27-2-2021

M. Maheeth Reddy
1801CS31

CS341 - MidSem Assignment

Ans 1:

Operating System (OS):

An operating system is a software that manages a computer's hardware. It also provides a basis for application programs and acts as intermediary between the computer user and the computer hardware

History of Operating Systems:

The first computers did not have an operating system. Programmers had to manually program the computer for performing a particular task. During its initial years, computers were only used for simple mathematical calculations and hence an OS wasn't needed.

As years went by, the complexity of calculations increased. In response to this problem, development of system software to facilitate writing and execution of programs began. The foundations for operating systems' concepts began here.

The first OS was created in early 1950's by General Motors for their IBM 701. It was called GMOS.

The second-generation OS was based on a single stream of batch processing system because it collects all similar jobs in groups or batches and then submits the jobs to the operating system using a punch card to complete all jobs in a machine. At each completion of jobs (either normally or abnormally), control transfer to the operating system that is cleaned after completing one job and then continues to read and initiates the next job in a punch card. After that, huge machines called mainframes which were used by professional operators.

The third generation OS was capable of multiprogramming which meant multiple tasks in a single computer program could be performed. Also, the addition of overlap I/O, interrupts and direct memory access for I/O devices, improved the performance and responsiveness of computer.

The introduction of multiprogramming plays a very important role in developing operating systems that allow a CPU to be busy every time by performing different tasks on a computer at the same time.

All the above meintioned developments took place as part of Phase 1 of OS development. In the Phase 2, interactive time-sharing features were incorporated into the OS'es, so that, through the use of cheap terminals, multiple users interact with the system at the same time. OS gives an illusion as if each user has their own computer. This is achieved by sacrificing CPU Time to get better response time for users.

In the next phase of OS development, hardware became cheaper and humans became expensive. ~~for jobs.~~ Since, computers came cheaply, everyone could now afford a computer. This was the rise of Personal Computing. Finally in phase 4, the idea of distributed system came to the fore since people could afford multiple PC's and wanted to manage them. This co-ordination b/n PC's could be done by allowing

geographically distributed machines to interact, share resources and data easily.

Key Features of Classes of Operating Systems

OS Class	Period	Prime Concern	Key concepts
Batch Processing	1960s	CPU idle time	Automate transition between jobs
Multi-programming	1960s	Resource utilization	Program priorities, pre-emption
Time-sharing	1970s	Good response time	Time slice, Round-robin scheduling
Real Time	1980s	Meeting time constraints	Real-time scheduling
Distributed	1990s	Resource Sharing	Distributed control, transparency.

MULTICS and UNIX

MULTICS was a time-sharing OS developed b/n 1965 and 1972 at MIT. It was used for running continuously on a large complex mainframe computer.

with a vast file system of shared programs and data.

In 1969, Ken Thompson and Dennis Ritchie of Bell Labs began to work on UNIX, as an OS for minicomputers. Ritchie, having worked on MULTICS and knowing its complexity, simplified MULTICS to create UNIX. UNIX was written in C language. The size, simplicity and clean design of the UNIX system encouraged programmers to experiment with UNIX development. This rose to the development of 4BSD which was fundamental to development of internet. UNIX could run on mainframes, workstations, minicomputers, supercomputers and personal computers.

GUI in OS:

Apple was the first company to provide GUI to OS'es. The Macintosh released in January 1984 had a GUI-style OS which allowed users to interact with the computer through click buttons, pull-down menus and other image options on the screen.

Later Microsoft also incorporated the GUI into its Windows 1.0 to compete with Apple. And the GUI has been evolving ever since.

Concepts of OS:

Operating Systems went through various phases of development since their inception as mentioned previously. ~~An OS is~~ To write an OS, the following concepts must be in mind:

① Processes and Process Management:

A process is basically an instance of a program. It is executed ~~as a~~ sequentially. When a ~~program~~ program is loaded into memory, it becomes a process. The memory can be divided into 4 sections – stack, heap, text, data.

Stack – contains temporary data (Local variables, function parameters, return address)

Heap – dynamically allocated memory

Text – Program Counter & contents of processor registers.

Data – Global & Static Variables

A process can have different states during execution – Start, Ready, Running, Waiting, Exit.

All info related to a process is stored in Process Control Block (PCB)

② Threads and Concurrency

Threads are lightweight processes which are used to improve application performance through parallelism.

~~Threads represent a software appro~~

They reduce overhead equivalent to a classical process

③ Scheduling

It is an essential part of multiprogramming systems.

It is used for deciding which process in the ready queue must be given the CPU's core for execution

It aims to ~~minize~~ minimize CPU idle time.

④ Memory Management

It is the functionality of an OS which handles or manages primary memory and moves processes back and forth b/n main memory and disk during execution.

It decides which process will get memory at what time.

⑤ Inter-Process Communication (IPC):

It is a mechanism that allows processes to communicate to each other and synchronize their actions.

Communication is done by: Shared Memory or Message Parsing.

(6) I/O Management:

It is an important feature of an OS, to manage various I/O Devices among application I/O ~~devices~~ requests, sending respones from I/O devices back to applications. This is done through: I/O Mapped I/O instructions,
Memory mapped I/O instructions,
Direct Memory Access (DMA)

(7) Virtualization:

It is a technology that allows us to ~~create multiple~~ abstract the hardware of a single computer into serveral different execution environments, creating the illusion that each seperate environment runs on its own private computer.

(8) Distributed File Systems:

It is a client/server based application that allows clients to access and process data stored on the server as if it were their own computer.

Ans 2 Classical Problems of Synchronization

Producer - Consumer Problem / Bounded-Buffer Problem:

The producer consumer problem illustrates the need of synchronization in systems where many processes share a same resource.

In this problem, two processes share a fixed-size buffer. One process produces information and puts it in the buffer, while the other process consumes information from the buffer. (Hence, the name producer-consumer). These processes access the buffer concurrently.

So, to synchronize these processes, we have to block the producer when buffer is full and consumer when the buffer is empty.

So, producer and consumer should work as follows:

Producer

① Producer will create a new item.

② If buffer is full,
producer goes to sleep until
consumer sends a "wakeup call" if buffer is empty.

③ Producer will put new item in the buffer.
If producer went to sleep in step ②, it won't wake up until buffer is empty. So, buffer will never overflow.

④ Then, the producer checks to see if buffer is empty.
If it is, producer assumes ~~buffer~~ consumer is sleeping, so it will wake the consumer. ~~Keep in mind~~ An interrupt b/n any of these steps allows consumer to run.

Consumer:

① Consumer checks to see if buffer is empty. If so, consumer puts itself to sleep until producer wakes it up, if it finds the buffer empty after it puts an item into the buffer.

② Then, the consumer will remove an item from the buffer. The consumer will never try to remove a widget from an empty buffer because it will not wake up until the buffer is full.

③ If the buffer was full before it removed the ~~widget~~ item, the consumer will wake the producer.

④ Finally, the consumer will consume the widget.

An interrupt b/n any of these steps allows producer to run.

Solution using Semaphores:

Semaphores simplify implementation of Sleep and Wakeup policy. In this problem, semaphores are used for mutual exclusion and synchronization

In this solution, three semaphores have been used:

full → to count filled slots
empty → empty slots
mutex → to enforce mutual exclusion.

~~BufferSize = 3;~~

~~Semaphore~~

Solution:

```
Semaphore mutex = 1;
Semaphore empty = BufferSize;
Semaphore full = 0;
```

// Producer code:

```
Producer() {
    int item;
    while(TRUE) {                    → loop forever
        make_new(item);
        down(&empty);                → decrement empty [semaphore]
        down(&mutex);                → decrement mutex [enter critical section]
        put_item(item);
        up(&mutex);                  → leave critical section [increment mutex]
        up(&full);                   → increment full semaphore
    }
}
```

// Consumer code:

```
Consumer() {
    int item;
    while (TRUE) {                   → loop forever
        down(&full);                 → decrement full
        down(&mutex);                → enter critical section
        remove_item(item);
        up(&mutex);                  → leave critical section
        up(&empty);                  → increment empty.
        consume_item(item);
    }
}
```

Dining – Philosophers Problem:

There are 5 philosophers sharing a circular table and they eat and think alternatively. There is a bowl of rice for each of the philosophers and 5 chopsticks. A philosopher needs both their right and left chopstick to eat. A hungry philosopher may only eat if there are both chopsticks available. Otherwise a philosopher puts down their chopstick and begins thinking again.

This is a classic synchronization problem as it demonstrates a large class of concurrency control problems.

Solution using Semaphore:

To represent the chopsticks, semaphores are used. A chopstick can be picked up by executing a wait operation on the semaphore and released by executing a signal semaphore.

Code for philosopher "i"

```
semaphore chopsticks[5];

do {
   wait(chopsticks[i]);                    → left chopstick
   wait(chopsticks[(i+1)%5]);              → right chopstick.

   eat();
   signal(chopsticks[i]);
   signal(chopsticks[(i+1)%5]);

   think();
} while(TRUE);
```

Steps:
1. Wait operation is performed on chopsticks[i] and chopsticks[(i+1)%5]
   ↳ Philosopher has picked up chopsticks on his sides.
2. Philosopher eats
3. Signal operation is performed on chopsticks[i] and chopsticks[(i+1)%5]
   ↳ Philosopher has eaten and put down chopsticks.
4. Philosopher thinks.

This solution can lead to deadlock when all philosophers pick their left chopstick simultaneously.

To avoid deadlock:

1. Almost four philosophers
2. Even no. philosopher picks right chopstick and then left. Reverse for odd no. philosopher.
3. Philosopher is allowed to pick bothe chopsticks if they are available at same time

Readers and Writers Problem:

The readers and writers problem is useful to model processes that are competing for a limited shared resource. A practical example is a ticket reservation system consisting of a huge database with many processes that read and write the data. Reading information from the database will not cause a problem since no data is changed. The problem lies in writing information to the database. If no constraints are put on access to the database, data may change at any moment. By the time a reading process displays the result of a request for information to the user, the actual data

in the database may have changed. For instance, a process reads the no. of available seats on a flight to be one and reports it to the customer. Before the customer has a chance to make their reservation, another process makes a reservation for another customer, changing the no. of available seats to zero.

Solution using Semaphores

Semaphores can be used to restrict access to the database under certain conditions. In this problem, semaphores are used to prevent any writing processes from changing information in the database while other processes are reading from the database.

Code

```
semaphore mutex = 1;
semaphore db    = 1;
int reader_count;
Reader() {
    while
```

Code

```
semaphore mutex = 1;   ~~> control access to reader count
                       [Access if 0]
semaphore db = 1;      ---> control access to database
int reader_count;      ---> no. of processes accessing data.

Reader() {
    while(true) {      ---> loop forever
        down(&mutex);  ---> gain access to reader_count
        reader_count++; ---> increase reader_count
        if(reader_count == 1)   } if this is the first process
            down(&db);          } to read database, this will
                                  prevent access by writing process
        up(&mutex);    ---> allow other process to access reader_count
        read_db();
        down(&mutex);
        reader_count--;
        if(reader_count == 0)
            up(&db);
        up(&mutex);
}
```

```
Writer() {
    while(true){
        create_data(); → create data to enter into database.
        down(&db);
        write_db(); → write info to database.
        up(&db);
    }
}
```

(P.T.O)

Sleeping Barber Problem

This is another classical synchronization problem. The barber shop has one barber, a barber chair and n chairs for waiting customers. If there are no customers, the barber sleeps in the barber chair and a new customer must wake him up. Customers who arrive when the barber is at work, either wait or leave (if all chairs are full)

↳ (if there are empty chairs)

We have to program the barber and customers without race conditions.

Solution using Semaphores:

Three semaphores are used:

customers → to count waiting customers

barbers → whether barber is idle or not.

mutex → for mutual exclusion.

A variable waiting is also used to count waiting customers.

The reason for having waiting is there is no way to directly read a semaphore.

Code:

```
semaphore customers = 0;
semaphore barbers = 0;
semaphore mutex = 1;
int waiting = 0;

barber() {
    while(true) {
        down(&customers);   // go to sleep if no. of customers is 0
        down(&mutex);       // acquire access to waiting
        waiting--;          // decrement count of waiting customers
        up(&barbers);       // one barber is ready to cut hair
        up(&mutex);         // release waiting
        cut_hair();         // cut hair
    }
}

customer() {
    down(&mutex);           // enter critical region
    if (waiting < CHAIRS) { // leave if no free chairs
        waiting++;
        up(&customers);     // wake up barber if necessary
        up(&mutex);         // release access to waiting.
        down(&barbers);     // go to sleep if no free barbers
        get_haircut();
    } else { up(&mutex); }  // shop is full, do not wait
}
```

Note:
customer doesn't have an infinite while loop as he/she requires only one haircut

Ans 3: Monitor based Solutions for classical problems of synchronization

Producer-Consumer Problem:

Monitors make solving the producer-consumer problem easier. Mutual exclusion is achieved by placing critical section of the program inside a monitor. Once inside the monitor, a process is blocked by Wait & Signal primitives if it can't continue.

```
monitor ProducerConsumer
    condition full, empty;
    int count;

procedure enter() {
    if (count == N) wait(full);  // if buffer is full, block
    put_item(item);
    count++;                     // increment no. of full slots
    if(count == 1) signal(empty); // if buffer is empty,
                                  wake consumer.
}
```

```
procedure remove() {
    if (count==0) wait(empty);        → if buffer is empty, block
    remove_item(item);
    count--;                          → decrement count of full slots
    if (count == N-1) signal(full);   → if buffer is full, wake producer.
}

count=0;
end monitor;

Producer() {
    while (TRUE) {
        make_item(item);
        ProducerConsumer.enter;       → call enter function in monitor
    }
}

Consumer() {
    while (TRUE) {
        ProducerConsumer.remove;      → call remove function in monitor.
        consume_item(item);
    }
}
```

Dining Philosophers Problem

A deadlock-free solution using monitors. This solution imposes the restriction that a philosopher may pick up their chopsticks only if both are available.

Code:

```
monitor DiningPhilosophers {
    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 left & right neighbours
        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;
    }
}
```

Philosopher i can set the state[i] = EATING only if their two neighbours are not eating.

condition self[5]; allows philosopher i to delay themselves when hungry but unable to get chopsticks.

Readers and Writers Problem:

Monitors can be used to restrict access to the database. Read and Write functions used by processes which access the database are in a monitor called ReadersWriters. If a process wants to write to the database, it must call the writeDatabase function, and readDatabase function for reading.

Code:

```
monitor ReadersWriters
    condition OKtoWrite, OKtoRead;
    int ReaderCount = 0;
    Boolean busy = false;

procedure StartRead() {
    if (busy)                          block, if database is not free
        OKtoRead.wait;
    ReaderCount++;
    OKtoRead.Signal();
}

procedure EndRead() {
    ReaderCount--;
    if (ReaderCount == 0)
        OKtoWrite.signal();
}

procedure StartWrite() {
    if(busy || ReaderCount != 0) OKtoWrite.wait();
    busy = true;
}
```

```
procedure EndWrite() {
    busy = false;
    if(OKtoRead.Queue) OKtoRead.signal();
    else OKtoWrite.Signal();
}

Reader() {
  while(true) {
      ReadersWriters.StartRead();
      readDatabase(); ——→ call this function in monitor
      ReadersWriters.EndRead();
  }
}

Writer() {
  while(true) {
      make_data(&info);
      ReadersWriters.StartWrite();
      writeDatabase(); ——→ call this function in monitor
      ReadersWriters.EndWrite();
  }
}
```

Sleeping Barber Problem:

monitor SleepingBarber is the monitor inside which the barber & customer processes interact.

```
monitor SleepingBarber
    int numCustomer = 0;
    condition barber;
    condition customer;

procedure getCustomer() {
    if(numCustomer == 0)
        barber.wait;        ~~> wait for customers
    numCustomer--;
    customer.signal;        ~~> inform customer that
                                barber is ready
}

procedure haircut() {
    numCustomer++;
    barber.signal;          ~~> inform barber that
                                customer is in.
    customer.wait;          ~~> wait for barber.
    do_haircut();
}
```

```
Barber() {
    while(true) {
        SleepingBarber.getCustomer();
        cut_hair();
    }
}

Customer() {
    hair_cut();
}
```

Ans 4 CPU Scheduling Algorithms – Pros and Cons

First Come First Serve (FCFS):

In this algorithm, the process that requests the CPU first is allocated the CPU first. It is implemented using a First-In-First-Out (FIFO) queue. It is non-preemptive scheduling algorithm.

Pros:

The algorithm is simple and easy to implement.

Cons:

① The average waiting time is quite long in CPU scheduling.

② There may be ~~an overlap~~ a poor overlap of I/O and CPU since, CPU-bound processes will force I/O-bound processes to wait for the CPU, leaving I/O devices idle.

This causes lower CPU and device utilization than possible if shorter processes are allowed to go first.

③ Not useful for interactive systems as it is possible for one process keeping CPU for itself for a long period, which is not desired in those system.

Shortest Job First Scheduling (SJF):

This algorithm associates with each process the length of the process' next CPU burst. When, the CPU is available, it is assigned to the process that has the smallest next CPU burst & FCFS is used as a tie-breaker.

Pros:

1. Shortest jobs are favored over longer ones. It gives minimum-waiting time for a given set of processes by reducing waiting for shorter ones and the opposite for longer ones.

Cons:

1. Starvation can happen if shorter processes keep coming. This is solved by Aging i.e., by increasing the priority of the starving process.
2. It can't be implemented at the level of CPU scheduling in short-term as there is no way to know the length of next CPU burst. Hence, it is frequently used for long term scheduling.

Round-Robin Scheduling (RR)

This is similar to FCFS but preemption is enabled so that system can switch between processes. A small unit of time called a time quantum or time slice is defined. The ready queue is treated as circular queue. The CPU scheduler goes around the ready queue, allocating CPU to each process for a time interval of 1 quantum.

Pros:

① Every process gets an equal share of the CPU.

② Starvation is avoided since Round Robin scheduling is cyclic in nature.

Cons:

① In general, average waiting time under Round Robin is often long.

② The performance is dependent on quantum time.

→ If the quantum is too less,
Overhead increases and
CPU efficiency decreases.

→ If the quantum is too high,
Response is poor for short processes.

Priority Scheduling:

In this algorithm, each process is given a priority (for e.g., 0, 1, 2, 3, ---). Smaller value have more priority in general. CPU is allocated to the highest priority. FCFS is used as a tie-breaker. There is a pre-emptive and non-preemptive variant of priority scheduling.

Pros:

① High priority process need not wait for long because processes are executed as per priority.

② Suitable for applications with fluctuating time and resource requirements.

Cons:

① If high priority processes use up a lot of CPU time low priority processes may starve.

② Assigning priorities to processes is a huge challenge.

③ If the system crashes, all low priority processes get lost.

Aging is used to avoid the indefinite blockage of low priority processes.

Multilevel Queue Scheduling

Multilevel Queue scheduling partitions the ready queue into several seperate queues. Processes are assigned to queues, based on some memory size, priority, process type etc. Each queue has its own scheduling algorithm. ~~eg.~~

~~Advanta~~ Pros:

① Partitions processes into several seperate queues based on the process type [response-time requirements], so relevant scheduling algorithm can be used.

Eg:
- System Processes uses FCFS,
- Interactive Processes uses SJF,
- Batch Processes uses RR,
- Student Processes uses. Priority

high ↓ Priority ↓ low

Each queue has absolute priority over lower ones.

Cons:

① Processes at the lowest level face starvation.

Time-slicing among the queues can avoid this to an extent.

Multi-Level Feedback Queue Scheduling (MLFQ):

In multilevel queue scheduling, processes in the lowest priority queue can be starved as the tend to stay have lower priority. In multi-level feedback queues, processes can switch queues and change priorities. Priorities of processes change bases on CPU time, CPU/I/O boundedness, burst times, waiting times etc by aging.

Pros:

1. There is more flexibility for priorities
2. Processes can move between queues.
3. Starvation is prevented through aging.

Cons:

1. It is a complex algorithm
2. It generates more CPU overhead.
3. For selection of best scheduler, it requires other means to select values.

Comparision Table

Algorithm	Preemptive (Yes/No)	Complexity	Allocation	Waiting Time (Average)
FCFS	No	Simple	First Come First Serve	Large
SJF	Yes/No (2 variants present)	Slightly Complex	Least Burst Time runs first	Smaller than FCFS
Priority	Yes	Complex	Higher priority runs first	Smaller than FCFS
Round Robin	Yes	Complex	Fixed Time Slices, Order of Arrival	Larger than all above
MLQ	Yes	Complex	Higher priority queue first	Smaller than FCFS
MLFQ	Yes	Complex	Higher priority queue first	Smaller than FCFS

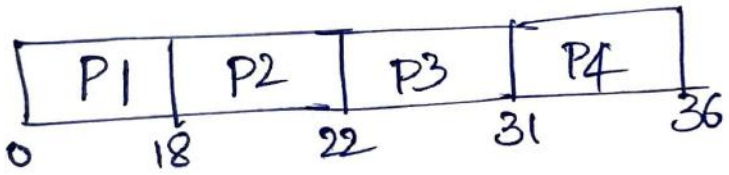
Comparision of FCFS, SJF, RR scheduling

Consider the following processes with the arrival & burst times (in ms).

Process	Arrival	Burst
P1	0	18
P2	1	4
P3	2	9
P4	3	5

With FCFS scheduling.

Gantt Chart:



The waiting times for P1 is 0ms
P2 is 17 ms
P3 is 21ms
P4 is 28 ms

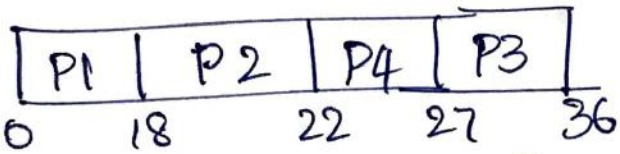
So, average waiting time = 16.25 ms

The turnaround time for P1 is 18ms
P2 is 21 ms
P3 is 29 ms
P4 is 33ms

Hence, average turnaround time is 25.25ms.

With SJF Scheduling,

Gantt Chart:

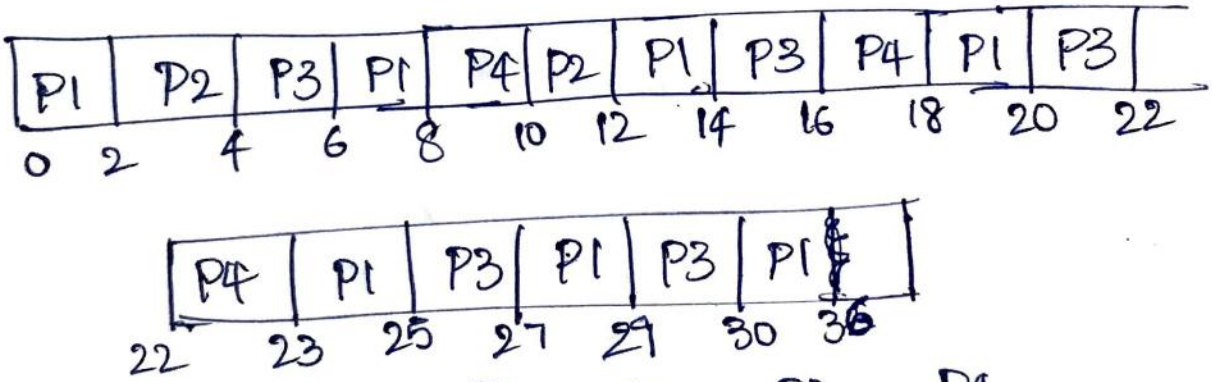


	P1	P2	P3	P4
Waiting Times	0	17	25	19
Turnaround Times	18	21	34	24

Hence, average waiting time is 15.25 ms
avg. turnaround time is 24.25 ms

With RR scheduling, (Time Quantum = 2)

Gantt Chart:



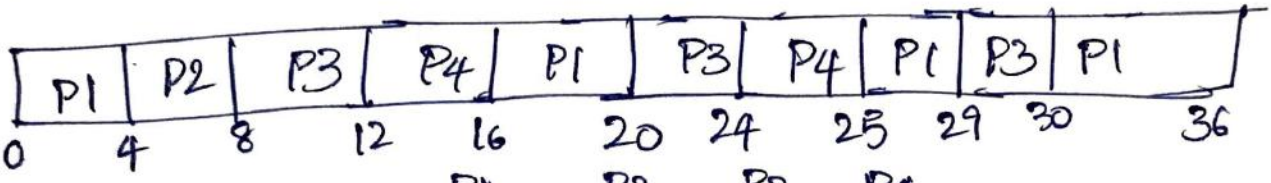
	P1	P2	P3	P4
Waiting Times:	18	7	19	15
Turnaround times:	36	11	28	20

Avg Waiting Time = 14.75 ms

Avg Turnaround Time = 23.75 ms.

With RR scheduling (TQ = 4)

Gantt Chart:

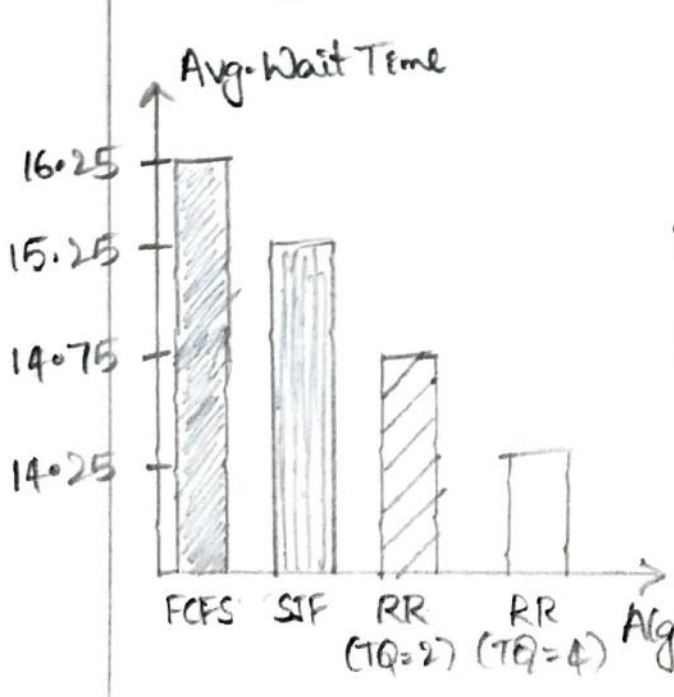


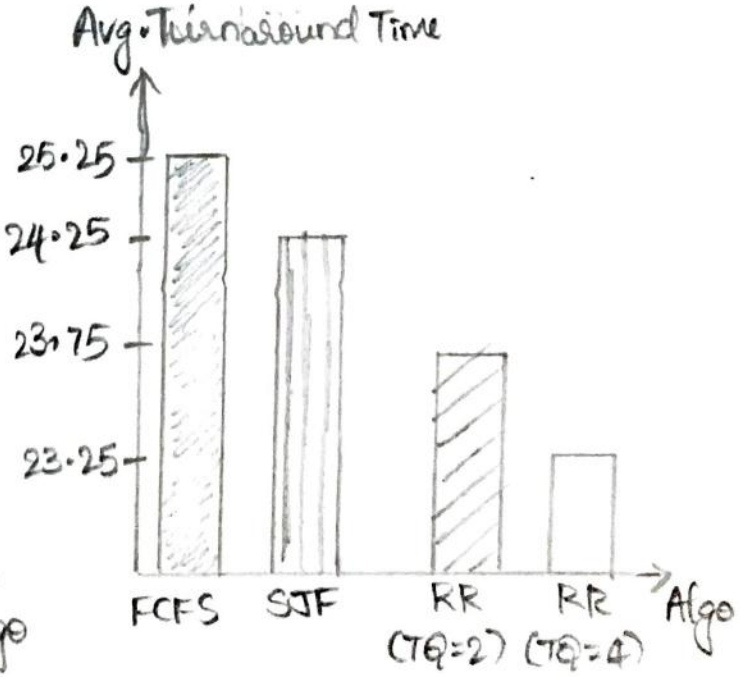
	P1	P2	P3	P4
Waiting Times	18	3	19	17
Turnaround times.	36	7	28	22

Avg. Waiting Time = 14.25 ms

Avg. Turnaround Time = 23.25 ms.

In FCFS, the longer process P1 took up the CPU for longer time before other processes and caused increase in avg. waiting time and average turnaround time.





In SJF, since the shortest job gets preference, P4 got executed before P3. Otherwise it would've been same as FCFS. We can see a dip in the avg. wait time and avg. turnaround times.

In RR(TQ=2), clearly the amount of time the processes get is distributed due to the time slices. This reduces the effect seen in FCFS and hence a decrease in the avg. wait times and avg. turnaround times.

In RR(TQ=4), on comparision with RR(TQ=2), the avg. wait times and turnaround times have reduced ~~further because the no. of context switches to~~ further because the processes take lesser time to complete, so turn around time decreases.

But all these explanations are not true in all scenarios. ~~They~~ The changes in the avg. wait, turn-around times can vary. We can't expect a specific pattern of change always.