Unit-2: Process Management

-Ankit Pangeni

DATE

Introduction

Process

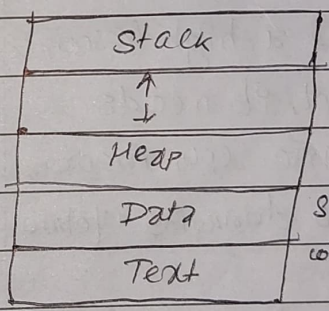
A process is an instance of a program running in a computer. It is close in meaning to task, a term used in some operating systems.

-> In UNIX and some other operating systems, a process is started when a program is initiated.

-> A process is basically a program in execution

=> example: we write our programs in a text file and when we execute this program, it becomes a process which performs all the tasks mentioned in the program

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



Stack: contains temporary data local variables, method/function parameter.

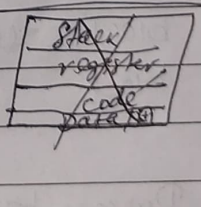
Heap: dynamically allocated memory

Data: contains global & static variable.

Text: includes the current activity represented by value of PC & content of processor registers

fig: Layout of a process inside main memory.

or



Program

Program is an executable file containing the set of instructions written to perform a specific job on your computer. Ex. chrome.exe is an executable file to view the web pages. Programs are not stored in primary memory.

DATE

They are stored in the disk or a secondary memory

→ It is basically a piece of code which may be a single line or a million of line.

→ When we compare a program with a process, we conclude that, process is the dynamic instance of a program.

	Process	Program
i)	A process is an instance of a program running in a computer. It is an activity or a task.	i) Program is a set of instructions to perform a specific task or job in your computer
ii)	Process is a dynamic entity.	ii) Program is a static entity.
iii)	Process has a high resource requirement, it needs resources like CPU, memory address, I/O during its lifetime	iii) Program does not have any resource requirement, it only requires memory space for storing the instructions
iv)	Process has its own control block called process control block.	iv) Program does not have any control block
v)	Process has a shorter and limited lifespan because it gets terminated after the completion of a task.	v) A program has a longer lifespan because it is stored in the memory until it is manually deleted

DATE

Multiprogramming

Multiprogramming is the ability of an operating system to execute more than one program on a single processor machine. More than one task / program / job / process can reside into the main memory at a point of time.

Ex: Computer running Microsoft word and chrome at the same time.

→ In a multiprogramming system, there are one or more programs located in the main memory which are ready to execute. Only one program at a time is able to get the cpu for executing its instructions while all the others are waiting for their turn.

→ During this time, cpu switches back and forth from process to process. It happens in few milliseconds.

→ Its main purpose is to maximize the use of cpu time without wasting any of it (ie. make the cpu busy as long as there are processes to execute).

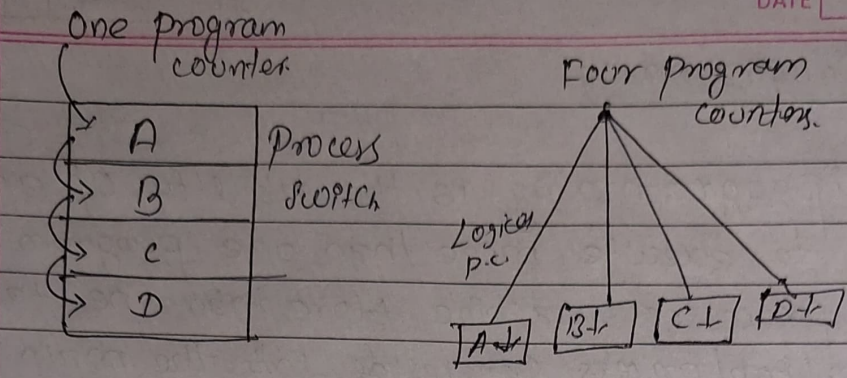
→ Note that for a such system to work properly, it should be able to load multiple programs into separate areas of the main memory & it should be able to make fragmentation as programs enter or leave the memory.

Process Model.

In process model, all the runnable software on the computer is organized into a number of sequential processes. Each process has its own virtual central processing unit (cpu). The real cpu switches back and forth from process to process. A process is basically an activity. It has a program, input, output, and a state.

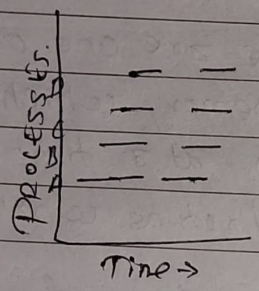
classmate

PAGE



(a) MULTIPROGRAMMING FOUR PROGRAMS

(b) CONCEPTUAL MODEL FOR FOUR INDEPENDENT, SEQUENTIAL PROCESSES



(c) ONLY ONE PROGRAM IS ACTIVE AT ONCE

As we see in fig(a), a computer is multiprogramming four programs in its memory. In fig(b), we see four processes each with its own logical program counter and each one is running independently of the other ones. As there is only one physical program counter, so when each process runs, its logical program counter is loaded into the real PC. When it is finished, the physical PC is saved in the process stored logical PC in memory. In fig (c), we see that, when viewed over a long enough time interval, all the processes have made progresses, but at any given instant only one process is actually running.

DATE

Process states (Process life cycle)

When a process executes, it passes through different states. These stages may differ in different operating systems, & the names of these states are not standardized. In general, a process can have one of the following five states at a time.

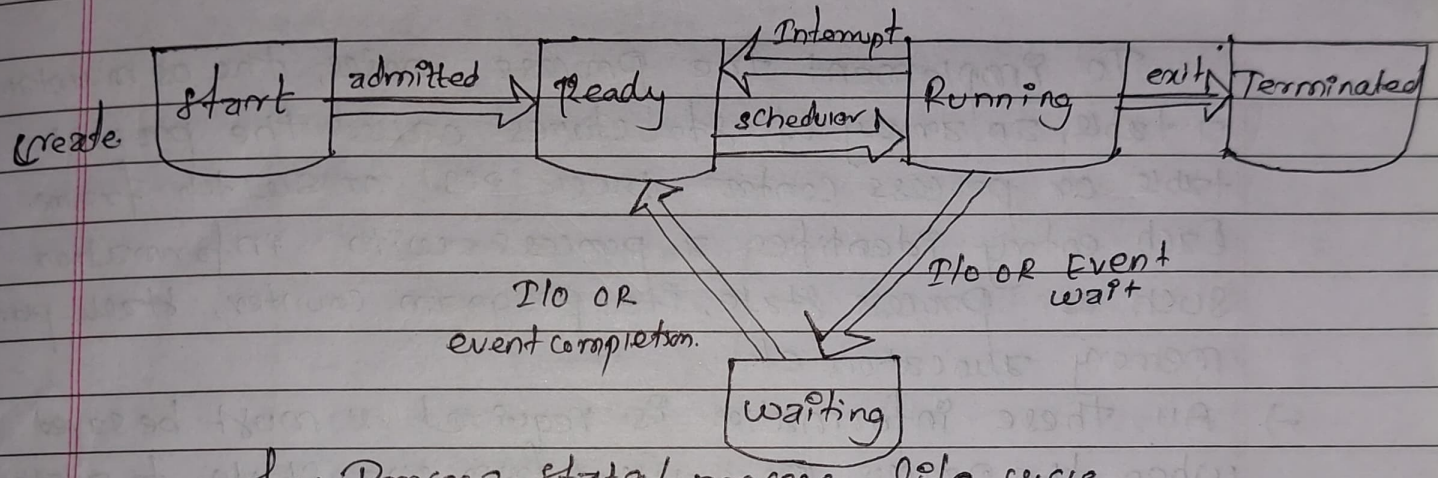


fig: Process states / process life cycle.

1. Start: This is the initial state, when a process is first started / created.
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 OS, so they can run.
3. Running: After ready state, the process state is set to running and the processor executes its instructions.
4. Waiting: 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.
5. Terminated or exit: Once the process finishes its execution or it is terminated by OS, it is moved to the terminated state where it waits to be removed from the main memory.

DATE

Note: In ready state, process may come into this state after START state or while RUNNING it by but interrupted by the scheduler to assign cpu to some other process.

Process Control Block/ Process Table / Switch frame.

To implement the process model, the OS maintains a table, an array of structures called the process table or process control block (PCB) or switch frame. Each entry identifies a process with information such as process state, its program counter, stack pointer, memory allocation, etc.

→ All these information is required & must be saved when the process is switched from one state to another

→ When the process makes transitions from one state to another, the OS must update information in the processes PCB.

The following information is stored in a process table (components of PCB): reqd when it is switched from one state to another

- Pointer: It is a stack pointer to retain current position of process
- Process state: which may be start, ready, running, waiting or terminated.
- Process number: Each process is identified by its unique process number called process ID.
- Program Counter: Which indicates the address of the next instruction to be executed.
- CPU registers: which vary in number and type depending on the concrete microprocessor architecture.

classmate

PAGE

DATE

- Memory management Information : which includes base and bounds registers or page table.
- I/O status information : Composed I/O requests, I/O devices allocated to this process, a list of open files & so on.
- Processor scheduling information: which includes process priority, pointers to scheduling queues & any other scheduling parameters.

Pointer	State
Process number	
Program counter	
Registers	
Memory limits	
Open files lists	
MPSC Accounting & status data	

fig: Process Table Architecture

Threads

A thread is a single sequential flow of execution of tasks of a process. There is a way of thread execution inside the process of any operating system.

→ There can be more than one thread inside the process & each thread of same process makes use of separate program counter. & no thread can exist outside process.

→ Thread is often referred to as light weight process.

→ Thread provide a way to improve application performance through parallelism.

→ The process can be split down into so many threads. Ex: in a browser, many tabs can be viewed as threads.

PAGE

classmate

DATE

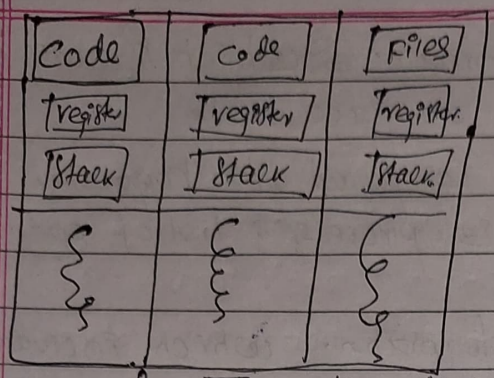


Fig: Three threads of same process

Need of Thread:

- It takes far less time to create a new thread in an existing process than to create a new process
- Threads can share common data, they do not need to use Inter-process communication
- context switching is faster when working with threads
- It takes less time to terminate a thread than a process

	Process	Thread
i)	A process is an instance of a program running in a computer.	i) A thread is a single sequential flow of execution of tasks of a process.
ii)	It is heavy weight or resource intensive.	ii) It is light weight, taking lesser resources than a process.
iii)	In multiple processes, each process operates independently of the others.	iii) One thread can read, write or change another threads data.
iv)	System calls involved in process.	iv) There is no system calls involved
v)	OS treats different processes differently.	v) All user level threads are treated as single task for OS.
vi)	Different process have different copies of Data, files, code.	vi) Threads share same copy of code & data but different stack & register memory.

DATE

vii)	Context switching is slower.	vii) Context switching is slower
viii)	Blocking a process will not block another process.	viii) Blocking a thread will block entire process.

Similarities: Like processes, threads share CPU & only one thread active at a time. Like processes, thread can create children

Types of Threads:

Threads are implemented in following two ways:

1) User level thread.
   - -> User level thread are implemented by the users.
   - -> The operating system does not recognize the user-level thread & these are easily implemented by the users.
   - -> If a user performs a user level thread blocking operation, the whole process is blocked.
   - -> The kernel level thread knows nothing about user level thread.
   - -> They require no system calls.

Advantages:

- It can be easily implemented & is more faster than kernel level
- It is more faster & efficient
- Context switching is faster.
- It is simple to create, switch & synchronize threads without interfering the process

Disadvantages

- User level threads lack coordination between the thread & kernel
- If a user performs a user level thread blocking operation, the whole process is blocked.

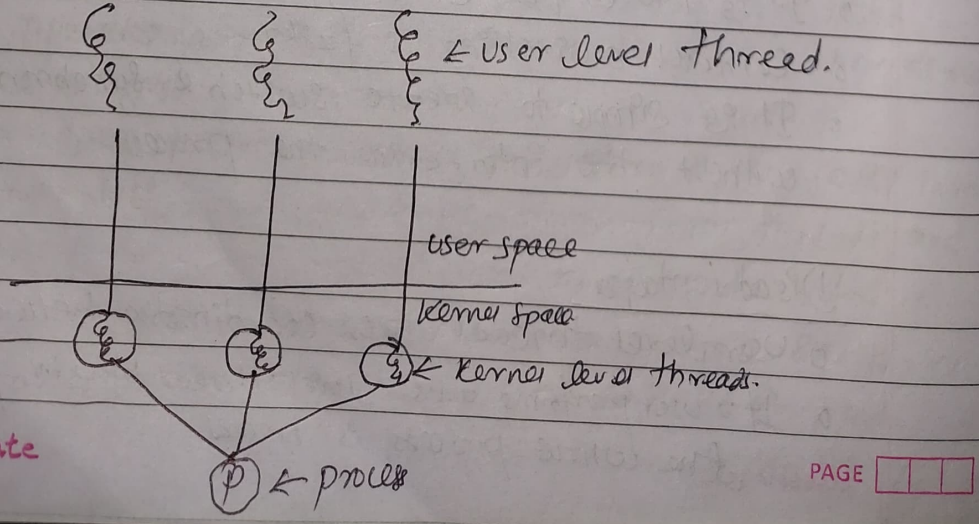
DATE

ii) Kernel level thread.

→ Kernel level thread are implemented by kernel
→ The kernal thread recognizes the operating system & is managed by OS
→ The kernel level thread offers a system call to create & manage threads from user-space.
→ The implementation of kernel threads is more difficult than the user threads.
→ If a kernel thread performs a blocking operation, the whole process is not blocked.

Advantages. level
- The kernel thread is fully aware of all threads.
- Kernel-level threads are good for applications that ~~require~~ block the frequency.
- If a kernel thread performs a blocking --- (same as above)

Disadvantages
- The kernel thread manages & schedules all threads
- Context switching is slower
- Implementation is difficult
- Kernel level threads are slower than user-level threads



classmate

PAGE

DATE

Note: Some benefits of threads (imp)
- Enhanced throughput of the system
- Effective

* Inter process Communication.

→ Inter-process communication is a mechanism that the exchange of data between processes.
→ Ipc allows one application to control another, thereby enabling data sharing without interference.
→ It enables data communication by allowing processes to use segments, semaphores, and other methods to share memory and information.
→ The idea of IPC is based on Task Control Architecture (TCA). It is a flexible technique that can send and receive variable length arrays, data structures, and lists.

→ There are three issues:- The first is how one process can pass information to another process. The second is to make sure two or more process don't get in each others way. The third is proper sequencing when dependencies are present. ex: If process A produces data & process B prints, B has to wait until A has produced some data before starting to print.

PAGE

classmate

DATE

Race condition.

A race condition is an undesirable situation that occurs when a system attempts to perform two or more process/operations at the same time but because of the nature of the system, the operation must be done in proper sequence to be done correctly.

It occurs when two or more threads can access the shared data & they try to change it at the same time. In some OS, processes that are coorking together may share common storage that each one can read and write. The shared storage may be in main memory or it may be a shared file.

Ex: Let's consider two processes P_1 & P_2 which are running in parallel.

int shared=5

P_1	P_2
1. int X= shared;	int y= shared;
2. X++;	y--;
3. sleep(1);	sleep(1);
4. shared= X;	shared= y;

Here, P_1 & P_2 share common variable called shared. If P_1 gets cpu first and then the value of X becomes 5. Then, next instruction X++ makes X value 6. Then, the next instruction sleep(1) gets executed. Since, cpu can't be idle, so it executes another parallel process P_2 by context switching. After saving values of P_1 in PCB. Then, y becomes 5. After then, y-- makes y=4. Then, after executing sleep(1) it again gives cpu

DATE

access to P_1. Then, 4th instruction of P_1 gets executed and value of shared variable becomes 6. And then process P_1 gets terminated. Then, CPU is accessed by P_2 and 4th instruction of P_2 gets executed and value of shared variable becomes 4.

But, this is wrong, since the value of shared variable should be 5 after increasing & decreasing its value by 1 (ie. x++ & y--). So, this problem is called race condition.

→ To prevent ~~synchr~~ race condition proper synchronization methods are used. It can also be avoided if the critical section is treated as an atomic instruction.

To avoid such issues, synchronization mechanisms like locks, semaphores, and mutexes can be used to ensure that only one process accesses the shared resource at a time.

* Critical Section. (region)

→ The part of the program where the shared memory is accessed is called the critical region or sectn.

→ If we could arrange matters such that no two processes were ever in their critical regions at the same time, we could avoid race conditions.

→ The critical section is given as follows:

```
do
{
    Entry section        ← control the entry into critical section & gets a lock on required resources
    critical section     ← The critical part
    Exit section         ← Remove the lock from the resources left the others known that critical section is over
    Remainder section    ← Rest of the section.
} while (TRUE);
```

DATE

The critical section problem needs a solution to synchronize the different processes. The solution to the critical section problem must satisfy the following conditions:

- (Mutual Exclusion): It implies that 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.

- (Progress): It means, if a process is not using the critical section, then it should not stop any process from accessing it. In other words, any process can enter a critical section if it is free.

- (Bounded Waiting): It means each process must have a limited waiting time. It should not wait endlessly to access the critical region.

DATE

* Implementing Mutual Exclusion.

Mutual exclusion states that only one process can be inside the critical section at any time. so that race condition will not occur.
For implementing mutual exclusion we use following.

1. Mutual exclusion with Busy waiting
   - → Disable Interrupt
   - → Lock variables
   - → strict Alternation
   - → Peterson solution
   - → Test & set lock.
2. sleep and wake up.
3. Semaphore, Monitors & Message Passing.

* Mutual Exclusion with Busy waiting.

In this section, we will examine various proposals for achieving mutual exclusion, so that while one process is accessing the shared memory, no other process will enter its critical region & cause trouble.

→ Disable Interrupt :

The simplest solution is to have each process disable all interrupt just after entering its critical region & re-enable just before leaving it. While interrupts disabled, no clock interrupts can occur. This approach is generally

classmate PAGE
The cpu is only switched from process to Process as a result of clock interrupt & with interrupt off cpu won't switch

DATE

unattractive because it is unwise to give user processes the power to turn off interrupts. Suppose that one of them did it & never turned them on again? That could be the end of the system.

→ Lock variables:

It is a software sowtion for mutual exclusion. When there is a single shared variable, lets say 0. When a process wants to enter its critical region it first tests the lock. If the lock is 0, the processor sets it to 1 and enters the critical region. If the lock is already 1, the process just waits until it becomes 0.

Thus, a 0 means that no process is in its critical region and 1 means that there is some process in its critical region

→ Strict Alteration. or Turn variable.

This approach is also a software mechanism implemented at user mode. It is a busy waiting sowtion which can be implemented only for two processes. In this approach, a turn variable is used which is actually a lock.
Let the two processes be Pi and Pj & they share a variable called turn variable. The pseudo code of the program are as follows:

DATE

Pi	Pj
No-critical section	No-critical section
while (turn != i)	while (turn != j)
critical section	critical section
turn = j	turn = i
No-critical section.	No-critical section.

In, lock variable, process was entering in the critical section only when the lock variable is 1. More than one process could see the lock variable as 1 at the same time hence the mutual exclusion was not guaranteed.

But, in the turn variable approach, there are only two values possible for turn variable, i or j, if its value is not i then it will definitely be j or vice versa. Hence, in the entry section, in general, the process Pi wont enter critical section until its value is j or process Pj wont enter until its value is i. which addresses the problem of lock variable.

→ Peterson's solution.

By combining the idea of taking turns with the idea of lock variables & warning variables, a Dutch Mathematician T. Deckker made a software solution to mutual exclusion problem that doesn't require strict alternation.

Initially neither process 0 calls for entering critical region. It indicates its interest & set turn to 0. Since, process 1 is not interested,

DATE

It will return immediately from critical region. If process 1 calls to enter the critical region, it will hang there until interested process goes off. So, when process 0 leaves critical region, then only process 1 turn comes.

→ Test and Set lock (TSL Instruction)

It uses both software and hardware. The instruction is used as:

TSL, RX, LOCK where

TSL = Test & set lock

RX = Register

LOCK = Variable.

- It reads the contents of memory word (lock) into register RX. Then stores non-zero value at memory address lock.
- No other process can access the memory word until the instruction is finished.
- The CPU executing TSL instruction locks the memory bus to prohibit other CPUs from accessing memory until it is done.
- It's main disadvantage is that, busy waiting causes wastage of cpu time.

DATE

* Sleep and wake up

Both Peterson's solution and solution using TSL are correct but both have the defect of requiring busy waiting. It wastes cpu time.

→ Sleep is a system call that causes the caller to block, that is suspended until another process wakes it up

→ The wake up call has one parameter that awakes the process.

→ Alternatively, both sleep & wake up each have one parameter, a memory address used to match up sleeps with wakeups

* Semaphore.

In 1965, Dijkstra proposed a new and very difficult technique for maraging concurrent process by using the value of simple integer variable to synchronise the progress of interacting process. This integer variable is called semaphore.

→ So, it is basically a synchronising tool accessed by operation such as wait & signal designated by P(S) & V(S).

→ Semaphore is a variable which can hold only a non-negative integer value shared between all threads, with operation "wait" & "signal".

* Monitor

A monitor is a set of multiple routines which are protected by a mutual exclusion lock. None of the routines in the monitor can be executed by a

DATE

thread until that thread requires a lock. It means only thread can execute within the monitor at a time. Any other threads must wait for the thread thats currently executing to give up control of the lock.

However a thread can actually suspend itself inside a monitor & then wait for an event to occur. If this happens, another thread is given the opportunity to enter the monitor. The thread that was suspended will eventually be notified that the event it was waiting for has now occured which means it can wake up and reacquire the lock.

* Message Passing.

It is a technique for processes to communicate & to synchronise their actions. In message system, processes communicate without restoring to shared variable (with each other) This method of interprocess communication has two principes. send & receive.

send (destination, & message)
receive (source, & message)

If Process P and q want to communicate, they need to have following properties:-

i) Establish a communication link between them.
ii) Exchange message through send & receive functioning
iii) Implementation of communication link.
   - Physical (shared memory, Bus)
   - Logical (logical address)

DATE

Mode of communication between two processes can take place through two methods:

- Direct Addressing (communication)

In direct addressing, each process that wants to communicate must explicitly name the receiver or sender of the communication. The send and receive primitives are defined as:

Send (P, message) ⇒ Send a message to process P

Receive (Q, message) ⇒ Receive a message from process Q.

Here, a link is established automatically between every pair of processes that want to communicate. The processes need to know only each other's identity to communicate. Exactly one link exists between each pair of processes

- Indirect Addressing (communication).

With indirect addressing, the messages are sent to and received from mailboxes, or ports. A mailbox can be viewed abstractly as an object into which messages can be placed by processes & from which messages can be removed. Each mailbox has a unique identification. In this scheme, a process can communicate with some other process via a number of different mailboxes. Two process can communicate only if they share a mailbox.

DATE

The send and receive primitives are defined as:-

- Send (A, message) => send a message to mailbox A
- Receive (A, message) => Receive a message from mailbox A.

A link is established between a pair of processes only if both members have a shared mailbox. A link may be associated with more than two processes. A number of different links may exist between each pair of communicating processes, with each link corresponding to one mailbox.

DATE

Classical IPC Problems

The operating system literature is full of interesting problems that are analysed using a variety of synchronization methods. Here, we will examine three of the better known problems.

- Producer-Consumer problem
- Sleeping Barber problem
- Dining Philosopher problem.

* Producer-Consumer problem

In computing the Producer-consumer problem (bounded-buffer) problem is a classical example of a multi-process synchronisation problem. The problem describes two processes: the producer and consumer, who share a common fixed size buffer used as a queue.

The producer's job is to generate data and put it into the buffer and start again. At the same time, the consumer is consuming the data (ie. removing from buffer) one piece of data at a time.

The problem is to make sure that producer won't try to add data in the buffer when it is full and the consumer won't try to remove data from the empty buffer.

DATE

Soution:

The soution of the producer is to either go to sleep or discard the data if the buffer is full. The next time the consumer removes an item from the buffer, it notifies the producer, & then producer starts to fill the buffer again.
In the same way, the consumer can go to sleep if it finds the buffer empty. The next time producer puts data into buffer, it wakes up the sleeping customer

* Sleeping Barber Problem

Scenario:

→ ~~Customers arrive to a barber~~ A barber has ~~a barber~~ shop ~~room~~ where he cuts the hair of customer and it has only one chair. And another waiting room, where he has many chairs ~~for~~ customers.

→ Customers arrive to the barber, if there are no customers the barber sleeps in his chair. If the barber is in sleep, the customers must wake him up

→ This analogy is based upon a hypothetical barber shop with one barber. The barber has one barber chair and a waiting room with a number of chairs in it.

→ When the barber finishes cutting a customer's hair, he dismisses the customer then goes to the waiting room to see if there is any customers waiting.

DATE

→ If there are, he brings one of them back to the chair and cuts his hair. If there are no customers waiting, he returns to his chair & sleeps in it.

→ Each customer, when he arrives, looks to see what the barber is doing. If barber is sleeping, then the customer wakes him up & sits in the chair. If the barber is cutting hair, then the customer goes to waiting room & sits in it & waits his turn. If there is no free chair, the customer leaves.

Problem:

1) A customer may arrive and observe that the barber is cutting hair, so he goes to the waiting room. At the same time, while he is on his way, the barber finishes cutting hair & goes to check the waiting room. Since, there is no one there, the barber goes back to his chair & sleeps. The barber is now waiting for customer and customer is waiting for barber.

2) Two customers may arrive at same time while barber is cutting hair. There happens to be a single seat in waiting room. They go to waiting room and attempt to occupy single chair.

DATE

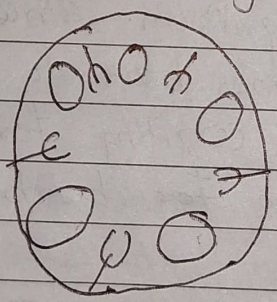
Solution:

There are many solutions available. The key element of each solution is using a mutex. The barber must acquire this mutex ~~exclusively~~ exclusion before checking for customers & release it when he begins either to sleep or cutting hair.

A customer must acquire it before entering the shop & release it once he sits in either waiting room or barber chair.
This will eliminate both problems discussed above.

* Dining Philosopher problem

In 1965, Dijkstra posed and solved a synchronizath problem called the dining philosophers problem.



Scenario:

→ There are N philosophers sitting around a circular table eating spaghetti and discussing philosophy.
→ Each philosophers needs 2 forks to eat and there are N forks.

DATE

The algorithm should be developed in such a way that philosophers can follow ensures that none starves as long as each philosopher eventually stops eating, & such that the maximum number of philosophers can eat at once.

- There are N philosophers sitting around a circular table.
- Philosophers can either eat or think
- Eating need 2 forks (one from right & one from left)
- Pick one fork at a time.
- How to prevent dead lock

The problem was designed to illustrate the problem of avoiding deadlock. One idea is to instruct each philosopher to behave as follows:

- Think until left fork is available; when it is, pick it up
- Think until right fork is available; " " "
- eat
- put the left fork down
- put the right fork down
- repeat from the start.

This solution is wrong. If all philosopher take their left fork simultaneously, none will be able to take their right forks and deadlock will occur.
We could modify the program so that after taking the left fork, the program checks to see if the right fork is available. If it's not, the philosopher puts down the left one, waits for some time, and then repeats the whole process.

DATE

* Process Scheduling

→ The part of operating system that makes the choice which process to run next whenever two or more processes are simultaneously in ready state is called the scheduler and the algorithm it uses is called the scheduling algorithm.

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

→ Process scheduling is an essential part of a multiprogramming operating system. Such OS allow more than one process to be loaded into the executable memory at a time & the loaded process shares the CPU using time multiplexing.

* Scheduling Goals & Objective.

Fairness: Each process gets fair share by CPU
Efficiency: When CPU is 100% busy then efficiency is increased
Response time: Minimize the response time for user
Troughput: Maximize jobs per given time period
Waiting time: Minimize total time spent waiting in queue

DATE

* Types of scheduling.

	Non preemptive scheduling	Preemptive scheduling
1.	In non preemptive, once resources are allocated to a process, the process holds it till it completes its burst time.	1. The resources are allocated to a process for a limited time.
2.	Process cannot be interrupted till it terminates.	2. Process can be interrupted in between
3.	If a process with long brust time is running cpu, then another process with less cpu burst time may starve.	3. If a high priority process frequently arrives in a queue, low priority process may starve.
4.	Non-preemptive scheduling doesn't have overheads.	4. Preemptive scheduling has overheads of scheduling the process.
5.	Non-preemptive scheduling is rigid.	5. Preemptive scheduling is flexible.

PAGE

classmate

DATE

* Characteristics of non-preemptive scheduling

- Picks a process to run until it release cpu
- Once a process has given cpu, it run until blocks for I/O or termination
- Treatment for all process is fair
- Response time are more predictable.
- Useful in real time system.
- No priority.

* Characteristics of preemptive scheduling.

- Picks a process and let it run for a maximum of fixed time.
- Process are allowed to run for a maximum of some fixed time.
- Useful in systems in which high priority process requires rapid attention.
- In time sharing system, preemptive scheduling is important in guaranteeing acceptable response.

DATE

* Scheduling algorithms

1) Batch system scheduling
   - First come first served
   - Shortest job first
   - Shortest remaining Time next

2) Interactive system scheduling
   - Round-Robin scheduling
   - Priority scheduling
   - Multiple queues

3) Real Time system scheduling (overview only)

BIR

* Batch system scheduling.

1. First come first serve.

- This is non-preemptive scheduling algorithm.
- It is the simplest scheduling algorithm in which jobs are scheduled in the order they are received
- Implementation of this technique is easy and is based on FIFO queue.
- It derives its concept from real life customer service

Characteristics:

- Jobs/process are schedule in order they are received
- Once the process has cpu, it runs to completion
- Fair to all processes.

Problem

- No guarantee of good response time (average waiting time is high).

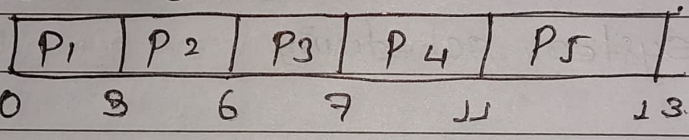
DATE

Ex: Consider 5 processes.

Process	Arrival Time	Processing / Burst Time / Execution
P_1	0	3 ms
P_2	2	3 ms
P_3	3	1 ms
P_4	5	4 ms
P_5	8	2 ms

Use FCFS scheduling algorithm to Draw Gantt chart. Compute average waiting time.

⇒ If process arrive as per the arrival time, the Gantt chart will be.



Average waiting time for:

$P_1 = 0$, $P_2 = 3$, $P_3 = 6$, $P_4 = 7$, $P_5 = 11$, $P_6 = 13$ ms

∴ Average waiting time: $\frac{0+3+6+7+11+13}{5} = 5.4$ ms.

Note: If all the processes arrive at the time 0, then the order of scheduling will be

P_3, P_5, P_1, P_2 & P_4.

classmate

PAGE

2. Shortest-Job First (SJF).

- This is also non-preemptive scheduling algorithm.
- It is the best approach to minimize waiting time
- It selects the waiting process with smallest execution time to execute next. This is the advantage, as short processes are handled very quickly.

Characteristics:

- The processing time is known in advanced.
- SJF selects process with shortest ~~waiting~~ processing time
- The design policies are based on CPU time.

Problem

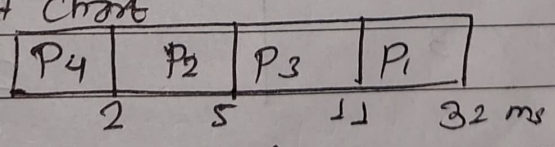
- Not applicable in timesharing system

Ex: Consider 4 process.

Process	Burst time (processing time)
P_1	21 ms
P_2	3 ms
P_3	6 ms
P_4	2 ms.

Use SJF scheduling algorithm to draw Gantt chart and find average waiting time.

=) Using Gantt chart



Average waiting time for

$P_4=0$, $P_2=2$, $P_3=5$ ms, $P_1=11$ ms

$\therefore$ Average waiting time: $\frac{0+2+5+11}{4} = 4.5$ ms.

DATE

3) Shortest Remaining Time Next. (SRTN)

- This is the preemptive scheduling algorithm.
- Here, process with the smallest amount of time remaining until completion is selected to execute.
- Shortest remaining time is advantageous because short process are handled very quickly.
- When a new process is added, the algorithm only compares the currently executing process with the new process, ignoring all the processes currently waiting to execute.

Advantages:

- Useful in time sharing system
- Low average waiting time

Disadvantages

- Little overhead than SJF
- Requires additional computation than SJF

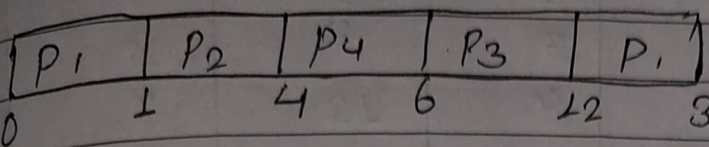
Ex: Consider 4 process:

Process	Arrival time	Burst/Processing time (ms)
P_1	0	21
P_2	1	3
P_3	2	6
P_4	3	2

Use SRTN scheduling algorithm to make the Grantt chart and find the average waiting time.

DATE

Let's make Gantt Chart



Process	Arrival time	Burst time	Completion time	Turnaround time.	waiting time
P_1	0	21	32	32	11
P_2	1	3	4	3	0
P_3	2	6	12	10	4
P_4	3	2	6	3	1

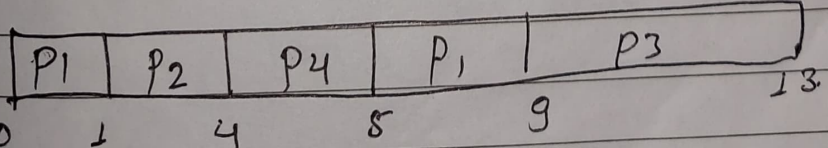
Note: TAT = CT - AT
WT = TAT - BT

∴ Average waiting time = $\frac{11+0+4+1}{4}$ = 4 ms.

Q.

Process	Arrival time	Burst time	Completion time
P_1	0	5	
P_2	1	3	
P_3	2	4	
P_4	4	1	

⇒ Let's make Gantt Chart



PAGE

classmate

Note: Response time: CPU 1st time - AT

Ex: for P3 = 9 - 2 = 7

DATE

Process	Arrival time	Burst time	Completion time	turn around time	waiting time
P1	0	5	9	9	4
P2	1	3	4	3	0
P3	2	4	13	11	7
P4	4	1	5	1	0

∴ Average waiting time = $\frac{4+0+7+0}{4}$ = 2.75 ms

Note: If asked: the time at which all the processes are completed is 13 ms.

DATE

* Interactive system scheduling

1) Round Robin scheduling

- In this algorithm, the process is allocated the cpu for specific time period called time slice.
- If the process completes its execution within this time slice, then it is removed from queue otherwise it has to wait for another time slice
- Pre-empted process is placed at the back of the ready list.

Advantage

- Fair allocation of cpu across the process
- Used in timesharing system.
- Low average waiting time when process length vary widely
- Poor average waiting time when process length are identical.

(Quantum size):- If the quantum is very large, each process is given as much time as needs for completion; If quantum is very small, system busy at just switching from one process to another process.

(optimal quantum size):-

Key idea: 80% of the CPU bursts should be shorter than the quantum. 20-50 msec reasonable for many general processes.

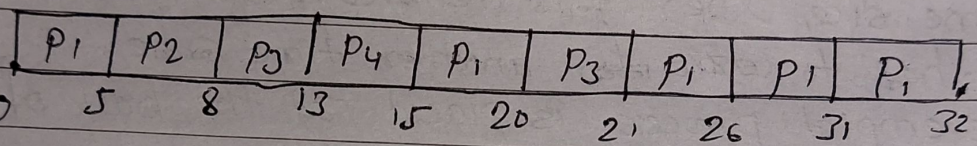
Ex: Consider following processes, arrives for execution in the same order, with arrival time 0, . Find average waiting time using the RRS.

DATE

Process	Burst Time
P_1	21
P_2	3
P_3	6
P_4	2

Quantum size = 5 ms

Let's make the Gantt Chart



P.ID	Arrival T.	BT	Completion T	TAT	WT
1	0	21	31	32	11
2	0	3	8	8	5
3	0	6	21	21	15
4	0	2	15	15	13

∴ Average Turn Around Time: $\frac{32+8+21+15}{4} = 19$ ms

∴ Average waiting time: $\frac{11+5+15+13}{4} = 11$ ms

DATE

2) Priority Scheduling

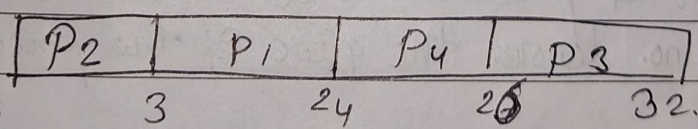
- In this scheduling algorithm the priority is assigned to all the processes & the processes with highest priority executed first.
- Priority ~~of~~ assignment of processes is done on the basis of internal factor such as CPU & memory requirements or external factor such as user's choice.
- It supports preemptive & non-preemptive scheduling policy
- The CPU is allocated with the highest priority process.

Problem: Starvation (low priority process may never execute)
Solution: Aging (as time progresses increase the priority of the process)

Ex: Consider following processes with arrival time 0, & given burst time. Find the average waiting time.

Process	Burst time.	Priority.
P_1	21	2
P_2	3	1
P_3	6	4
P_4	2	3

- Let's make Gantt Chart.



P.ID	A.T	BT	C.T	TAT	WT
1	0	21	24	24	3
2	0	3	3	3	0
3	0	6	32	32	26
4	0	2	26	26	24

∴ A.WT = A.TAT =

PAGE

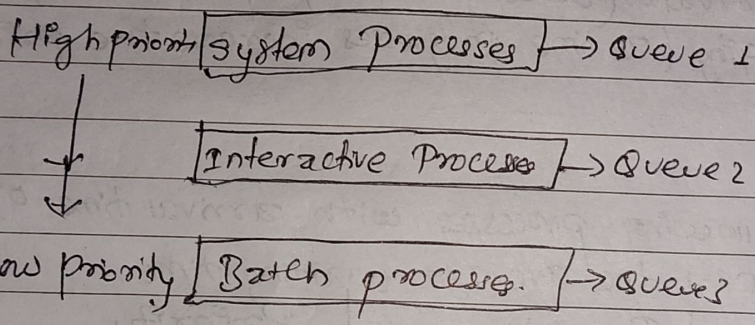
Thus, average waiting time = $\frac{0+3+24+26}{4} = \frac{53}{4}$

DATE

8) Multiple queues

→ The processes in queue can be divided in to different classes where each class has its own scheduling needs. Ex: A common division is a foreground (interactive) process & background (batch) process. These two classes have different scheduling needs.

Example, let's take 3 different types of process system processes, interactive processes & batch processes. All 3 processes have their own queue.



Here, all 3 different types of process have their own queue. Each queue has its own scheduling algorithm. Ex: queue 1 & queue 2 uses Round Robin while queue 3 uses FCFS to schedule their process.

Ex: Consider 4 processes under multilevel queue schedu Queue no. denotes the queue of the process.

Process	Arrival time	CPU Burst time	Queue number
P_1	0	4	1
P_2	0	3	1
P_3	0	8	2
P_4	10	5	1

priority of Queue 1 is greater than Queue 2. Queue 1 uses Round robin (Quantum size = 2ms) & Queue 2 uses FCFS

⇒ Let's make GANTT Chart:

P_1	P_2	P_1	P_2	P_3	P_4	P_4	P_4	P_3
0	2	4	6	7	10	12	14	15 20

PID	AT	BT	CT	TAT	WT
1	0	4	6	6	2
2	0	3	7	7	4
3	0	8	20	20	12
4	10	5	15	5	0

$= 38/4$ Ans

∴ A-TAT $= \frac{6+7+20+5}{4}$

A-WT $= \frac{2+4+12+0}{4}$

$= 18/4 = 4.5$

✗ Real Time System scheduling

→ A real time scheduling system is composed of scheduler, clock and the processing hardware elements.

→ In a real-time system, a process or task has schedulability; tasks are accepted by a real time system & completed as specified by the task deadline depending on the characteristic of the scheduling algorithm.

→ Modeling & evaluation of a real time scheduling system concern is the analyses of algorithm capability to meet a process deadline.

→ A task in a real time must be completed neither too early nor too late. A real time scheduling algorithm can be classified as static & dynamic.

→ A static scheduler determine task priority before System runs. A dynamic scheduler determines task ~~Priorities~~ priorities as it runs.

PAGE

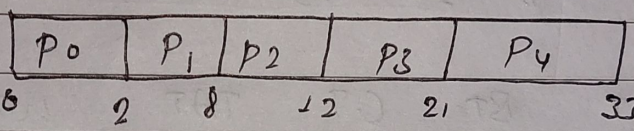
DATE

Some practice questions

Q. Given 5 processes. Find average waiting time and Turn Around Time for a process. using FCFS algorithm

P.ID	Burst time (ms)	Arrival (ms)
0	2	0
1	6	1
2	4	2
3	9	3
4	12	6

⇒ Let's make Gantt chart.



P.ID	Burst time	Arrival time	completion time	Turn Around Time	waiting time
0	2	0	2	2	0
1	6	1	8	7	1
2	4	2	12	10	6
3	9	3	21	18	9
4	12	6	33	27	15

∴ Average TAT = $\frac{2+7+10+18+27}{5}$ = 12.8 ms

∴ Average WT = $\frac{0+1+6+9+15}{5}$ = 6.2 ms

PAGE

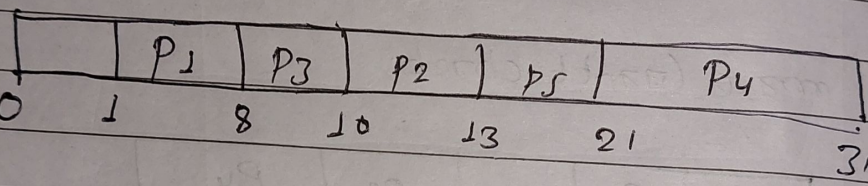
classmate

DATE

Q. Given 5 processes. Compute the average TAT & WT using SJF Algorithm

P.ID	Arrival time	Burst time
1	1	7
2	3	3
3	6	2
4	7	10
5	9	8

⇒ Let's make GANTT chart



P.ID	AT	BT	C.T	TAT	WT
1	1	7	8	7	0
2	3	3	13	10	7
3	6	2	10	4	2
4	7	10	31	24	14
5	9	8	21	12	4

∴ Average TAT = $\frac{7+10+4+24+12}{5} = 11.4$

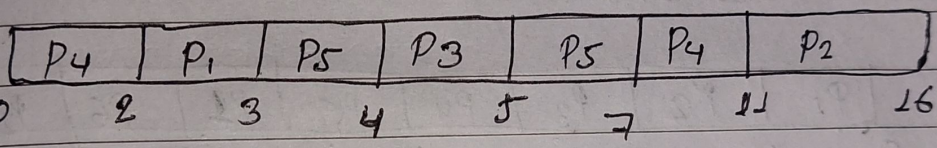
Average WT = $\frac{0+7+2+14+4}{5} = 5.4$

DATE

Q. Given 5 processes. Compute the average TAT & WT using SRTN algorithm

P.ID	AT	B.T
1	2	1
2	1	5
3	4	1
4	0	6
5	2	3

⇒ Let's make GANTT Chart.



P.ID	A.T	B.T	CT	TAT	WT
1	2	1	3	1	0
2	1	5	16	15	10
3	4	1	5	1	0
4	0	6	11	11	5
5	2	3	7	5	2

∴ Average TAT = $\frac{1+15+1+11+5}{5}$ = 6.6

∴ Average WT = $\frac{0+10+0+5+2}{5}$ = 3.4

PAGE

DATE

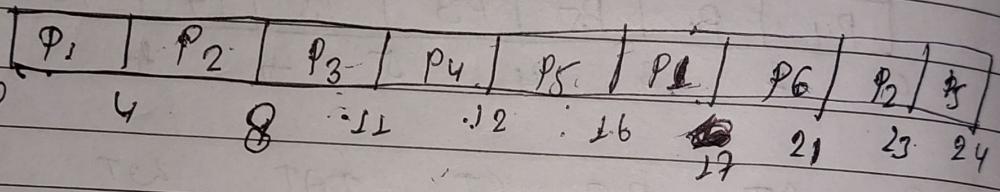
Q. Given 6 processes. find the average TAT and WT using Round Robin scheduling algorithm.

P.ID	Arrival time.	Burst time.
1	0	5
2	1	6
3	2	3
4	3	1
5	4	5
6	6	4

Quantum size = 4 ms

Ready queue: P1 P2 P3 P4 P5 P1 P6 P2 P5

⇒ Let's make GANTT Chart



P.ID	AT	BT	CT	TAT	WT
1	0	5	17	17	12
2	1	6	23	22	16
3	2	3	11	9	6
4	3	1	12	9	8
5	4	5	24	20	15
6	6	4	21	15	11

∴ Average TAT = $\frac{17+22+9+9+20+15}{6}$ = 15.833 ms

∴ Average WT = $\frac{12+16+6+8+15+11}{6}$ = 11.833 ms

PAGE

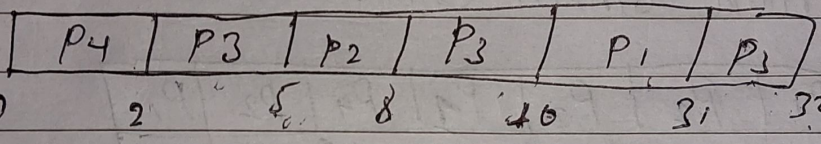
Note: If arrival time given, then priority scheduling is preemptive.

DATE

Q. Given 4 processes find the average TAT & WT using priority scheduling algorithm.

P.ID	Arrival time.	B.T	Priority.
1	10	21	2
2	5	3	1
3	0	6	4
4	0	2	3

⇒ Let's make Gantt Chart.



P.ID	A.T	B.T	CT	TAT (=CT-AT)	WT (=TAT-BT)
1	10	21	31	21	0
2	5	3	8	3	0
3	0	6	32	32	26
4	0	2	2	2	0

$$\text{Average WT} = \frac{26}{4}$$

$$\text{TAT} = \frac{21+3+32+2}{4} = \frac{58}{4} =$$

DATE

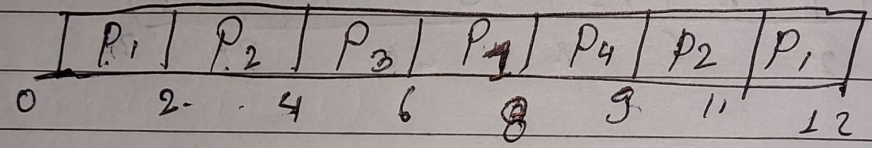
Q. Given, 4 process, find average W.T & T.A.T using RR

Process no.	Arrival time	Burst Time.
P_1	0	5
P_2	1	4
P_3	2	2
P_4	4	1

Quantum time = 2

→ Ready queue. $P_1 | P_2 | P_3 | P_1 | P_4 | P_2 | P_1$

Let's make GANTT chart.



P.no	AT	BT	CT	TAT	WT	Response time = 1st time - AT
P_1	0	5	12	12	7	0
P_2	1	4	11	10	6	1
P_3	2	2	6	4	2	2
P_4	4	1	9	5	4	4

Avg WT = WT sum / 4

Avg TAT = TAT sum / 4

If asked, no. of context switching = 5

Q. Given, 4 process. Find average WT & TAT using priority scheduling. [Given higher the no. higher the priority]

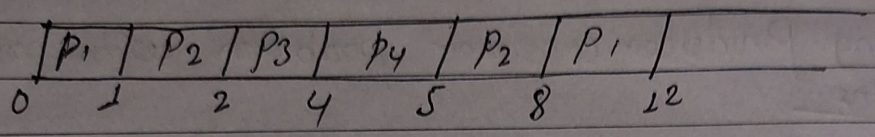
P.ID	Priority	A.T	B.T
P_1	10	0	5
P_2	20	1	4
P_3	30	2	2
P_4	40	4	1

Classmate

PAGE

DATE

Let's make GANTT CHART



P.ID	AT	B.T	CT	TAT	WT
P_1	0	5	12	12	7
P_2	1	4	8	7	3
P_3	2	2	4	2	0
P_4	4	1	5	1	0

Average TAT = $\frac{12+7+2+1}{4} = 5.5$, Average W.T = $\frac{7+3+0+0}{4} = 2.5$

PAGE

2076. (10 marks)

DATE

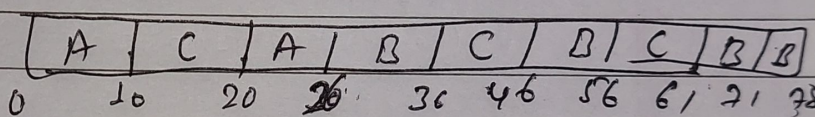
Q. Consider the following process data and compute average waiting time and average turn around time for RR (quantom 10) and priority scheduling algorithm.

P.ID	Burst time	Arrival time	Priority
A	16	0	1
B	37	12	2
C	25	7	3

⇒ 1st for RR.

Ready queue: A | C | A | B | C | B | C | B | B

Grantt Chart:

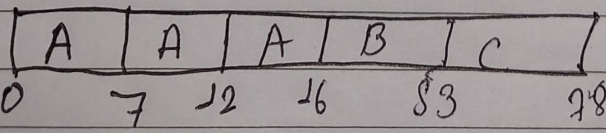


P.ID	Arrival Time	Burst time	Completion Time.	Turn around time.	Waiting time
A	0	16	26	26	10
B	12	37	78	66	29
C	7	25	61	54	29

∴ Average Turn Around time = $\frac{26+66+54}{3} = 48.66$

Average waiting time = $\frac{10+29+29}{3} = 22.66$

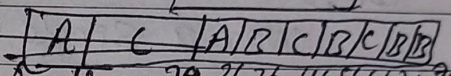
⇒ 2nd for Priority Scheduling

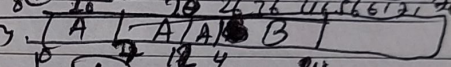


Questions asked from this chapter

Q. What kind of problem arises with sleep and wakeup mechanism of achieving mutual exclusion? Explain with suitable code snippet. (2078 - 10 marks)
⇒ Explain producer-consumer problem

Q. Define interactive system goals? List various interactive scheduling algorithms. Consider following process data and compute average waiting time and average turn around time for RR (quantum 10) and priority scheduling algorithms. (2076 / 10 marks)

~~for RR~~ 

~~for priority~~ 

PID	Burst Time	Arrival time	Priority
A	16	0	1
B	37	12	2
C	25	7	3

Q. Why threads are better than processes when? Explain the concept of user level threads in detail. (2076 - 5 marks)

Q. Define critical section problem? Describe the criteria to be satisfied for solving this problem? (2075 - 5 marks)

Q What is problem associated with semaphores? Explain the concept of monitors in brief. (2074 - 5 marks)

Q Are there any linkage between semaphore & deadlock condition? If yes, explain with example. (2074 - 5 marks)

Q. How producer consumer problem can be solved with sleep & wakeup primitives? Explain (2075 - 5 marks)

DATE

Q. Do you think a process can exist without any state? Justify with the help of process state diagram (2074 - 5 marks)

Q. Explain 'race condition' and also state how process synchronization is handled using Semaphore? Explain with algorithms. (2074 - 10 marks)

Q. Describe how multithreading improves performance over a single threaded solution. (2070, 2072 - 5 marks) (2069, 5 marks)

Q. Differentiate between I/O bound process & CPU bound process. (2066 - 5 marks)

Q. Differentiate between busy waiting & Blocking? (2075 - 5 marks)

Q. Differentiate between race conditions & critical regions. (2075 - 5 marks)

Q. Differentiate between program & process (Imp)

Q. Five batch jobs A through E, arrive at a computer center at almost same time. They have estimated running times of 10, 8, 4, 2 and 6. Their priorities are 3, 5, 2, 1 & 4 respectively with 5 being the highest priority. Determine average turn around time & waiting time for. (2075 - 10 marks)

a) Round Robin (Quantum = 2)
b) Priority scheduling
c) SJF

classmate

PAGE

DATE

Q=) ans

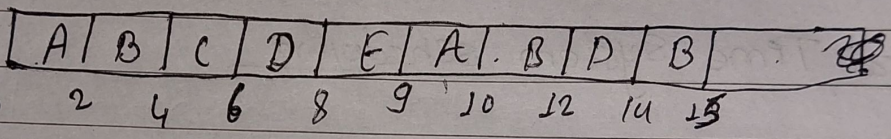
Batch Jobs	Burst time	Priority
A	10	3
B	8	5
C	4	2
D	2	4
E	6	1

5 has higher priority

Average turn around time & waiting time for

a) Round Robin (Quantum=2)

Let's make GANTT chart.

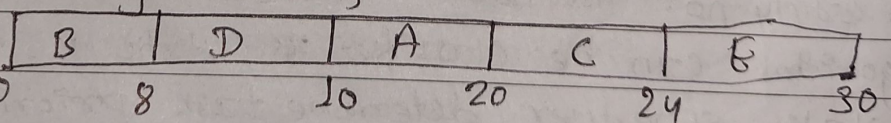


Batch Jobs	Burst time	Completion Time	Turn Around Time	waiting time
A	10	10	10	0
B	8	15	15	7
C	4	6	6	2
D	2	14	14	12
E	6	9	9	3

∴ Average Turn around time = $\frac{10+15+6+14+9}{5} = 10.8$

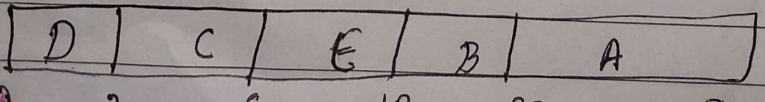
∴ Average waiting time = $\frac{0+7+2+12+3}{5} = 4.8$

b) Priority scheduling.



calculate ATAT & AWT

c) Shortest Job first



PAGE

2076. (10 marks)

DATE

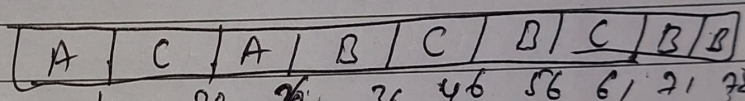
Q. Consider the following process data and compute average waiting time and average turn around time for RR (quantom 10) and priority scheduling algorithm.

P.ID	Burst time	Arrival time	Priority
A	16	0	1
B	37	12	2
C	25	7	3

⇒ 1st for RR.

Ready queue: A | C | A | B | C | B | C | B | B

Grantt Chart:

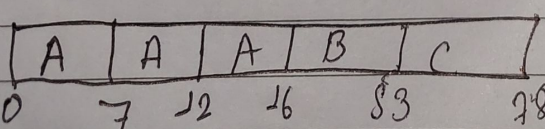


P.ID	Arrival Time	Burst time	Completion Time.	Turn around time.	Waiting time.
A	0	16	26	26	10
B	12	37	78	66	29
C	7	25	61	54	29

∴ Average Turn Around time = $\frac{26+66+54}{3} = 48.66$

Average waiting time = $\frac{10+29+29}{3} = 22.66$

⇒ 2nd for Priority Schedulin



PID	Arrival Time	Burst Time	Completion Time	Turn Around Time	Waiting Time
A	0	16	16	16	0
B	12	37	53	41	4
C	7	25	78	71	46

∴ Average Turn Around Time = $\frac{16+41+71}{3}$ = 42.66

Average Waiting Time = $\frac{0+4+46}{3}$ = 16.66

Q. What kind of problem arises with sleep & wake up mechanism in achieving mutual exclusion? Explain with a solution of [Let's take an ex. of producer consumer] code. 2078-10 marks

DATE

⇒ * Producer consumer problem (sleep & wake up mechanism)

```
#define BUFFER_SIZE 100
int itemCount=0;
void producer()
{
while(true)
{
item = produceItem();
if (itemCount == BUFFER_SIZE)
{ sleep();
}
putItemIntoBuffer(item);
itemCount = itemCount+1;
if (itemCount==1)
{ wakeup(consumer); }
}
}
void consumer()
{
while(true)
{
if(itemCount==0)
{
sleep(); }
item= removeItemFromBuffer();
itemCount = itemCount-1;
if (itemCount== BUFFER_SIZE-1)
{ wakeup(producer); }
consumeItem(item);
}
}
```

⇒ * problems that arise with sleep & wake up mechanism in producer consumer problem while having mutual exclusion are:-

The problem with sleep & wake up mechanism is that it contains a race condition that can lead to a deadlock: consider following scenario.

1. The consumer has just read the variable item Count, noticed its zero and just about to move inside the if block.
2. Just before calling sleep, the consumer is interrupted & the producer is resumed
3. The producer creates an item, puts it into the buffer, & increases item count.
4. Because the buffer was empty prior to the last addition, the producer tries to wake up the consumer.
5. Unfortunately, the consumer wasn't yet sleeping, & the wakeup call is lost. When the consumer resumes, it goes to sleep & will never be awakened again. This is because the consumer is only awakened by the producer when item count is equal to 1.
6. The producer will loop until the buffer is full, after which it will also go to sleep.
7. Since both process will sleep forever, we have run into dead lock. This solution/mechanism therefore is unsatisfactory.