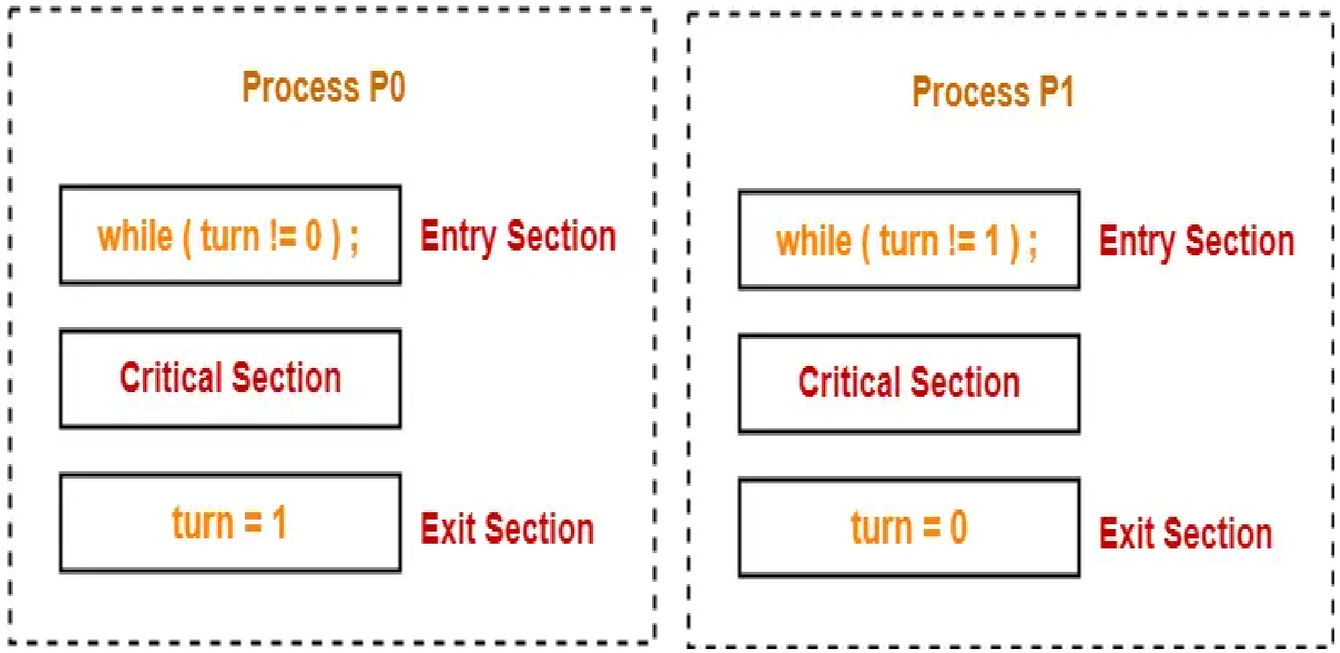
Turn variable: Strict Alternation method

-It is two process solution, it can’t work in more than two process
-It works in user mode.
-It always ensures mutual exclusion
-Progress may not be possible, because one process can protect another.
-Bounded waiting: Each process gets the chance, once a previous process is executed the next process gets the chance therefore turn variable ensures bounded waiting.

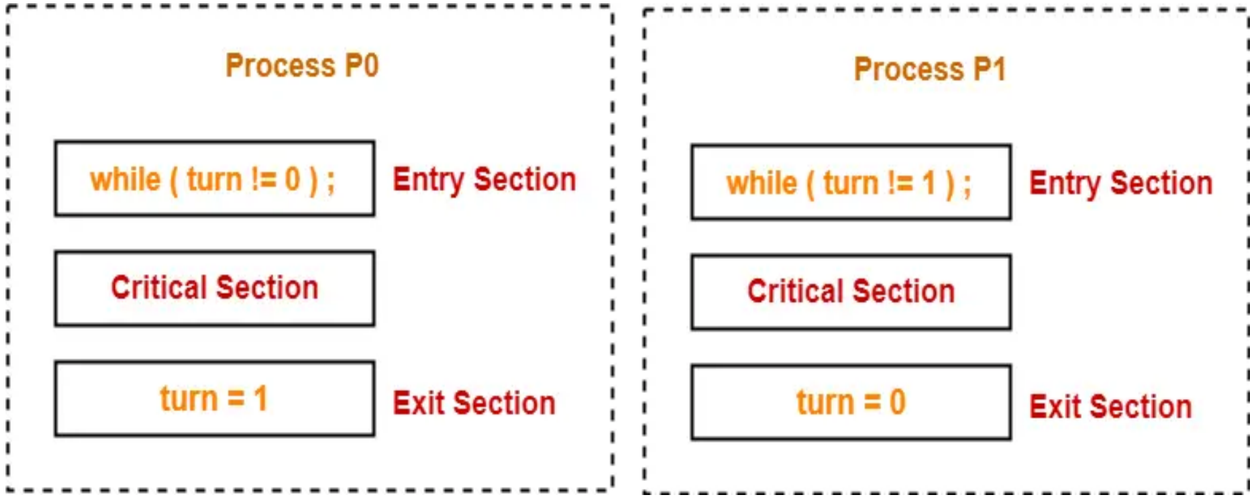
Turn variable: Strict Alternation method

Initially, turn value is set to 0.

Turn value = 0 means it is the turn of process P0 to enter the critical section.
Turn value = 1 means it is the turn of process P1 to enter the critical section.



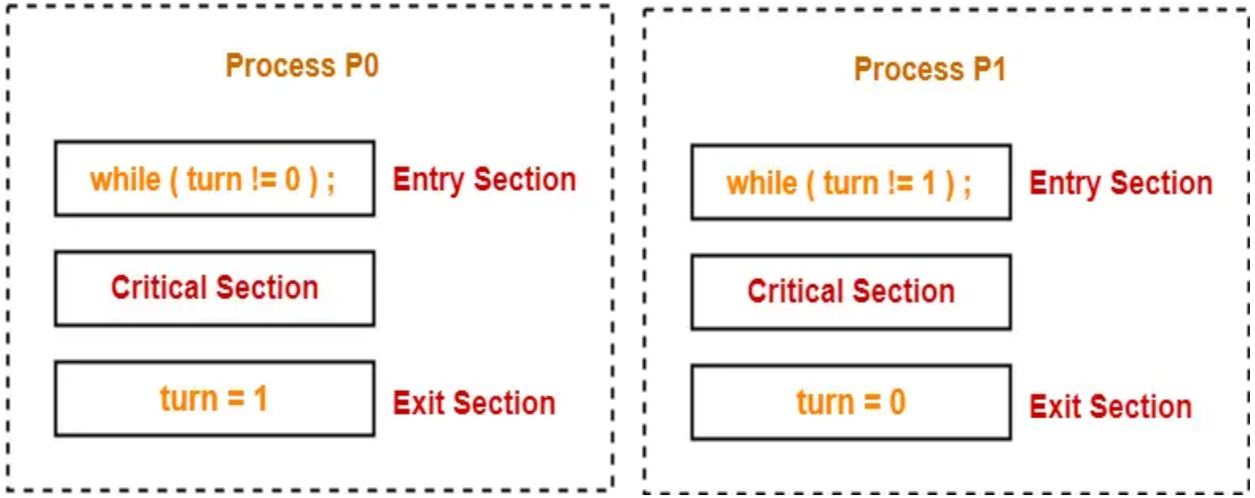
Turn variable: Strict Alternation method



Case 1: if turn value is 0

1. if Process P0 arrives.
2. It executes the turn!=0 instruction.
3. Since turn value is set to 0, so it returns value 0 to the while loop.
4. The while loop condition breaks.
5. Process P0 enters the critical section and executes.
6. Now, even if process P0 gets preempted in the middle, process P1 can not enter the critical section.
7. Process P1 can not enter unless process P0 completes and sets the turn value to 1.

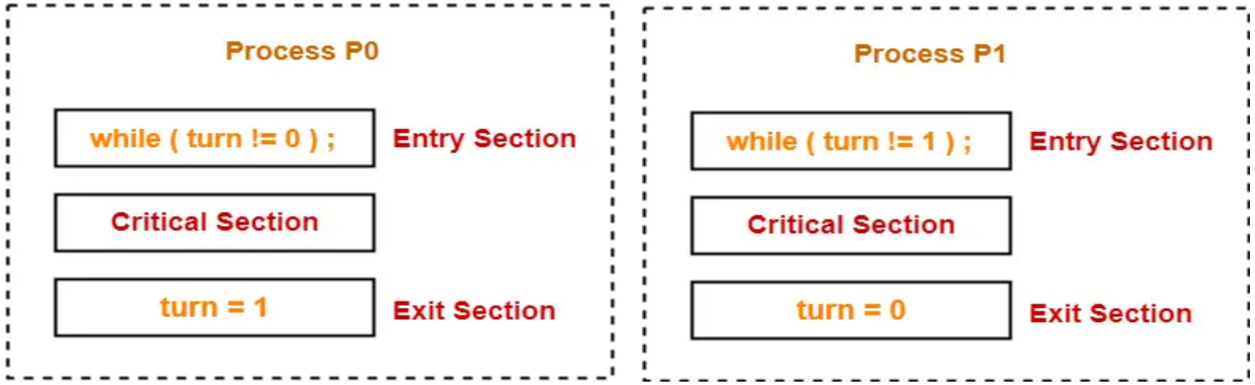
Turn variable: Strict Alternation method



Case 2: if turn value is 0

1. if Process P1 arrives.
2. It executes the turn!=1 instruction.
3. Since turn value is set to 0, so it returns value 1 to the while loop.
4. The returned value 1 does not break the while loop condition.
5. The process P1 is trapped inside an infinite while loop.
6. The while loop keeps the process P1 busy until the turn value becomes 1 and its condition breaks.

Turn variable: Strict Alternation method



Case 3:

1. Process P0 comes out of the critical section and sets the turn value to 1.
2. The while loop condition of process P1 breaks.
3. Now, the process P1 waiting for the critical section enters the critical section and execute.
4. Now, even if process P1 gets preempted in the middle, process P0 can not enter the critical section.
5. Process P0 can not enter unless process P1 completes and sets the turn value to 0.

Peterson's Solution

- This is a software based solution to Critical Section Problem.
- Doesn't work on modern architectures.
- It's for only 2 processes which alternate execution between then critical section and remainder section. Say, P1 is the first process and P2 is the second process.
- It is a humble algorithm

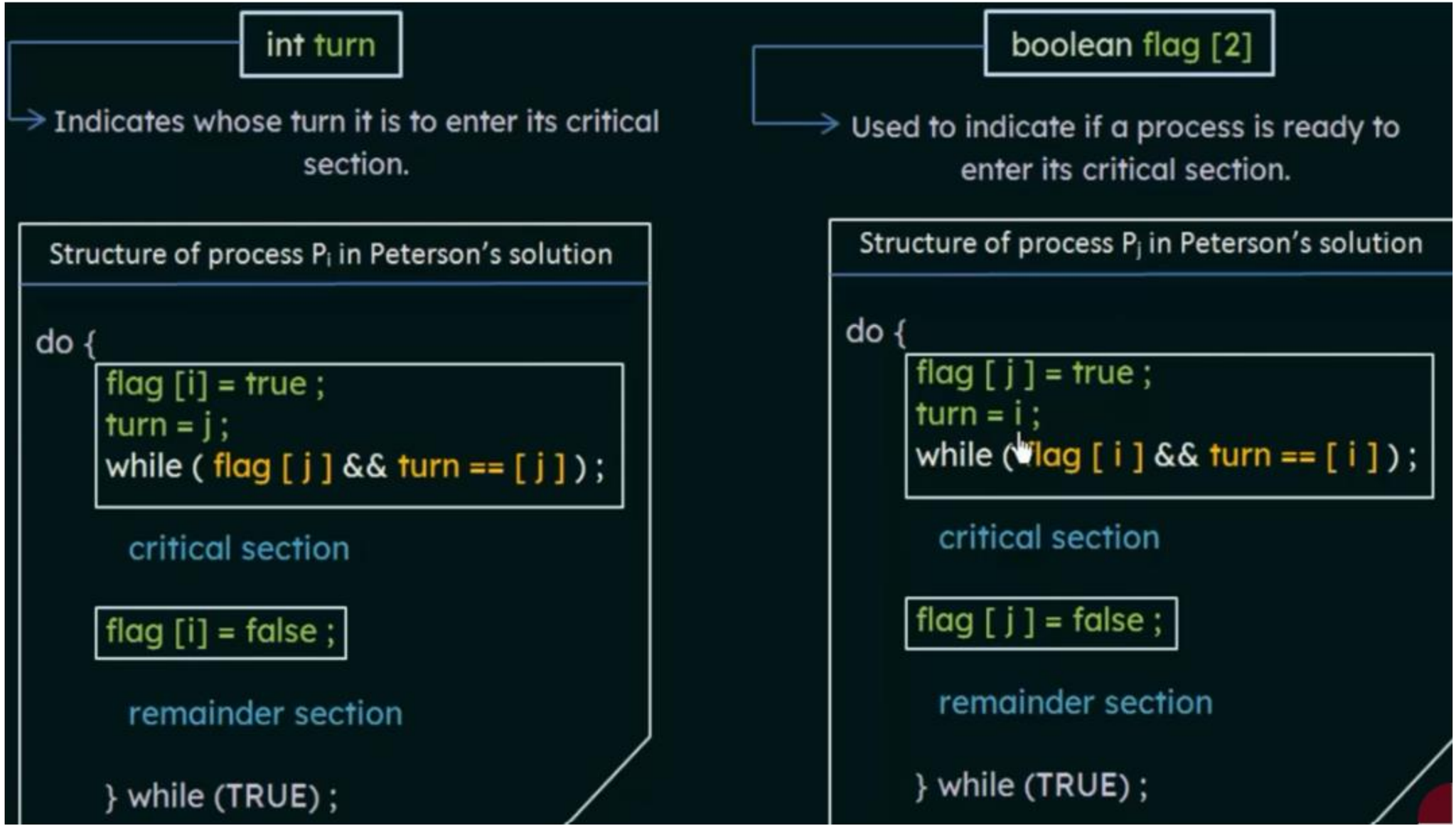
Peterson's Solution

Requires Two data items to be shared between the process:

1. **int turn** : indicates whose turns to enter into critical section.
2. **Boolean flag [2] :** used to indicate if process is ready to enter into critical section

Let flag[i] indicate process Pi.
-If flag[i] = true , then Process Pi is ready to execute in its critical section.
Let flag[j] indicates process Pj.
-If flag[j] = true, then Process Pj is ready to execute in its critical section.

Peterson's Solution



Peterson's Solution

Mutual Exclusion (satisfied): The method provides mutual exclusion for sure. In entry section, the while condition involves the criteria for two variables therefore a process cannot enter in the critical section until the other process is interested and the process is the last one to update turn variable.

Progress (satisfied): An uninterested process will never stop the other interested process from entering in the critical section. If the other process is also interested then the process will wait.

Bounded waiting (satisfied): The interested variable mechanism failed because it was not providing bounded waiting.

Mutual Exclusion without Busy Waiting

- Sleep and wakeup (producer-consumer/ Bounded Buffer)
- Semaphore
- Message Passing
- Monitor

Sleep and wakeup

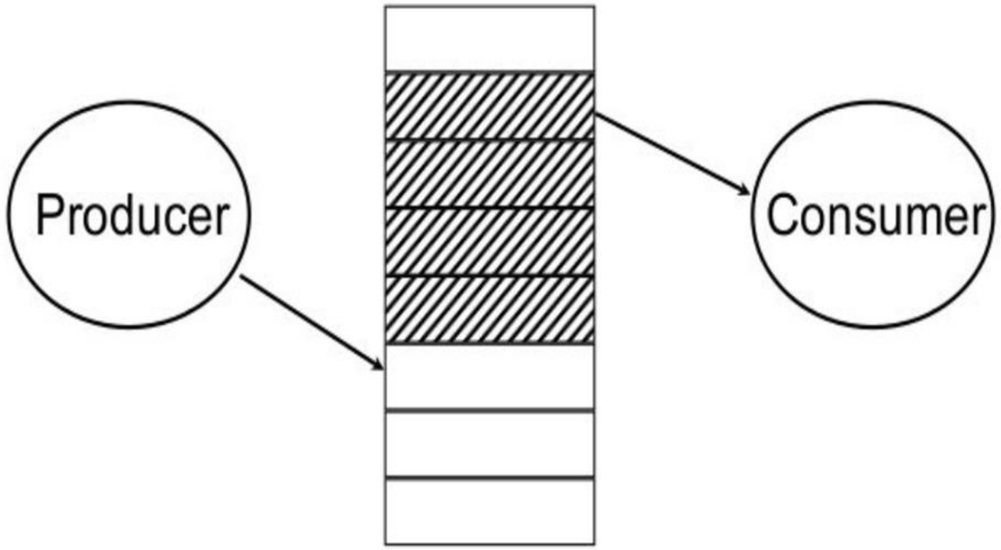
- Also called producer-consumer problem or bounded buffer problem.
- The Producer-Consumer problem is a classical multi-process synchronization problem, that is we are trying to achieve synchronization between more than one process.
- Sleep and wakeup are system calls that blocks process instead of wasting 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.

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



Examples to use Sleep and Wakeup primitives:

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.

Examples to use Sleep and Wakeup primitives:

```
#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 */
}}
```

Examples to use Sleep and Wakeup primitives:

```
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

Examples to use Sleep and Wakeup primitives:

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.**

Examples to use Sleep and Wakeup primitives:

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.

Examples to use Sleep and Wakeup primitives:

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.

Semaphore

Semaphores refer to the integer variables that are primarily used to solve the critical section problem via combining two atomic procedures, wait and signal, for the process synchronization.

They are used to enforce mutual exclusion, avoid race conditions, and implement synchronization between processes.

The process of using Semaphores provides two operations:
wait (p) and signal(v):

Operation on Semaphore

There are various operations in entry code and exit code:

1. P(), wait, down: it is code for the entry section. This operation tests if there is any process in the critical section. If it is not there, it will decrease and value and enter into a critical section. If any process is in a critical section while the condition will be true a

Pseudo code:

```
P(semaphore S)
{
while (S<=0);
S--;
}
```

Elaboration of pseudocode to test: consider S = 2

```
P(semaphore S) {
S.value = s.value-1;
If(S.value<0){
Put process in PCB suspended list, sleep();
else{
return;  //insert into critical section.
} }
```

Semaphore

2. V() or Signal or up or release or post :

-This operation increments the actual value of its argument V.

-Signal is used to notify other process waiting to enter into the critical section by incrementing it’s value.

-It is code for the exit section.

Pseudo code:

```
V(Semaphore S)
{
S++;
}
```

Elaboration of pseudocode to test:

```
P(semaphore S) {
S.value = s.value+1;
If(S.value<=0){
select process from suspended list, and wakeup();
} }
```

Critical questions

1. How many processes are there inside the critical section when the semaphore value is 0?

Answer: 0

2. How many processes are there inside the critical section when the semaphore value is 10?

Answer: 10

3. How many processes are there inside the block section when the semaphore value is -3?

Answer: 3

Critical questions

4. The current value of the semaphore is 10, if you perform 6P operation and 4V operation, what will be the final value of the semaphore?

Answer: 10-6 = 4

4+4 = 8 (final Semaphore value is 8).

5. The current value of the semaphore is 17, if you perform 6P operation, 3V operation, 1P operation. what will be the final value of the semaphore?

Answer: 17-6 = 1

11+3 = 14

14-1 = 13 (final Semaphore value is 13).

Types of semaphore

The two common kinds of semaphores:

1. Binary semaphores:

-It is also known as Mutex lock.

-It can only have two possible values: 0 and 1, and its value is set to 1 by default.

-if the semaphore value is 0: the critical section is busy

-if the semaphore value is 1: the critical section is free

Example: check the following P and V codes with 0 and 1:

```
P(semaphore S) {
while (S<=0);
S--;
}
```

```
V(Semaphore S) {
S++;
}
```

Types of semaphore

2. Counting semaphores: used to count resources and its value varies from (–infinity to +infinity).

Solving a producer-consumer problem using semaphore

Problems:

-Producer should not insert the data into the full buffer

-Consumer should not remove data from an empty buffer

-Producer and consumer should not insert and remove data simultaneously.

We can solve the above problem using semaphore.

Solving a producer-consumer problem using semaphore

We use three semaphore to solve above problem:

1. m(Mutex): a binary semaphore which is used to acquire and release the lock.

2. empty: a counting semaphore whose initial value is the number of sluts in buffer, since initially all slots are empty.

3. full: It keeps the track of how many slots are filled in the buffer. For example, if the buffer size is 10 and 5 blocks are filled, then full value is 5. a counting semaphore, whose initial value is 0 because we consider initially buffer is empty so full value is 0.

Solving a producer-consumer problem using semaphore

Producer

```
do {
  wait (empty); // wait until empty>0
                and then decrement 'empty'
  wait (mutex); // acquire lock
  /* add data to buffer */
  signal (mutex); // release lock
  signal (full);  // increment 'full'
} while(TRUE)
```

Consumer

```
do {
  wait (full); // wait until full>0 and
                then decrement 'full'
  wait (mutex); // acquire lock
  /* remove data from buffer */
  signal (mutex); // release lock
  signal (empty);  // increment 'empty'
} while(TRUE)
```

Advantages of Semaphores

- A simple and effective mechanism for process synchronization
- Supports coordination between multiple processes
- Provides a flexible and robust way to manage shared resources.
- It can be used to implement critical sections in a program.
- It can be used to avoid race conditions.

Disadvantages of Semaphores

- It Can lead to performance degradation due to overhead associated with wait and signal operations.
- Can result in deadlock if used incorrectly.
- It can cause performance issues in a program if not used properly.
- It can be difficult to debug and maintain.
- It can be prone to race conditions and other synchronization problems if not used correctly.
- It can be vulnerable to certain types of attacks, such as denial of service attacks.

The Dining philosophers problem

-The dining philosopher's problem is the classical problem of synchronization which says that Five philosophers are sitting around a circular table and their job is to think and eat alternatively.

-To eat a philosopher needs both their right and left chopstick.

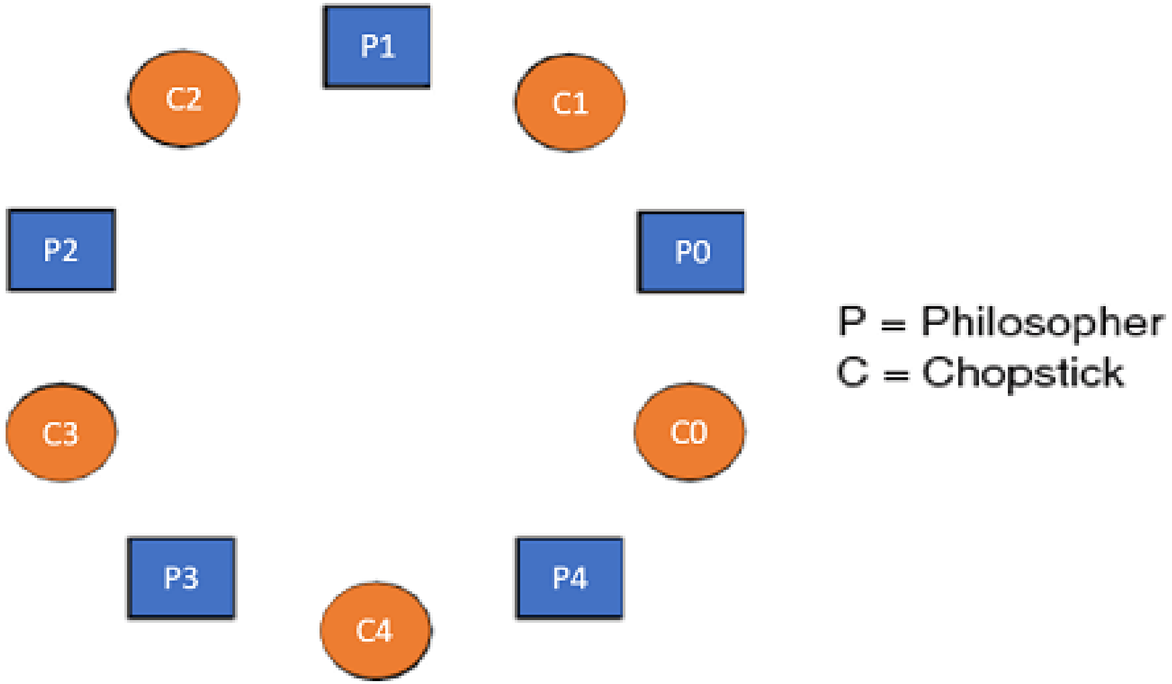
-A philosopher can only eat if both the immediate left and right chopsticks of the philosopher is available.

The Dining philosophers problem

-In case if both immediate left and right chopsticks of the philosopher are not available then the philosopher puts down their (either left or right) chopstick and starts thinking again.

The Dining philosophers problem

The five Philosophers are represented as P0, P1, P2, P3, and P4 and five chopsticks by C0, C1, C2, C3, and C4.



The Dining philosophers problem

```
Void Philosopher (void)
 {
 while(true)
  {
   thinking();
   take_chopstick[i];  //take left chopistick first
   take_chopstick[ (i+1) % 5] ;   //take right chopstick
    .
   . EATING THE NOODLE
   put_chopstick[i] );          //after eating put down left chopstick first
   put_chopstick[ (i+1) % 5] ;
   .
THINKING
 } }
```

The Dining philosophers problem

Suppose Philosopher P0 wants to eat, it will enter in Philosopher() function, and execute take_chopstick[i]; by doing this it holds C0 chopstick after that it execute take_chopstick[(i+1) % 5]; by doing this it holds C1 chopstick(since i =0, therefore (0 + 1) % 5 = 1)

Similarly suppose now Philosopher P1 wants to eat, it will enter in Philosopher() function, and execute take_chopstick[i]; by doing this it holds C1 chopstick after that it execute take_chopstick[(i+1) % 5]; by doing this it holds C2 chopstick(since i =1, therefore (1 + 1) % 5 = 2)

But Practically Chopstick C1 is not available as it has already been taken by philosopher P0, hence the above code generates problems and produces race condition.

The Dining philosophers problem using semaphore

Let's modify the above code of the Dining Philosopher Problem by using semaphore operations wait and signal to solve race condition.

```
void Philosopher
 {
 while(1)
  {
   Wait( take_chopstickC[i] );
   Wait( take_chopstickC[(i+1) % 5] ) ;
   . .
   . EATING THE NOODLE
   .
   Signal( put_chopstickC[i] );
   Signal( put_chopstickC[ (i+1) % 5] ) ;
   .
   . THINKING
  } }
```

The Dining philosophers problem using semaphore

Explanation of the above code:

Let value of i = 0, Suppose P0 wants to eat, it will enter in Philosopher() function, and execute Wait(take_chopstickC[i]); by doing this it holds C0 chopstick and reduces semaphore C0 to 0, after that it execute Wait(take_chopstickC[(i+1) % 5]); by doing this it holds C1 chopstick(since i =0, therefore (0 + 1) % 5 = 1) and reduces semaphore C1 to 0

Similarly, if P1 wants to eat, it will enter in Philosopher() function, and execute Wait(take_chopstickC[i]); by doing this it will try to hold C1 chopstick but will not be able to do that, since the value of semaphore C1 has already been set to 0 by philosopher P0, therefore it will enter into an infinite loop because of which philosopher P1 will not be able to pick chopstick C1 whereas if Philosopher P2 wants to eat, it will enter in Philosopher() function, and execute Wait(take_chopstickC[i]); by doing this it holds C2 chopstick and reduces semaphore C2 to 0, after that, it executes Wait(take_chopstickC[(i+1) % 5]); by doing this it holds C3 chopstick(since i =2, therefore (2 + 1) % 5 = 3) and reduces semaphore C3 to 0.

The Dining philosophers problem using semaphore

Conditions:

1. Adjacent philosophers can’t eat at a time.
2. two opposite philosophers can eat at a time.

Deadlock condition:

If all the philosophers pick their left chopstick and preampted, which leads to the condition of deadlock.

The Dining philosophers problem using semaphore

General solution of such deadlock are:

1. Maximum number of philosophers on the table should not be more than four
2. A philosopher at an even position should pick the right chopstick and then the left chopstick while a philosopher at an odd position should pick the left chopstick and then the right chopstick.
3. Only in case if both the chopsticks (left and right) are available at the same time, only then a philosopher should be allowed to pick their chopsticks
4. All the four starting philosophers (P0, P1, P2, and P3) should pick the left chopstick and then the right chopstick, whereas the last philosopher P4 should pick the right chopstick and then the left chopstick.

Monitors

-To overcome the deadlock generated by semaphore, monitor comes into existence.
-Monitors are a higher-level synchronization construct that simplifies process synchronization by providing a high-level abstraction (Abstract data type) for data access and synchronization.

-A monitor data type presents set of programmer-defined operations that provide mutual exclusion within the monitor.

Syntax Monitors

```
Monitor monitorName{

//shared variables declaration;

procedure p1{ ... };
procedure p2{ ... };
    ...
    procedure pn{ ... };
    {
        initializing_code;
    }
}
```

1. procedures are operations that can be performed on shared variables. We can have multiple procedures.
2. Local variables of monitor can access by only procedure defined within monitor.
3. Only one process can be active at a time within the monitor.

Syntax Monitors

Additional considerations to achieve synchronize:

Condition construct:

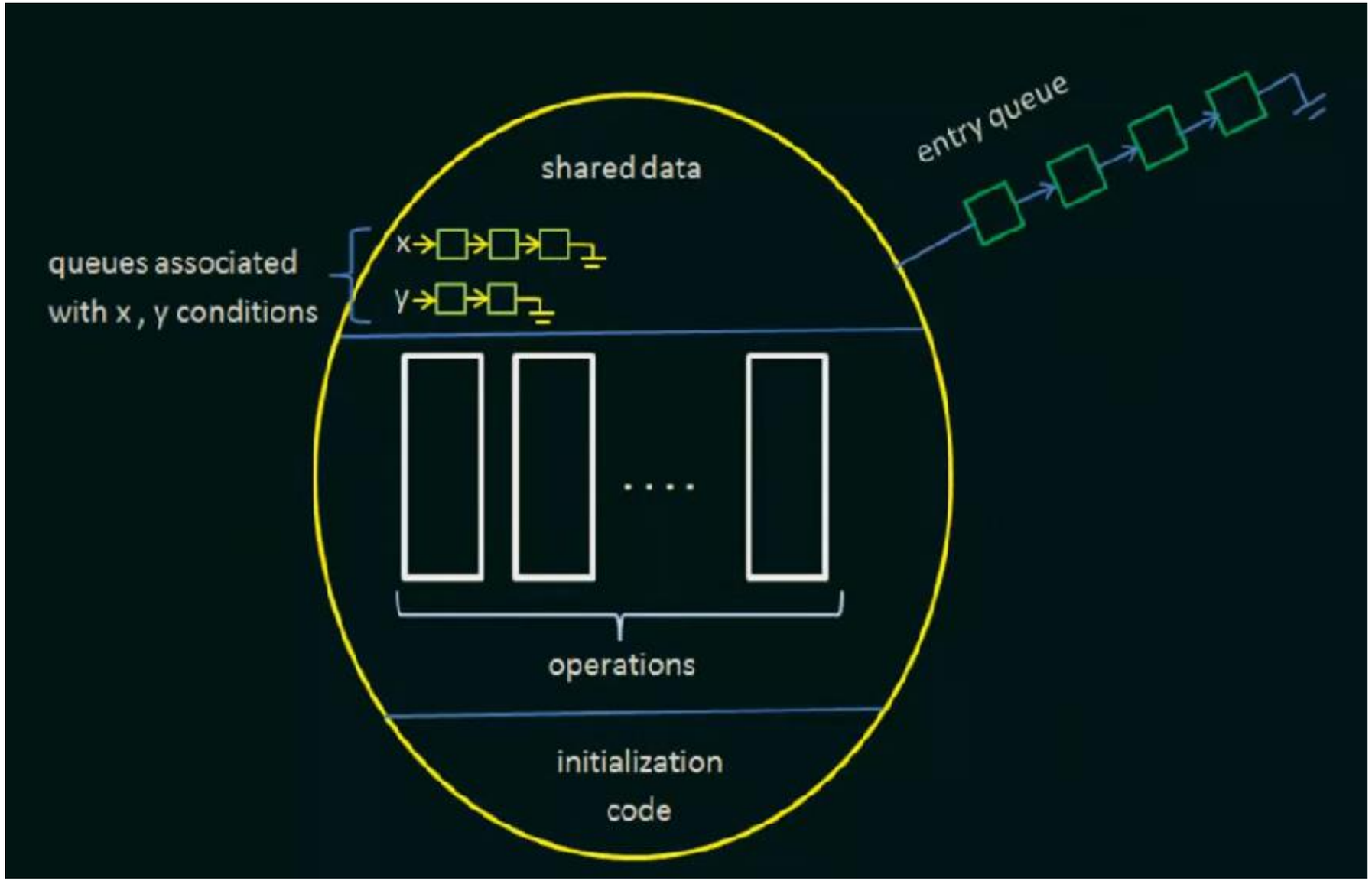
Declaration: condition x,y; //x and y are condition variables it perform wait and signal operations.

Operations that can invoke on condition variables are wait() and signal ().

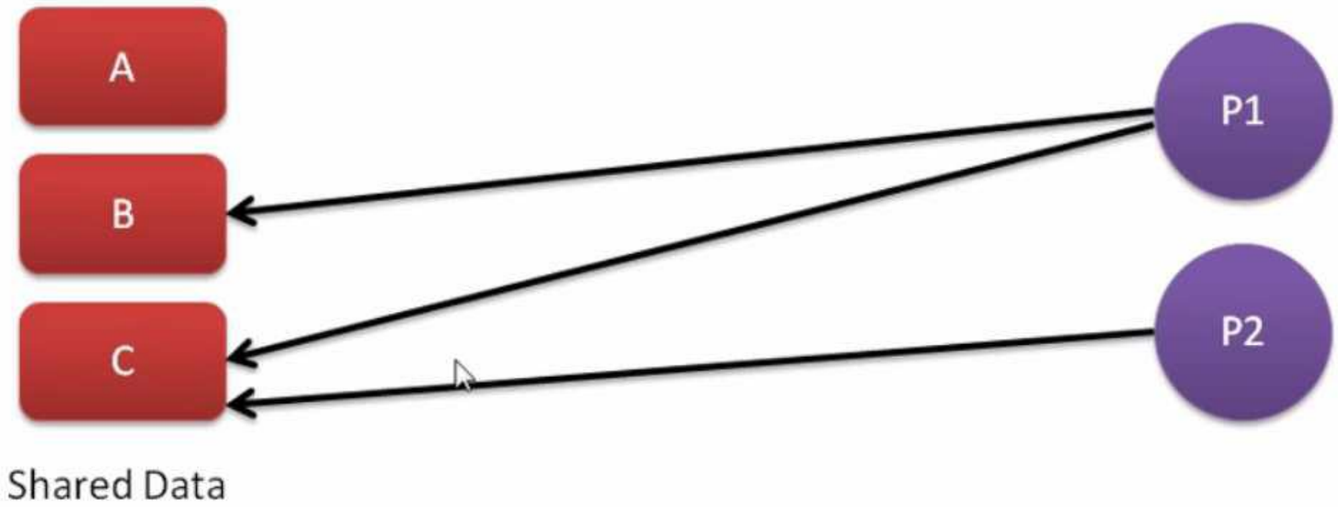
The operation x.wait(); means the process invoking this operation is suspended until another process is invoked x.signal();

x.signal(): resumes the suspended process

Schematic view of Monitors



Monitor: Race condition



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

Dining philosophers problem using monitor

-Monitor provides dead-lock-free solution to the dining philosopher problem.

-this solution imposes the restriction that a philosopher may pickup his chopsticks only if both of them are available.

-To achieve this we need to distinguish three different states, for this purpose we define the following data structure.

```
enum {thinking, hungry, eating} state [5];  //state of 5 philosopher
```

Philosopher i can set the variable state[i] = eating only if his two neighbors are not eating. : (state[(i+4)%5!=eating) and (state[(i+1)%5!=eating).

We also need to declare condition self [5]; where philosopher i can delay himself when he is hungry but unable to obtain the chopsticks.

Dining philosophers problem using monitor

```
monitor dp  {
    enum { THINKING, HUNGRY, EATING } state [5];
    condition self [5] ;

    void pickup (int i)  {
            state [i] = HUNGRY;
            test (i) ;
            if (state [i] != EATING)
            self [i] .wait() ;
            }
    void putdown(int i)  {
            state [i] = THINKING;
            test ((i + 4) % 5) ;
            test ((i + 1) % 5) ;
            }
```

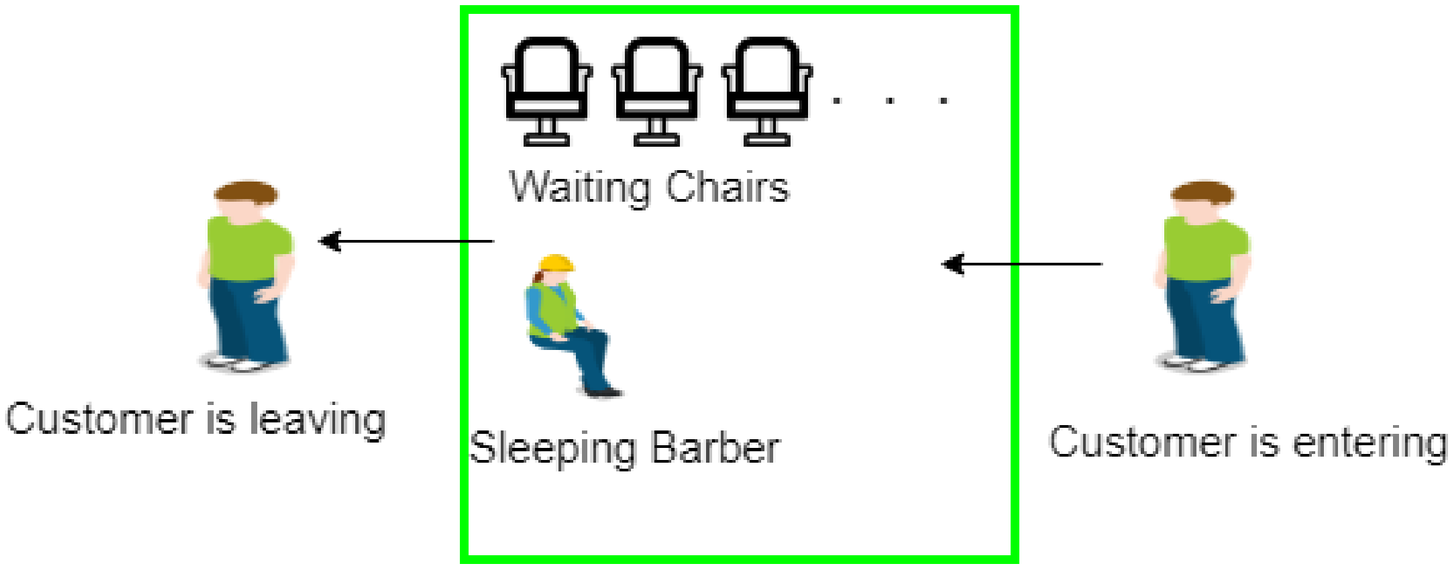
```
    void test (int i)  {
            if ((state [(i + 4) % 5] != EATING) &&
                    (state [i] == HUNGRY) &&
                    (state [(i + 1) % 5] != EATING)) {
                            state [i] = EATING;
                            self [i].signal() ;
                            }
                    }
    initialization-code () {
            for (int i = 0; i < 5; i++)
            state [i] = THINKING ;
            }
```

Sleeping barber problem

-The Sleeping Barber problem is a classic problem in process synchronization that is used to illustrate synchronization issues that can arise in a concurrent system.

-There is a barber shop with one barber and a number of chairs for waiting customers.
-Customers arrive at random times and if there is an available chair, they take a seat and wait for the barber to become available.
-If there are no chairs available, the customer leaves.
-When the barber finishes with a customer, he checks if there are any waiting customers.
-If there are, he begins cutting the hair of the next customer in the queue.
-If there are no customers waiting, he goes to sleep.

Sleeping barber problem

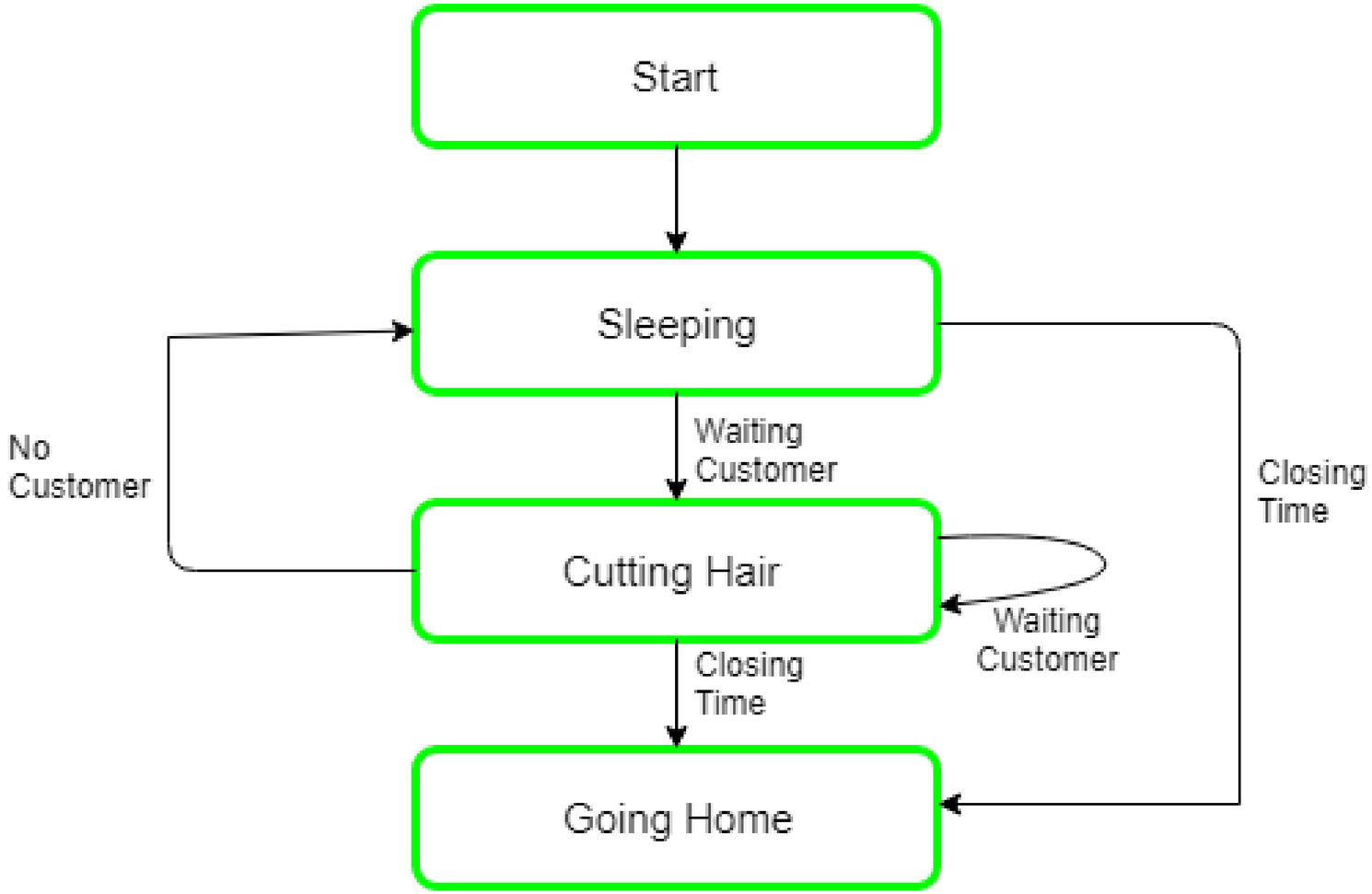


Sleeping barber problem

Conditions:

- 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.

Sleeping barber problem



Sleeping barber problem

Problems:

-Starvation: when a customer enters to barber room but the barber is busy, then he goes to the waiting room and waits, then suddenly the customer decides to leaves waiting room temporarily. The barber finish cutting of previous customer and checks in waiting room, but at that time there is no one (waiting customer has temporarily outside), the barber goes to sleep again. Then the customer return back and sits into the waiting chair. At that time barber goes on sleeping and customer goes on waiting. Such a situation is starvation.

Sleeping barber problem using Semaphore

We take three semaphore to solve such problem

```
Semaphore Customers = 0;
Semaphore Barber = 0;
Semaphore Mutex = 1;
int FreeSeats = N;
Barber {
   while(true) {
    wait(Customers);    /* waits for a customer (sleeps). */
    wait (mutex) /* mutex to protect no. of available seats. M =1*/
       FreeSeats++;            /* bring customer for haircut.*/
   signal(Barber);             /* release the mutex on the chair.*/
   signal(mutex);
     }}
```

Sleeping barber problem using Semaphore

```
Customer {
 while(true) {  /* protects seats so only 1 customer tries to sit  in a chair if that's
the case.*/
        wait(Seats); /* sitting down.*/
        if(FreeSeats > 0) {
             FreeSeats--;          /* notify the barber. */
             signal(Customers);   /* release the lock */
            wait(Seats);         /* wait in the waiting room if barber is busy. */
             signal(Barber);     // customer is having hair cut
        } else {                  /* release the lock */
             signal(Seats);        // customer leaves
        }
    }
}
```

Process Scheduling

Part 4

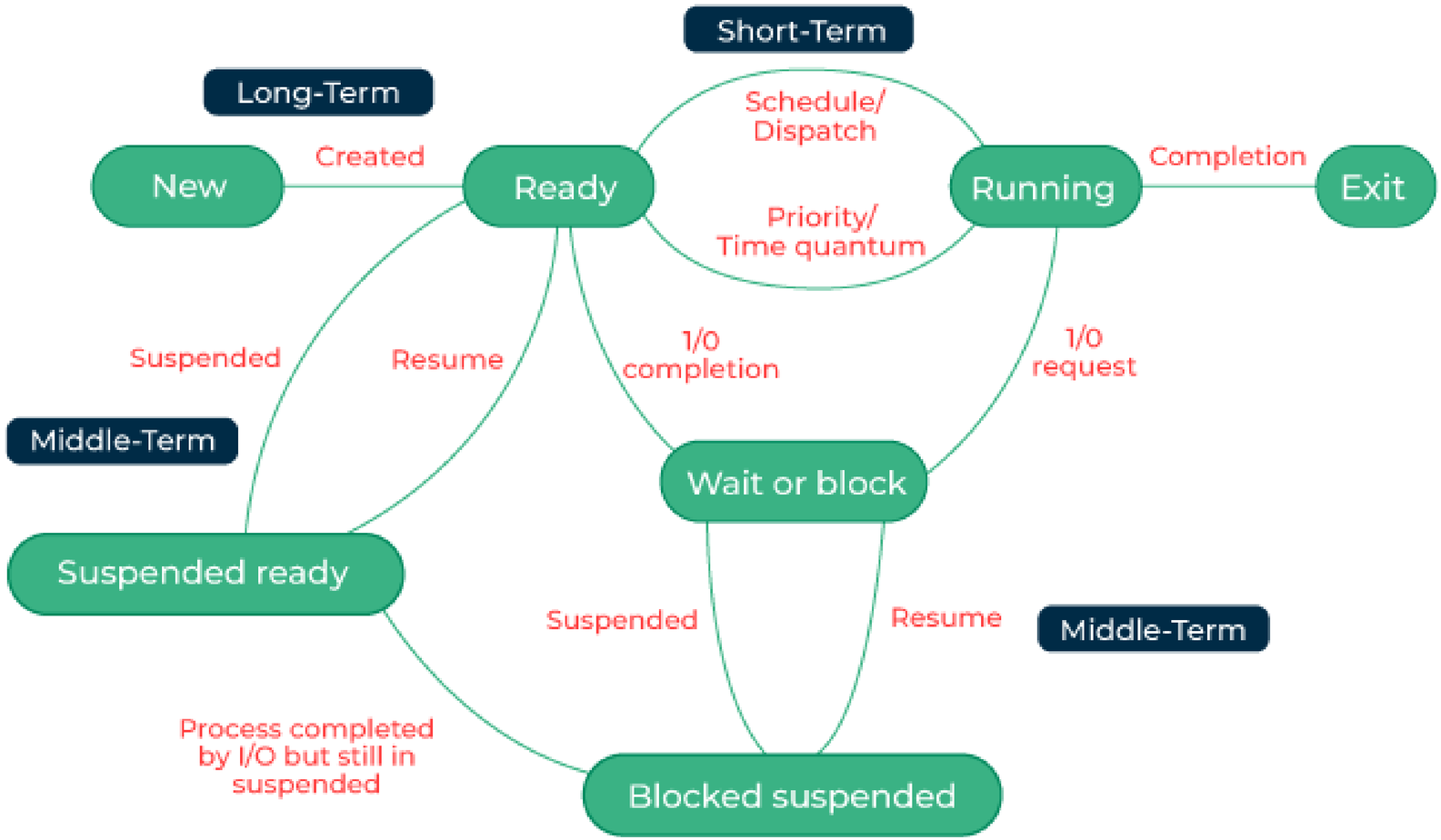
Scheduling: Introduction

-Process scheduling is the activity of the process manager that handles the removal of the running process from the CPU and the selection of another process based on a particular strategy.

-Process scheduling is an essential part of a Multiprogramming operating system.

-Such operating systems allow more than one process to be loaded into the executable memory at a time and the loaded process shares the CPU using time multiplexing.

Process state and use of scheduler



Scheduler

1. Long term scheduler

Long term scheduler is also known as job scheduler. It chooses the processes from the pool (secondary memory) and keeps them in the ready queue maintained in the primary memory.

Rom secondary memory to primary memory

2. Short term scheduler

Short term scheduler is also known as CPU scheduler. It selects one of the Jobs from the ready queue and dispatch to the CPU for the execution.

From primary memory to CPU for execution

3. Medium term scheduler

Medium-term scheduler takes care of the swapped-out processes. If the running state processes need some IO time for the completion then there is a need to change its state from running to waiting. Running to waiting.

Methods of scheduling

There are mainly two types of scheduling methods:

Preemptive Scheduling: Preemptive scheduling is used when a process switches from running state to ready state or from the waiting state to the ready state.

Non-Preemptive Scheduling: Non-Preemptive scheduling is used when a process terminates , or when a process switches from running state to waiting state.

Types of CPU scheduling Algorithms

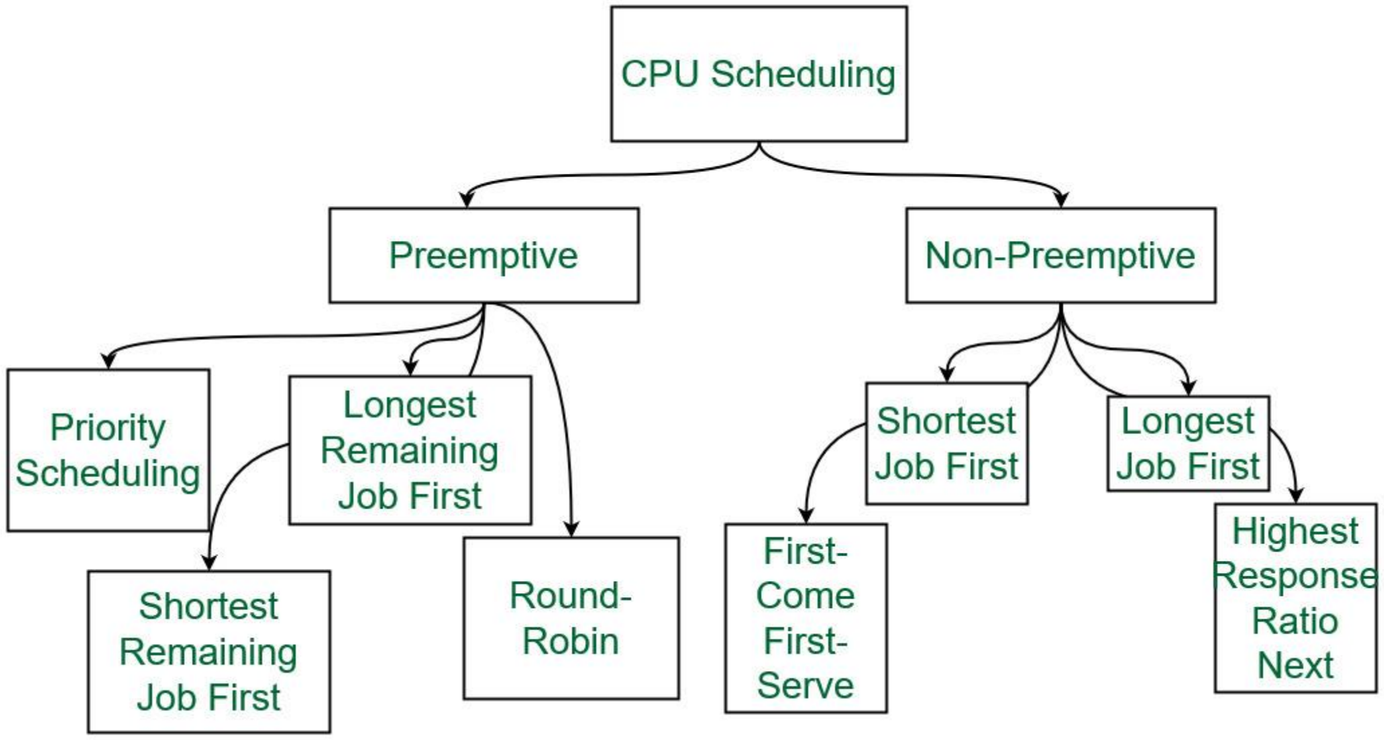
Preemptive:

- Priority Scheduling
- Longest Remaining Job First
- Shortest Remaining Job-First
- Multiple Queues
- Round Robin(RR) Scheduling
- Shortest Remaining Time First

Non-primitive

First-Come, First-Served (FCFS)

Shortest-Job-First (SJF)



Category of scheduling

Categories of Scheduling Algorithms

- **Batch System scheduling**

1. First-Come, First-Served (FCFS)
2. Shortest-Job-First (SJF)
3. Shortest Remaining Time First (SRTF)

- **Interactive system scheduling**

1. Priority Scheduling
2. Longest Remaining Job First
3. Shortest Remaining Job-First
4. Multiple Queues
5. Round Robin(RR) Scheduling

- **Real-time system scheduling**

The objective of scheduling

Objectives of Process Scheduling Algorithm:

- Utilization of CPU at maximum level. Keep CPU as busy as possible.
- Allocation of CPU should be fair.
- Throughput should be Maximum. i.e. Number of processes that complete their execution per time unit should be maximized.
- Minimum turnaround time, i.e. time taken by a process to finish execution should be the least.
- There should be a minimum waiting time and the process should not starve in the ready queue.
- Minimum response time. It means that the time when a process produces the first response should be as less as possible.

The CPU scheduling Terminology

What are the different terminologies to take care of in any CPU Scheduling algorithm?

Arrival Time: Time at which the process arrives in the ready queue.

Completion Time: Time at which process completes its execution.

Burst Time: Time required by a process for CPU execution.

Turn Around Time: Time Difference between completion time and arrival time.

Turn Around Time = Completion Time – Arrival Time (CT-AT)

Waiting Time(W.T): Time Difference between turn around time and burst time.

Waiting Time = Turn Around Time – Burst Time (TAT-BT)

First Come First Serve Algorithm

FCFS considered to be the simplest of all operating system scheduling algorithms. First come first serve scheduling algorithm states that the process that requests the CPU first is allocated the CPU first and is implemented by using **FIFO queue.**

First Come First Serve Algorithm

1. Problem: Consider the following table of arrival time and burst time for five processes P1, P2, P3, P4 and P5.

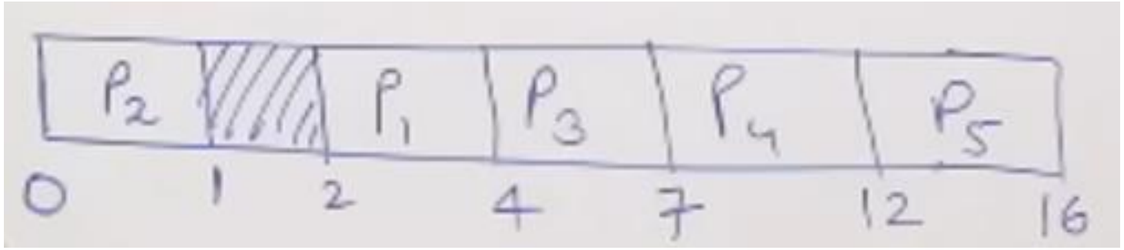
Create Gantt chart and calculate:

1. Completion time
2. Turn Around Time
3. Waiting time
4. Response time
5. Average waiting time
6. Average Response time

Processes	Arrival Time	Burst Time
P1	2	2
P2	0	1
P3	2	3
P4	3	5
P5	4	4

First Come First Serve Algorithm

-Criteria is arrival time
-It uses the queue data structure (FCFS)
-CT= completion time of process
-RT is equal to WT in non-preemptive



Proc	AT	BT	CT	TAT= CT-AT	WT=TAT-BT	RT=WT
P1	2	2	4	2	0	0
P2	0	1	1	1	0	0
P3	2	3	7	5	2	2
P4	3	5	12	9	4	4
P5	4	4	16	12	8	8

First Come First Serve Algorithm

Characteristics of FCFS:

- FCFS supports non-preemptive CPU scheduling algorithms.
- Tasks are always executed on a First-come, First-serve concept.
- This algorithm is not much efficient in performance, and the wait time is quite high.

Advantages of FCFS:

- Easy to implement

Disadvantages of FCFS:

- FCFS suffers from the Convoy effect.

Convoy effect: if processes with higher burst time arrived before the processes with smaller burst time then the smaller process has to wait for a long time for a longer process to release the CPU.

- The average waiting time is much higher than the other algorithms.

First Come First Serve Algorithm

Characteristics of FCFS:

- FCFS supports non-preemptive and preemptive CPU scheduling algorithms.
- Tasks are always executed on a First-come, First-serve concept.
- FCFS is easy to implement and use.
- This algorithm is not much efficient in performance, and the wait time is quite high.

Advantages of FCFS:

- Easy to implement
- First come, first serve method

Disadvantages of FCFS:

- FCFS suffers from Convoy effect.
- The average waiting time is much higher than the other algorithms.
- FCFS is very simple and easy to implement and hence not much efficient.

First Come First Serve Algorithm

Homework:

Write algorithm for first come first serve.

Shortest Job First (SJF) Algorithm

The shortest job first (SJF) or shortest job next, is a scheduling policy that selects the process with the smallest arrival time then burst time to execute next.

-SJF can be solved using both approaches (i.e. preemptive and non-preemptive).

-By default it is non-preemptive

-If it is solved using a preemptive approach then this is called Shortest Remaining Time First (SRTF).

Characteristics of SJF Scheduling:

- It is a Greedy Algorithm.
- It may cause **starvation** if shorter processes keep coming.
- SJF can be used in specialized environments where accurate estimates of running time are available.
- It can improve process **throughput** by making sure that shorter jobs are executed first. (Throughput is the amount of work completed in a unit of time)

Shortest Job First (SJF) Algorithm

1. Problem: Consider the following table of arrival time and burst time for four processes P1, P2, P3 and P4. Solve it using non-preemptive approach.
Create a Gantt chart and calculate:

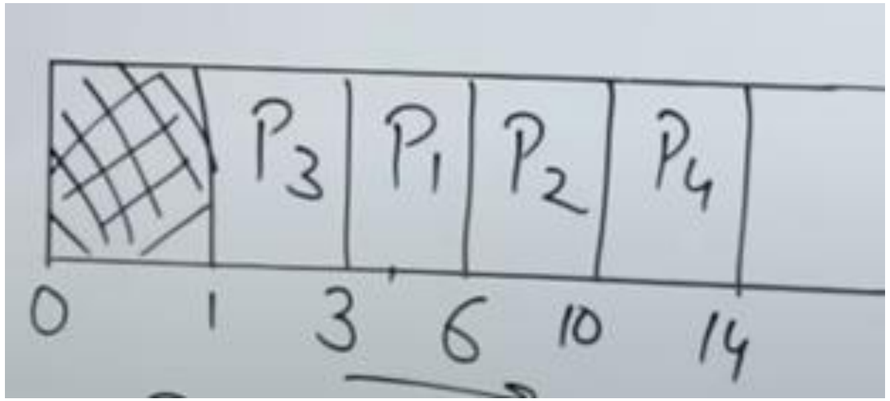
1. Completion time
2. Turn Around Time
3. Waiting time
4. Response time
5. Average waiting time
6. Average Response time

Processes	Arrival Time	Burst Time
P1	1	3
P2	2	4
P3	1	2
P4	4	4

Shortest Job First (SJF) Algorithm

-Criteria is shortest burst time first

-Non-preemptive approach



Proc	AT	BT	CT	TAT= CT-AT	WT=TAT-BT	RT=WT
P1	1	3	6	5	2	2
P2	2	4	10	8	4	4
P3	1	2	3	2	0	0
P4	4	4	14	10	6	6

Shortest Job First (SJF) with preemption/Shortest Remaining Time First (SRTF)

1. Problem: Consider the following table of arrival time and burst time for five processes P1, P2, P3, P4 and P5. Solve it using Preemptive approach.
Create a Gantt chart and calculate:

1. Completion time
2. Turn Around Time
3. Waiting time
4. Response time
5. Average waiting time
6. Average Response time

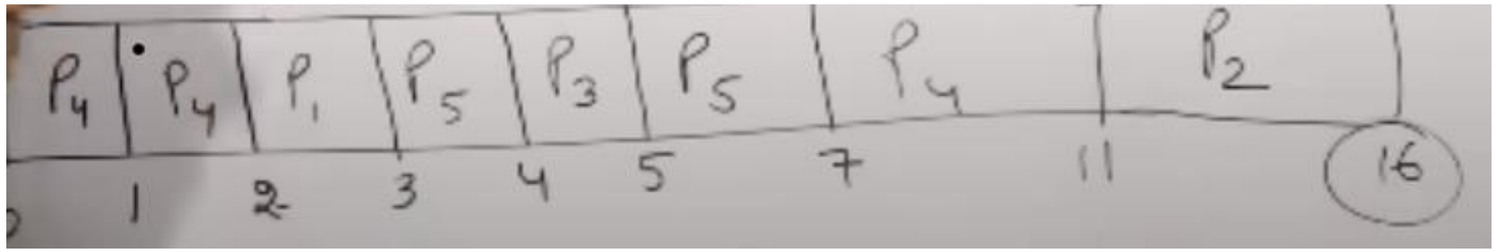
Processes	Arrival Time	Burst Time
P1	2	1
P2	1	5
P3	4	1
P4	0	6
P5	2	3

Shortest Job First (SJF) with preemption/Shortest Remaining Time First (SRTF)

-Whenever new process comes, there may be preemption of the running process

-Consideration: burst time

-preemptive approach



Proc	AT	BT	CT	TAT= CT-AT	WT=TAT-BT	RT=first CPU allocated time – AT
P1	2	1	3	1	0	2-2=0
P2	1	5	16	15	10	11-1=10
P3	4	1	5	1	0	4-4=0
P4	0	6	11	11	5	0-0=0
P5	2	3	7	5	2	3-2=1

Shortest Job First (SJF) with preemption/Shortest Remaining Time First (SRTF)

1. Problem 2: Consider the following table of arrival time and burst time for four processes P1, P2, P3, and P4. Solve it using the Preemptive approach.
Create a Gantt chart and calculate:

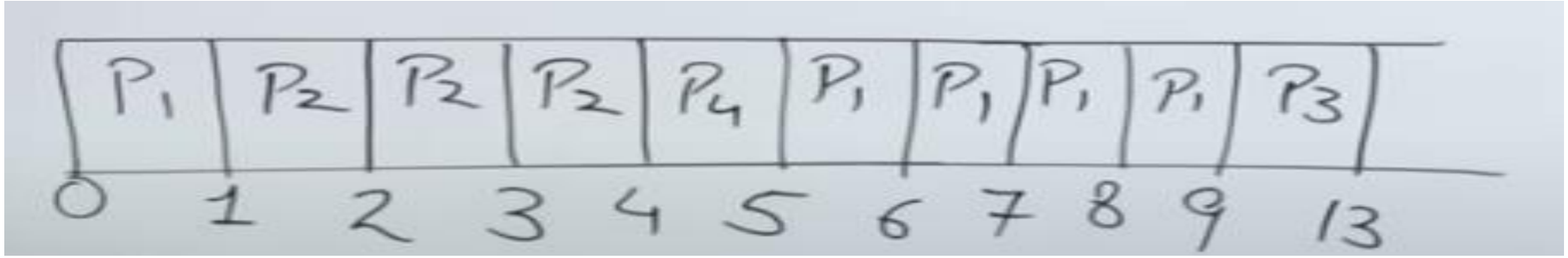
1. Completion time
2. Turn Around Time
3. Waiting time
4. Response time
5. Average waiting time
6. Average Response time

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

Shortest Job First (SJF) with preemption/Shortest Remaining Time First (SRTF)

-Consideration: burst time

-preemptive approach



Proc	AT	BT	CT	TAT= CT-AT	WT=TAT-BT	RT=first CPU allocated time – AT
P1	0	5	9	9	4	0
P2	1	3	4	3	0	0
P3	2	4	13	11	7	7
P4	4	1	5	1	0	0

Shortest Job First (SJF) with preemption/Shortest Remaining Time First (SRTF)

Advantages:

-SRTF gives minimum WT and TAT

-.Better response time than FCFS

-MAX throughput (Throughput is the amount of work completed in a unit of time)

Disadvantages:

- SJF may cause very long turn-around times or starvation.
- In SJF job completion time must be known earlier, but sometimes it is hard to predict.
- Sometimes, it is complicated to predict the length of the upcoming CPU request.

Starvation A process that is present in the ready state and has low priority keeps waiting for the CPU allocation because some other process with higher priority comes with due respect time. So low priority process keeps waiting.

Preemptive: Priority scheduling Algorithm

Priority Scheduling is a method of scheduling processes that is based on priority. In this algorithm, the scheduler selects the tasks to work as per the priority.

-Priority can be solved using both approaches (i.e. preemptive and non-preemptive).

Preemptive: Priority scheduling Algorithm

Characteristics of Priority Scheduling

1. A CPU algorithm that schedules processes based on priority.
2. It used in Operating systems for performing batch processes.
3. If two jobs having the same priority are READY, it works on a FIRST COME, FIRST SERVED basis.
4. In priority scheduling, a number is assigned to each process that indicates its priority level.

Preemptive: Priority scheduling Algorithm

1. Problem: Consider the following table of arrival time and burst time and priority for four processes P1, P2, P3, and P4. Solve it using the Preemptive approach. Consider: Higher the value higher the priority

Create a Gantt chart and calculate:

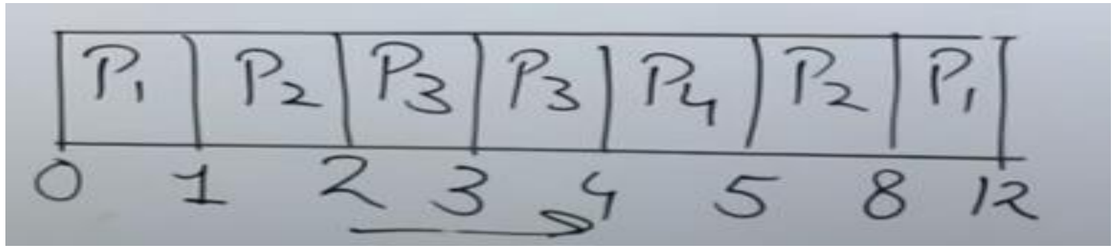
1. Completion time
2. Turn Around Time
3. Waiting time
4. Response time
5. Average waiting time
6. Average Response time

Priority	Proc	AT	BT
10	P1	0	5
20	P2	1	4
30	P3	2	2
40	P4	4	1

Preemptive: Priority scheduling Algorithm

-Criteria: priority (higher the number higher the priority)

-preemptive approach



Priority	Proc	AT	BT	CT	TAT= CT-AT	WT=TAT-BT	RT=first CPU allocated time – AT
10	P1	0	5	12	12	7	0
20	P2	1	4	8	7	3	0
30	P3	2	2	4	2	0	0
40	P4	4	1	5	1	0	0

Preemptive: Priority scheduling Algorithm

Advantages:

1. Priority-based scheduling ensures that high-priority processes are executed first, which can lead to faster completion of critical tasks.
2. Priority scheduling is useful for real-time systems that require processes to meet strict timing constraints.
3. Priority scheduling can reduce the average waiting time for processes that require a significant amount of CPU time.

Disadvantages:

1. Starvation: If the system is heavily loaded with high-priority processes, low-priority processes may never get a chance to execute.
2. A process will be blocked when it is ready to run but has to wait for the CPU because some other process is running currently.

Preemptive: Round Robin Algorithm

-Round Robin is a CPU scheduling algorithm where each process is cyclically assigned a fixed time slot.
-It is the preemptive version of the First come First Serve CPU Scheduling algorithm.
-Round Robin CPU Algorithm generally focuses on the sharing technique.
-The period of time for which a process or job is allowed to run in a pre-emptive method is called time quantum.

Preemptive: Round Robin Algorithm

1. Problem: Consider the following table of arrival time and burst time for four processes P1, P2, P3, and P4. Solve it using the RR Preemptive approach.
Create a Gantt chart and calculate:

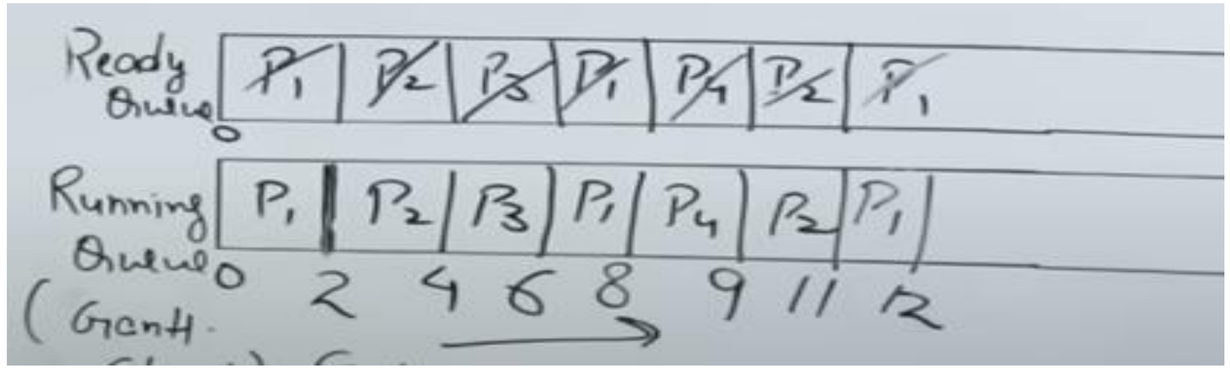
1. Completion time
2. Turn Around Time
3. Waiting time
4. Response time
5. Average waiting time
6. Average Response time

Processes	Arrival Time	Burst Time
P1	0	5
P2	1	4
P3	2	2
P4	4	1

Consider time quantum=2

Preemptive: Round Robin Algorithm

-Consideration: time quantum
-preemptive approach



Proc	AT	BT	CT	TAT= CT-AT	WT=TAT-BT	RT=first CPU allocated time – AT
P1	0	5	12	12	7	0
P2	1	4	11	10	6	1
P3	2	2	6	4	2	2
P4	4	1	9	5	4	4

Preemptive: Round Robin Algorithm

Consider the following table of arrival time and burst time for five processes P1, P2, P3, P4 and P5. Solve it using the RR Preemptive approach.

Create a Gantt chart and calculate:

1. Completion time
2. Turn Around Time
3. Waiting time
4. Response time
5. Average waiting time
6. Average Response time

Processes	Arrival Time	Burst Time
P1	0	8
P2	0	2
P3	0	7
P4	0	3
P5	0	5

Consider time quantum=2

Consider context switch time= 1

Preemptive: Round Robin Algorithm

	BT	AT	CT	TT	WT	R.T
P_1	~~32~~ 28	0	32	32	24	0
P_2	2	0	6	6	4	4
P_3	~~7~~ 4	0	34	34	27	7
P_4	3	0	40	40	11	11
P_5	22	0	29	27	24	15

let, Context switch time is 1 unit

$T_q = 3$

Ready queue

~~P_1~~	~~P_2~~	~~P_3~~	~~P_4~~	~~P_5~~	P_1	P_3	P_5	P_1	P_3

Gantt chart:

P_1	P_2	P_3	P_4	P_5	P_1	P_3	P_5	P_1	P_3

0 3 6 9 11 14 15 18 19 22 23 26 27 29 30 32 33 34

no. of context switcher = 9

CPU utilization = $\frac{\text{Execeted time}}{\text{Actual time}}$

$\frac{25}{34} \times 100\%$

$= 73.5\%$

Preemptive: Round Robin Algorithm

The Advantages of Round Robin CPU Scheduling are:

- A fair amount of CPU is allocated to each job.
- Because it doesn't depend on the burst time, it can truly be implemented in the system.
- It is not affected by the convoy effect or the starvation problem as occurred in First Come First Serve CPU Scheduling Algorithm.

Disadvantages

- Low Operating System slicing times will result in decreased CPU output.
- Round Robin CPU Scheduling approach takes longer to swap contexts.
- Time quantum has a significant impact on its performance.
- The procedures cannot have priorities established.

Find me

9851083215

Santosh.it288@mail.com

www.phtechno.com

Kathmandu