# UNIT - II

## PROCESS

① It is a heavy weight task
OS treats different process different-ly
System calls involved in process.

② Different process have different copies of Data, files, code.

③ Context switching is slower.

④ Blocking a process will not block another.

⑤ Independent

## THREADS

① It is a light weight task

② There is no system call involved.

③ All user level threads treated as single task for OS.

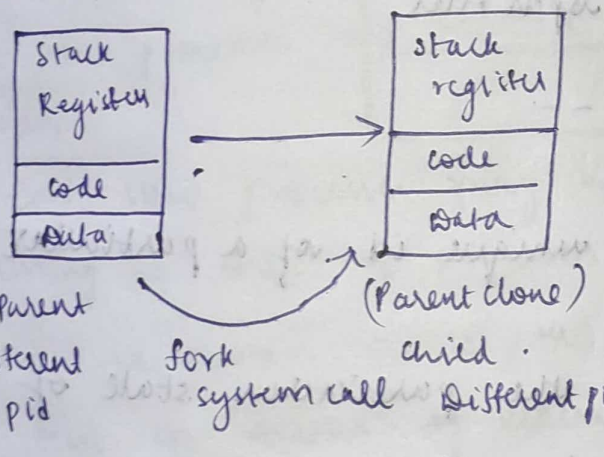
④ Threads share same copy of code and data

⑤ Context switching is faster

⑥ Blocking a thread will block entire process

⑦ Inter dependent

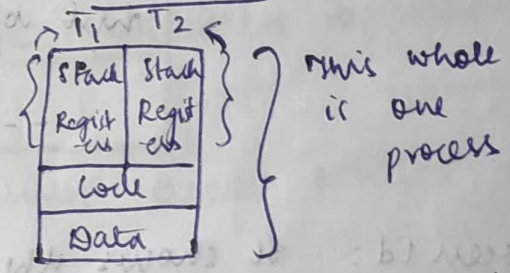
**Process:**




But same parent pid.
They both are different process

> For context switching first we have to save the values in Process control block and switch to other so it is more time consuming

**Threads:**




They have different stacks & registers.

> It is faster because only switching happens for registers and stacks.

> If $T_1$ is blocked (due to any I/O request). ~~Treat~~ Threading is user level responsibility and so kernel assumes that whole thing as a single process so it will be blocked entirely with all the threads.

A process can be thought of as a program in execution

A thread is the unit of execution within a process. A process can have anywhere from just one thread to many threads.

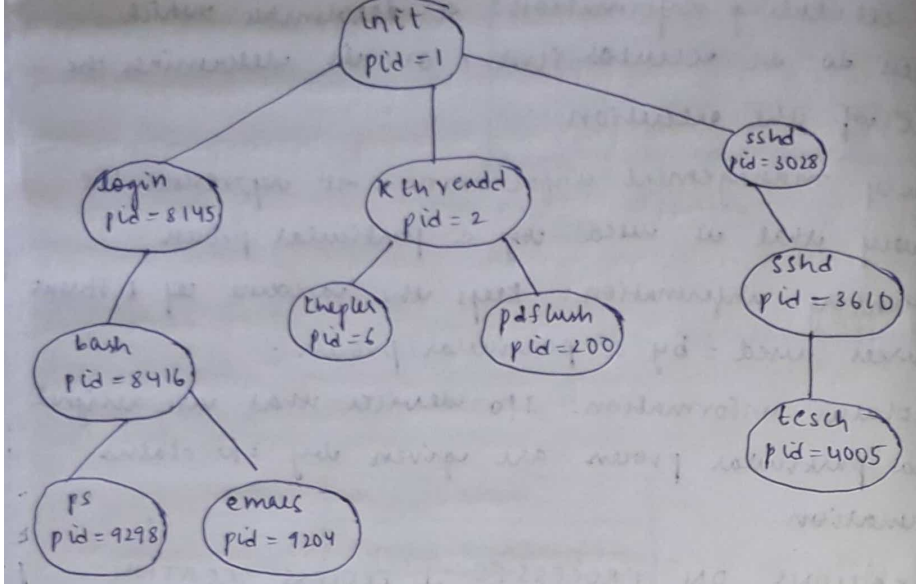
→ Process control Block:
Each process is represented in the operating system by a process control block.

| |
|---|
| Process state |
| Process number / ID |
| Program counter |
| Registers |
| Memory limits |
| List of open files. |
| ---- |

Process id: It shows the unique id of a particular process.
Process state: It tells us the particular state of the process at a particular moment.
Program counter: It indicates the address of the next instruction that has to be executed.
[Eg: When a process is being executed the lines of code are executed line by line, when they are executed the program counter job is to tell the address of the next line of instruction that has to be executed.]
CPU registers: It tells us the registers used by a particular process

CPU scheduling information: It determines which process to be executed first, or it determines the order of the execution
Memory management information: It represents the memory that is used by a particular process
Accounting information: Keeps the account of different resources used by a particular process.
I/O status information: I/O devices that are assigned to a particular process are given by I/O status information

→ OPERATIONS ON PROCESSES: 1) PROCESS CREATION:
A process may create several new processes, via create process system call, during the course of execution.
> The creating process is called a parent process, and the new processes are called the children of that process.
> Each new processes may inturn create other processes forming a tree of processes.
> When a process creates a new process, two possibilities exist in terms of execution:
1. The parent continues to execute concurrently with its children.
2. The parent waits until some or all of its children have terminated.
> There are also two possibilities in terms of the address space of the new process.
1. The child process is the duplicate of the parent process (it has same program and data as the parent).
2. The child process has a new program loaded into it.




A tree processes on a typical Linux system

Program:

```
#include<sys/types.h>
# include < stdio.h>
# include < unistd.h>
int main()
{ pid_t pid;
    pid = fork();
    if (pid<0)
    { fprintf (stderr, "fork failed");
      return 1;
    }
    else if (pid ==0)
    { execlp ("/bin/ls","ls", NULL);
    }
    else
    { wait (NULL);
      printf ("child complete");
    }
}
```

ii) PROCESS TERMINATION:

> A process terminates when it finishes executing its final statement and asks the OS to delete it by using the exit () system call.

> At that point, the process may return a status value (typically an integer) to its parent process (via the wait() system call).

> All the resources of the process - including physical and virtual memory, open files, and I/O buffers - are deallocated by the OS.

Termination can occur in other circumstances as well

> A process can cause the termination of another process via an appropriate system call.

> Usually, such a system call can be invoked only by the parent of the process that is to be terminated.

> Otherwise, users would arbitrarily kill each other's jobs.

A parent may terminate the execution of one of its children for a variety of reasons, such as these:

> The child has exceeded its usage of some of the resources that it has been allocated. (To determine whether this has occurred, the parent must have a mechanism to inspect the state of its children).

> The task assigned to the child is no longer required

> The parent is exiting, and the OS does not allow a child to continue if its parent terminates.

## ⇒ INTER PROCESS COMMUNICATION:

Processes executing concurrently in the OS may be either independent processes or cooperating processes.

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

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

Eg: Any process that shares data with other process is a cooperating process.

We require inter process communication in cooperating processes.

There are several reasons for providing an environment that allows process cooperation.

i) Information sharing: Since several users may be interested in the same piece of information, eg: a shared file, we must provide an environment to allow concurrent access.

ii) Computation speed: If we want a particular task to run faster, we must break it into subtasks, each of which will be executing in parallel with the others. Speed up can be achieved only if the computer has multiple processing cores.

iii) Modularity: We may want to construct the system in a modular fashion by dividing the system functions into separate processes or threads.

iv) Convenience: Even an individual user may work on many tasks at the same time. Eg: A user may be editing, listening to music, and compiling in parallel.

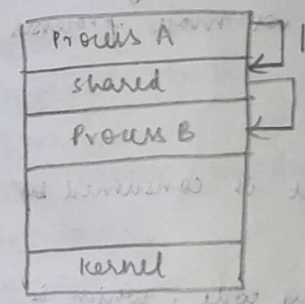
> Cooperating processes require an interprocess communication (IPC) mechanism that will allow them to exchange data and information.

There are two fundamental models of interprocess communication:

i) Shared Memory: Here, a region of memory that is shared by cooperating processes is established.

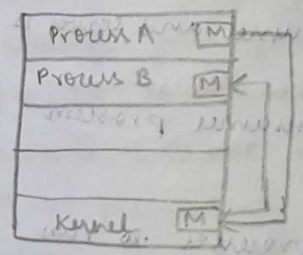
> Processes can then exchange information by reading and writing data to the shared region.

ii) Message Passing: Here, communication takes place by means of messages exchanged between the cooperating processes.




Shared memory.

When Process A wants to communicate with Process B, it will write the information that it wants to communicate in the shared memory region. Process B will read the information from the shared memory. Shared memory is accessible to both of them.




Message passing.

When Process A wants to communicate with Process B it will send the message to Process B, Process A creates the message and it will go to the Kernel, and the Kernel will send the message to Process B.

i. Shared Memory systems:

> Inter process communication using shared memory requires communicating processes to establish a region of shared memory.

> Typically, a shared memory region resides in the address space of the process creating the shared memory segment [Process A]

> Other processes that wish to communicate using this shared-memory segment must attach it to their address space. [In the prev diag Process B]

> Normally the operating system tries to prevent one process from accessing another process's memory.

> Shared memory requires that two or more processes agree to remove this restriction.

Producer Consumer Problem:

> A producer produces information that is consumed by a consumer process

Eg - i. A compiler may produce assembly code, which is consumed by an assembler.

ii. In client-server client is the consumer and server is the producer.

⇒ One solution to the producer-consumer problem uses shared memory.

> To allow producer and consumer processes to run concurrently, we must have available a buffer of items that can be filled by the producer and emptied by the consumer.

> ~~One solution to~~ This buffer will reside in a region of memory that is shared by the producer and consumer processes.

> A producer can produce one item while the consumer is consuming another item.

> The producer and consumer must be synchronised, so that the consumer does not try to consume an item that has not yet been produced.

Two kinds of buffers

Unbounded buffer

Places no practical limit on the size of the buffer. The consumer may have to wait for the new items, but the producer can always produce new items

Bounded buffer

Assumes a fixed buffer size. In this case, the consumer must wait if the buffer is empty, and the producer must wait if the buffer is full.

```
#define BUFFER_SIZE 10
typedef struct {
} item;
item buffer [BUFFER_SIZE];
int in = 0;
int out = 0;
```

```
item next_consumed;
while (true) {
while (in == out)
    next_consumed = buffer[out]
    ~~out = 1~~
    out = (out + 1) % buffer size
}
```

Message Passing systems:

Message passing provides a mechanism to allow processes to communicate and to synchronize their actions without sharing the same address space.

It is particularly useful in a distributed environment where the communicating processes may reside on different computers by a network.

A message passing facility provides atleast two operations:

> Send message & > Recieve message.

Messages sent by a process can be either of fixed or

variable size

Fixed size: The system-level implementation is straightforward. But makes the task of programming more difficult.

Variable size: It requires more complex system level implementation. But the programming task becomes simpler.

> If Processes P & Q wants to communicate, they must send messages to and recieve messages from each other. A communication link must exist between them.

This link can be implemented in a variety of ways. There are several methods for logically implementing a link and the send() / recieve() operations like

> Direct or indirect communication
> Synchronous or asynchronous communication
> Automatic or explicit buffering

} several issues related with features like naming, synchronization, buffering.

i, Naming:

Processes that want to communicate must have a way to refer to each other. They can use either direct or indirect communication.

Direct communication: Each process that ~~must~~ wants to communicate must explicitly name the recipent or sender of the communication

- send (P, message) - send a message to process P
- recieve (Q, message) - Recieve a message from process Q.

A communication link in this scheme has the following properties:

> 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.
> An link is associated with exactly two processes.
> Between each pair of processes, there exists exactly one link.

This scheme exhibits symmetric addressing; i.e., both the sender process and the reciever process must name the other to communicate.

Another variant of Direct communication:

Here only the sender names the recipent, the recipent is not required to name the sender.

- send (P, message) - send a message to process P
- recieve (id, message) - Recieve a message from any process. The variable id is set to the name of the process with which communication has taken place.

This scheme employs asymetry in addressing.

Disadvantage of the above scheme is limited modularity of the resulting process definitions. Changing the identifier of a process may necessitate examining all other process definitions.

ii Indirect communication: The messages are sent to and recieved from mail boxes, or ports.

> A mailbox can be viewed abstractly as an object into which messages can be placed by processes and from which messages can be removed.
> Each mailbox has an unique identification.

Two processes can communicate only if the processes have a shared mail box

- send (A, message) - send a message to mailbox A
- receive (A, message) - Recieve a message from mailbox A.

Communication link in this scheme has the following properties:

> A link is established b/w a pair of processes only if

both members of the pair have a shared mail box.

- A link may be associated with more than two processes.
- Between each pair of communicating processes, there may be no. of different links with each link corresponding to one mail box.

Q) Process $P_1$ sends a message to A, while both $P_2$ & $P_3$ execute recieve () from A. Which will recieve the message sent by P1?

A) Method 1: Allow a link to be associated with two processes almost.

Method 2: Allow at most one process at a time to execute a recieve() operation.

Method 3: Allow the system to select arbitarily which process will recieve the message. The system may also define an algorithm for selecting which process will recieve the message.

- A mailbox may be owned either by a process or by the OS.

i) Synchronisation:

communication b/w processes takes place through call to send () and recieve primitives. There are different design options for implementing each primitive.

Message passing may be either blocking or non-blocking also known as synchronous or asynchronous.

Blocking send: The sending process is blocked until the message is recieved by the recieving process or by the mailbox.

Non-Blocking send: The sending process sends the message and resumes operation.

Blocking recieve: The reciever blocks until the message is available.

Non-blocking recieve: The reciever retrieves either a valid message or null.

ii) Buffering:

whether communication is direct or indirect, messages exchanged by communicating processes reside in a temporary queue. Basically such queues are implemented in three ways:

Zero capacity: The queue has a maximum length of zero; thus the link cannot have any messages in it. In this case the sender must block until the recipient recieves the message.

Bounded capacity: The queue has finite length n; thus, at most n messages can reside in it. If the queue is not full when a message is sent, the message is placed in the queue and the sender can continue execution without waiting. The links capacity is finite, however, if the link is full, the sender must block until space is available in the queue.

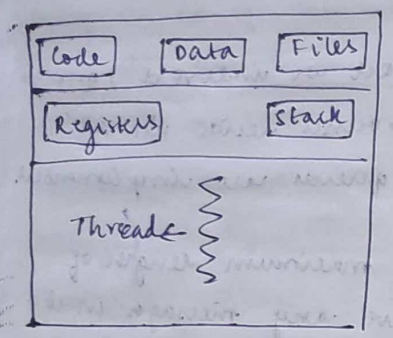
Unbounded capacity: The queues length is potentially infinite; thus, any no. of messages can wait in it. The sender never blocks.

⇒ THREADS:

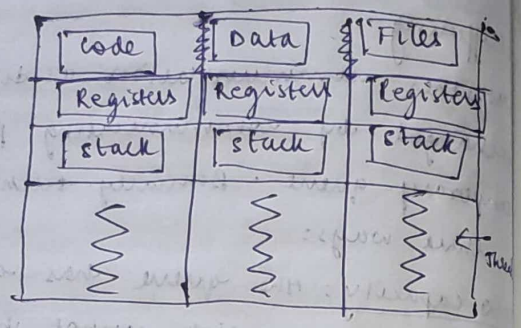
A thread is a basic unit of CPU utilization. It comprises:

- A thread ID
- A program counter
- A register set and
- A stack

A traditional / heavy weight process has a single thread of control.
If a process has multiple threads of control, it can perform more than one task at a time.



single threaded process



Multi-threaded process

Benefits:

Responsiveness: Multithreading an interactive application may allow a program to continue running even if part of it is blocked or is performing a lengthy operation, thereby increasing responsiveness to the user.

Resource sharing: By default threads share the memory and the resources of the process to which they belong. The benefit of sharing code and data is that it allows an application to have several different threads of activity within the same address space.

Economy: Allocating memory and resources for process creation is costly. Because threads share resources of the process to which they belong, it is more economical to create and context-switch threads.

Utilization of multiprocessor architectures: The benefits of multithreading can be greatly increased in a multiprocessor architecture, where threads may be running in parallel on different processors. A single threaded process can only run on one CPU, no matter how many are available. Multithreading on a multi-CPU machine increases concurrency.

→ MULTIPROGRAMMING:

> Multicore or multiprocessor systems putting pressure on programmers challenges include:
> Dividing activities
> Balance
> Data splitting
> Data dependency
> Testing and debugging.

Identifying tasks: This involves examining applications to find areas that can be divided into separate, concurrent tasks. Ideally, tasks are independent of one another and thus can run in parallel on individual cores.

Balance: While identifying tasks that can run in parallel, programmers must also ensure that the tasks perform equal work of equal value. In some instances, a certain task may not contribute as much value to the overall process as other tasks. Using a separate execution core to run that task may not be worth the cost.

Data splitting: Just as applications divided into separate tasks, the data accessed and manipulated by the tasks must be divided to run on separate cores.

Data dependency: The data accessed by the tasks must

be examined for dependencies b/w two or more tasks. When one task depends on data from another, programmer must ensure that the execution of the tasks is synchronized to accomodate the data flow.

Testing & Debugging: When a program is running in parallel on multiple cores, many different execution paths are possible. Testing and debugging such concurrent program is inherently more difficult than testing and debugging single thread applications.

> Parallelism implies a system can perform more than one task simultaneously.

> Concurrency supports more than one task making progress.

| Concurrency | Parallelism |
|---|---|
| > Illusion of parallelism, where only one task is performed at a time | > Quality of occurring at the same time |
| > Interleaving | > Using multiple processors |
| > Context switching | > Multiple CPU's for operating multiple processes |
| > Single processing unit is enough | > Requires multiple processing unit |

- Concurrent execution on a single-core system:

single core | $T_1$ | $T_2$ | $T_3$ | $T_4$ | $T_1$ | $T_2$ | $T_3$ | $T_4$ | $T_1$ | ... |

time →

- Parallelism on multicore system

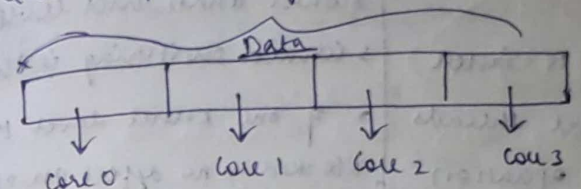
core 1 | $T_1$ | $T_3$ | $T_1$ | $T_3$ | $T_1$ |

core 2 | $T_2$ | $T_4$ | $T_2$ | $T_4$ | $T_2$ | ... |

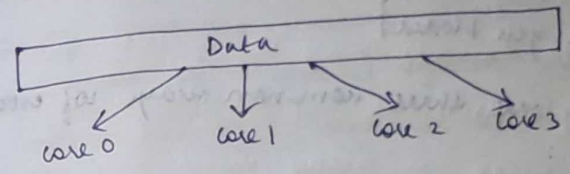
Types of Parallelism:

> Data Parallelism: Distribute subsets of same data across multiple cores, same operation on each.

> Task Parallelism: Distributing threads across cores, each thread performing unique operation.




Data Parallelism




Task Parallelism

Amdahl's Law:

$$\text{speedup} \leq \frac{1}{S + \frac{(1-S)}{N}}$$

It is a formula that identifies potential performance gains from adding additional computing cores to an application that has both serial and parallel components.

S is serial portion.

N is processing cores.

i.e., if application is 75% parallel / 25% serial, moving from 1 to 2 cores results in speedup of 1.6 times.

> As N approaches infinity, speedup approaches 1/S.

Serial portion of an application has disproportionate effect on performance gained by adding additional cores.

⟹ MULTITHREADING MODELS

| User Level Thread | Kernel Level Thread |
| --- | --- |
| > User level threads are managed by user level library. | > Kernel level threads are managed by OS system calls. |
| > User level threads are typically fast | > Kernel level threads are slower than user level. |
| > Context switching is faster | > Context switching is slower. |
| > If one user level threads perform blocking operation then entire process gets blocked | > If one kernel level thread blocked, no affect on others |

Ultimately there are three common ways of establishing the relationship:

1. Many to one
2. One to one
3. Many to Many Model.

i) Many-to-One Model:

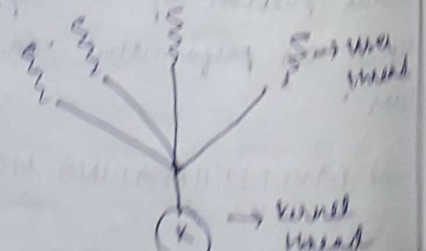
> Maps many user-level threads to one kernel thread.

> Thread management is done by the thread library in user space, so it is efficient

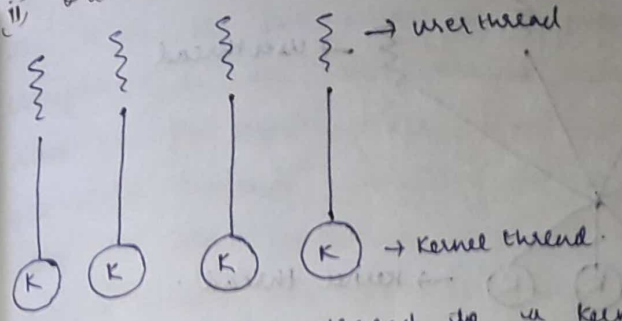
Disadvantage:

> The entire process will block if a thread makes a blocking system call.

> Because only one thread can access the kernel at a time, multiple threads are unable to run in parallel on multiprocessors.




ii) One-to-One Model:




> Maps each user thread to a kernel thread.

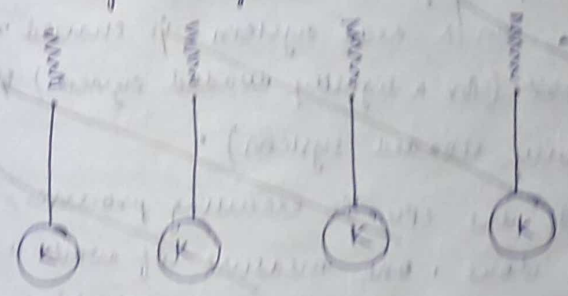
> Provides more concurrency than the many-to-one model by allowing another thread to run when a thread makes a blocking system call.

> Also allows multiple threads to run in parallel on multiprocessors.

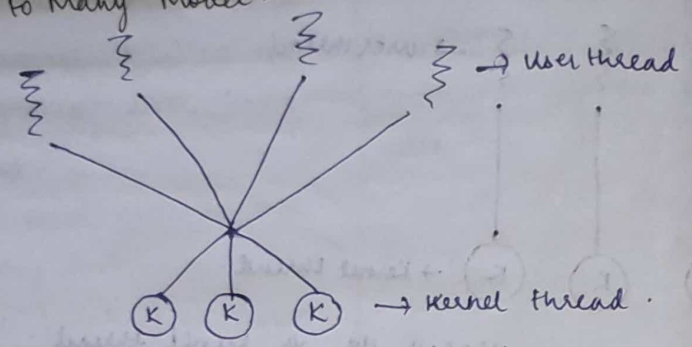
Disadvantage:

> Creating a user thread requires creating the corresponding kernel thread.

> Because the overhead of creating kernel threads can burden the performance of an application, most implementations of this model restrict the no of threads supported by the system.



Many to Many Model:




> Multiplexes many user-level threads to a smaller or equal number of kernel threads.

> The no. of kernel threads may be specific to either a particular application or a particular time.

> Developers can create as many users threads as necessary, and the corresponding kernel threads can run in parallel on a multiprocessor.

> Also, when a thread performs a blocking system call the kernel can schedule another thread for execution.

⇒ SCHEDULING CRITERIA:

i) CPU utilisation: We want to keep the CPU as busy as possible. Conceptually CPU utilisation can range from 0 to 100%. In a real system, it should range from 40% ~~to 90%~~ (for a lightly loaded system) to 90% (for a heavily loaded system).

ii) Throughput: If the CPU is executing processes, then work is being done. One measure of work is the no. of processes that are completed per unit time, called throughput.

iii) Turn around time: From the point of view of a particular process, the important criterion is how long it takes to execute that process. The interval from the time of submission of a process to the time of completion is the turn around time. Turn around time is the sum of the periods spent waiting to get into memory, waiting in the ready queue, executing on the CPU, and doing I/O.

Waiting time: The CPU scheduling algorithm does not affect the amount of time during which a process executes or does I/O; it affects only the amount of time that a process spends waiting in the ready queue. Waiting time is the sum of the periods spent waiting in the ready queue.

Response time: In an interactive system, turn around time may not be the best criterion. Often, a process can produce some output fairly early and can continue computing new results while previous results are being output to the user. Thus, another measure is the time from the submission of a request until the first response is produced. This measure, called response time, is the time it takes to start responding, not the time it takes to output the response. The turn around time is generally limited by the speed of the output device.

# LAB PROGRAMS:-

FCFS (First come first serve) Non Prememptive.

1. Question

| Process | Burst time | Complete time | Waiting time | TAT |
|---|---|---|---|---|
| $P_1$ | 24 | 24 | 0 | 24 |
| $P_2$ | 3 | 27 | 24 | 27 |
| $P_3$ | 3 | 30 | 27 | 30 |

Gantt chart :-

| $P_1$ | $P_2$ | $P_3$ |
|---|---|---|

0 24 27 30

Turn around time (TAT) = waiting time (WT) + Burst time (BT)

Turn around time (TAT) = complete time (CT) − Arrival time (AT)

Waiting time (WT) = Turn around time (TAT) − Burst time (BT)

2. Question

| Process | AT | BT | CT | TAT | WT | RT |
|---|---|---|---|---|---|---|
| $P_1$ | 2 | 2 | 4 | 2 | 0 | 0 |
| $P_2$ | 0 | 1 | 1 | 1 | 0 | 0 |
| $P_3$ | 2 | 3 | 7 | 5 | 2 | 2 |
| $P_4$ | 3 | 5 | 12 | 9 | 4 | 4 |
| $P_5$ | 4 | 4 | 16 | 12 | 8 | 8 |

Gantt Chart

| $P_2$ | Idel state | $P_1$ | $P_3$ | $P_4$ | $P_5$ |
|---|---|---|---|---|---|

0 1 2 4 7 12 16

TAT = 2 + 1 + 5 + 9 + 12 = 29

WT = 0 + 0 + 2 + 4 + 8 = 14

Avg TAT = $\frac{29}{5}$ = 5.8

3.

| Process | AT | BT | CT | TAT | WT |
|---|---|---|---|---|---|
| $P_1$ | 0 | 3 | 3 | 3 | 0 |
| $P_2$ | 1 | 1 | 4 | 3 | 2 |
| $P_3$ | 2 | 5 | 9 | 7 | 2 |
| $P_4$ | 3 | 2 | 11 | 8 | 6 |
| $P_5$ | 4 | 4 | 15 | 11 | 7 |

Gantt chart :

| $P_1$ | $P_2$ | $P_3$ | $P_4$ | $P_5$ |
|---|---|---|---|---|

0 3 4 9 11 15

$$\boxed{\begin{aligned} TAT &= CT - AT \\ TAT &= WT + BT \\ WT &= TAT - BT \end{aligned}}$$

SJF :- Auction  Shortest job first :- Non premptive

Eg 1 :-

| Process | Burst time | CT | TAT | WT |
|---|---|---|---|---|
| $P_1$ | 6 | 9 | 9 | 3 |
| $P_2$ | 8 | 24 | 24 | 16 |
| $P_3$ | 7 | 16 | 16 | 9 |
| $P_4$ | 3 | 3 | 3 | 0 |

Gantt chart :

| $P_4$ | $P_1$ | $P_3$ | $P_2$ |
|---|---|---|---|

0 3 9 16 24

Eg 2 :-

| Process | AT | BT | CT | TAT | WT |
|---|---|---|---|---|---|
| $P_1$ | 2 | 1 | 7 | 5 | 4 |
| $P_2$ | 1 | 5 | 16 | 15 | 10 |
| $P_3$ | 4 | 1 | 8 | 4 | 3 |
| $P_4$ | 0 | 6 | 6 | 6 | 0 |
| $P_5$ | 2 | 3 | 11 | 9 | 6 |

Gantt chart :-

| $P_4$ | $P_1$ | $P_3$ | $P_5$ | $P_2$ |
|---|---|---|---|---|

0 6 7 8 11 16

⇒ If the burst time is less but did not arrive at first, the first arrived process is executed. we should not leave computer idol.

Eg 3 :-

| Process | AT | BT | CT | TAT | WT |
|---|---|---|---|---|---|
| $P_1$ | 0 | 7 | 7 | 7 | 0 |
| $P_2$ | 2 | 4 | 12 | 10 | 6 |
| $P_3$ | 4 | 1 | 8 | 4 | 3 |
| $P_4$ | 5 | 4 | 16 | 11 | 7 |

Gantt chart

| $P_1$ | $P_3$ | $P_2$ | $P_4$ |
|---|---|---|---|

0 7 8 12 16

ROUND ROBIN (PRE EMPTIVE)  AT = 0 for all process

Time Quantum = 4

Eg

| Process | BT | CT | TAT | WT |
|---|---|---|---|---|
| $P_1$ | ~~24~~ ~~20~~ ~~16~~ | 30 | 30 | 6 |
| $P_2$ | ~~3~~ ~~12~~ ~~8~~ | 7 | 7 | 4 |
| $P_3$ | ~~3~~ | 10 | 10 | 7 |

Gantt chart

| $P_1$ | $P_2$ | $P_3$ | $P_1$ | $P_1$ |
|---|---|---|---|---|

0 4 7 10 14 18

Ready Queue

| $P_1$ | $P_1$ | $P_1$ |
|---|---|---|

22 26 30

| $P_1$ | $P_2$ | $P_3$ | $P_1$ | $P_1$ | $P_1$ | $P_1$ | $P_1$ |
|---|---|---|---|---|---|---|---|

Avg TAT :- 15.66

Avg WT :- 5.66

Q)

| Process | AT | BT | CT | TAT | WT | Response time |
|---|---|---|---|---|---|---|
| $P_1$ | 0 | ~~8~~ ~~5~~ ~~2~~ | 22 | 22 | 14 | 0 |
| $P_2$ | 5 | ~~2~~ | 11 | 6 | 4 | 4 |
| $P_3$ | 1 | ~~7~~ ~~4~~ ~~1~~ | 23 | 22 | 15 | 2 |
| $P_4$ | 6 | ~~3~~ | 14 | 8 | 5 | 5 |
| $P_5$ | 8 | ~~5~~ ~~2~~ | 25 | 17 | 12 | 9 |

Time Quantum = 3

RQ

| $P_1$ | $P_3$ | $P_1$ | $P_2$ | $P_4$ | $P_3$ | $P_5$ | $P_1$ | $P_3$ | $P_5$ |
|---|---|---|---|---|---|---|---|---|---|

G.C

| $P_1$ | $P_3$ | $P_1$ | $P_2$ | $P_4$ | $P_3$ | $P_5$ | $P_1$ | $P_3$ | $P_5$ |
|---|---|---|---|---|---|---|---|---|---|

0 3 6 9 11 14 17 20 22 23 25

$$\text{Avg TAT} = \frac{22+6+22+8+17}{5} = 15$$

$$\text{Avg WT} = \frac{14+4+15+5+12}{5} = 10$$

Q)

| Process | AT | BT | CT | TAT | WT | Response time |
|---|---|---|---|---|---|---|
| $P_1$ | 4 | 5 | 22 | 18 | 13 | 2 |
| $P_2$ | 6 | 7 | 25 | 19 | 12 | 6 |
| $P_3$ | 0 | 6 | 10 | 10 | 4 | 0 |
| $P_4$ | 3 | 4 | 16 | 13 | 9 | 1 |
| $P_5$ | 5 | 3 | 19 | 14 | 11 | 5 |

Avg TAT = 14.8

Avg WT = 9.8

Time Quantum = 2.

RQ :- | $P_3$ | $P_3$ | $P_4$ | $P_1$ | $P_3$ | $P_5$ | $P_2$ | $P_4$ | $P_1$ | $P_5$ | $P_2$ | $P_1$ | $P_2$ | $P_2$ |

GC :- | $P_3$ | $P_3$ | $P_4$ | $P_1$ | $P_3$ | $P_5$ | $P_2$ | $P_4$ | $P_1$ | $P_5$ | $P_2$ | $P_1$ | $P_2$ | $P_2$ |

0 2 4 6 8 10 12 14 16 18 19 21 22 24 25

Q)

| Process | AT | BT | CT | TAT | WT | Response time |
|---|---|---|---|---|---|---|
| $P_1$ | 0 | 5 | 13 | 13 | 8 | 0 |
| $P_2$ | 1 | 3 | 12 | 11 | 8 | 1 |
| $P_3$ | 2 | 1 | 5 | 3 | 2 | 2 |
| $P_4$ | 3 | 2 | 9 | 6 | 4 | 4 |
| $P_5$ | 4 | 3 | 14 | 10 | 7 | 5 |

Time Quantum = 2

RQ :- | $P_1$ | $P_2$ | $P_3$ | $P_1$ | $P_4$ | $P_5$ | $P_2$ | $P_1$ | $P_5$ |

GT :- | $P_1$ | $P_2$ | $P_3$ | $P_1$ | $P_4$ | $P_5$ | $P_2$ | $P_1$ | $P_5$ |

0 2 4 5 7 9 11 12 13 14

Avg TAT = ~~5.8~~ 8.6

Avg WT = 5.8

Q)

| Process | AT | BT | CT | TAT | WT | Response time |
|---|---|---|---|---|---|---|
| $P_1$ | 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 |

TQ = 2

Avg TAT = 7.75

Avg WT = 4.75.

RQ | $P_1$ | $P_2$ | $P_3$ | $P_4$ | $P_4$ | $P_2$ | $P_1$ |

GC | $P_1$ | $P_2$ | $P_3$ | $P_1$ | $P_4$ | $P_2$ | $P_1$ |

0 2 4 6 8 9 11 12

Response time = CPU first time − AT

d) Priority

TAT = BT+WT

18)

| P | Priority | BT | CT | TAT | WT |
|---|---|---|---|---|---|
| $P_1$ | 3 | 10 | 16 | 16 | 6 |
| $P_2$ | 1 | 1 | 1 | 1 | 0 |
| $P_3$ | 4 | 2 | 18 | 18 | 16 |
| $P_4$ | 5 | 1 | 19 | 19 | 18 |
| $P_5$ | 2 | 5 | 6 | 6 | 1 |

| $P_2$ | $P_5$ | $P_1$ | $P_3$ | $P_4$ |

0 1 6 16 18 19

Avg TAT = 12

Avg WT = 8.2

20)

| P | Priority | AT | BT | CT | TAT | WT |
|---|---|---|---|---|---|---|
| $P_1$ | 3 | 0 | 8 | 8 | 8 | 0 |
| $P_2$ | 4 | 1 | 2 | 17 | 16 | 14 |
| $P_3$ | 4 | 3 | 4 | 21 | 18 | 14 |
| $P_4$ | 5 | 4 | 1 | 22 | 18 | 17 |
| $P_5$ | 2 | 5 | 6 | 14 | 9 | 3 |
| $P_6$ | 6 | 6 | 5 | 27 | 21 | 16 |
| $P_7$ | 1 | 10 | 1 | 15 | 5 | 4 |

| $P_1$ | $P_5$ | $P_7$ | $P_2$ | $P_3$ | $P_4$ | $P_6$ |

0 8 14 15 17 21 22 27

30)

| P | AT | BT | Priority | CT | TAT | WT |
|---|---|---|---|---|---|---|
| $P_1$ | 0 | 11 | 2 | 11 | 11 | 0 |
| $P_2$ | 5 | 28 | 0 | 39 | 34 | 6 |
| $P_3$ | 12 | 2 | 3 | 51 | 39 | 37 |
| $P_4$ | 2 | 10 | 1 | 49 | 47 | 37 |
| $P_5$ | 9 | 16 | 4 | 67 | 58 | 42 |

| $P_1$ | $P_2$ | $P_4$ | $P_3$ | $P_5$ |

0 11 39 49 51 67

2. Program to simulate multi-level queue scheduling algorithm.

Eg1:

| Process | AT | BT | Q.No. | CT | TAT | WT |
|---|---|---|---|---|---|---|
| $P_1$ | 0 | ~~4~~ 2 | 1 | | | |
| $P_2$ | 0 | ~~3~~ 1 | 1 | | | |
| $P_3$ | 0 | ~~8~~ 0 | 2 | | | |
| $P_4$ | 10 | ~~5~~ 1 | 1 | | | |
| $P_5$ | 11 | 4 | 2 | | | |

$Q_1 > Q_2$

RR FCFS

T = 2

Gantt chart

| $P_1$ | $P_2$ | $P_1$ | $P_2$ | $P_3$ | $P_4$ | $P_4$ | $P_4$ | $P_3$ | $P_5$ |
|---|---|---|---|---|---|---|---|---|---|

0 2 4 6 8 10 12 14 15 20 24

Eg:-

| Process | AT | BT | Q.No. | CT | TAT | WT |
|---|---|---|---|---|---|---|
| $P_1$ | 0 | ~~10~~ ~~7~~ ~~4~~ 3 | 2 | | | |
| $P_2$ | 3 | ~~7~~ 3 | 1 | | | |
| $P_3$ | 4 | 6 | 2 | | | |
| $P_4$ | 12 | ~~8~~ ~~1~~ | 1 | | | |
| $P_5$ | 18 | ~~8~~ 4 | 1 | | | |

$Q_1 > Q_2$

RR RR

$\boxed{T=4}$ $\boxed{T=3}$

$Q_1$ – $P_2$ $P_4$ $P_5$

$Q_2$ – ~~$P_1$~~ $P_3$ $P_1$

Gantt chart:

| $P_1$ | $P_2$ | $P_2$ | $P_1$ | $P_4$ | $P_4$ | $P_1$ | $P_5$ | $P_5$ | $P_1$ | $P_1$ | $P_3$ | $P_3$ |
|---|---|---|---|---|---|---|---|---|---|---|---|---|

0 4 7 10 12 16 17 18 22 26 29 32 34 36

Q) Queue 1 uses round robin Queue 2 uses FCFS

Eg:

| P | Q.No. | AT | BT | CT | TAT | WT |
|---|---|---|---|---|---|---|
| $P_1$ | 1 | 0 | ~~4~~ ~~2~~ 0 | 6 | 6 | 2 |
| $P_1$ | 1 | 0 | ~~3~~ ~~2~~ 1 | 7 | 7 | 4 |
| $P_2$ | 2 | 0 | ~~8~~ 5 | 20 | 20 | 12 |
| $P_3$ | 1 | 10 | ~~5~~ ~~2~~ ~~1~~ 0 | 15 | 5 | 0 |
| $P_4$ | | | | | | |

Time Quantum = 2

Avg TAT = 9.5

Avg WT = 4.5

| $P_1$ | $P_2$ | $P_1$ | $P_2$ | $P_3$ | $P_4$ | $P_4$ | $P_4$ | $P_3$ |
|---|---|---|---|---|---|---|---|---|

0 2 4 6 7 10 12 14 15 20

Q

Eg:

| P | AT | BT | Pr | Q |
|---|---|---|---|---|
| $P_1$ | 3 | 4 | 2 | 1 |
| $P_2$ | 5 | 9 | 1 | 1 |
| $P_3$ | 8 | 4 | 2 | 2 |
| $P_4$ | 0 | 7 | 1 | 2 |
| $P_5$ | 12 | 6 | 1 | 1 |

$Q_1$ – NPP (Non preemptive)

$Q_2$ – SJF (Preemptive)

G.C

| $P_4$ | $P_1$ | $P_2$ | $P_5$ | $P_4$ | $P_3$ |
|---|---|---|---|---|---|

0 3 7 16 22 26 30

Eg:

| P | AT | BT | Pr | Q |
|---|---|---|---|---|
| $P_1$ | 10 | 5 | 2 | 1 |
| $P_2$ | 8 | ~~4~~ 2 | 2 | 2 |
| $P_3$ | 12 | 4 | 1 | 2 |
| $P_4$ | 3 | ~~3~~ | 2 | 2 |
| $P_5$ | 15 | 5 | 1 | 1 |

$Q_1$ – Preemptive priority

$Q_2$ – ~~Non~~ SJF

$Q_1$ | | $P_1$ | $P_5$

$Q_2$ | $P_4$ | $P_2$ | | | $P_2$

G.C

| | $P_4$ | | $P_2$ | $P_1$ | $P_1$ | $P_5$ | $P_2$ | $P_3$ |
|---|---|---|---|---|---|---|---|---|

0 3 6 8 10 12 15 20 22 26

$P_2$ $P_2$

24 26

# THEORY SUMS

## PREEMPTIVE CASE IN SJF:

→ for preemptive arrival time is compulsory.

eg 1:

| P | AT | BT |
|---|---|---|
| $P_1$ | 0 | ~~8~~ 7 |
| $P_2$ | 1 | ~~4~~ 0 |
| $P_3$ | 2 | 9 |
| $P_4$ | 3 | 5 |

| $P_1$ | $P_2$ | $P_2$ | $P_2$ | $P_2$ | $P_4$ | $P_1$ | $P_3$ |
|---|---|---|---|---|---|---|---|

0 1 2 3 4 5 10 17 26

G.C:

| $P_1$ | $P_2$ | $P_4$ | $P_1$ | $P_3$ |
|---|---|---|---|---|

0 1 5 10 17 26

eg:

| P | AT | BT |
|---|---|---|
| $P_1$ | 0 | ~~5~~ ~~4~~ 0 |
| $P_2$ | 1 | ~~3~~ ~~2~~ ~~1~~ 0 |
| $P_3$ | 2 | 4 |
| $P_4$ | 4 | ~~1~~ 0 |

| $P_1$ | $P_2$ | $P_2$ | $P_2$ | $P_4$ | $P_1$ | $P_3$ |
|---|---|---|---|---|---|---|

0 1 2 3 4 5 9 13

eg:-

| P | AT | BT |
|---|---|---|
| $P_1$ | 2 | ~~1~~ 0 |
| $P_2$ | 1 | 5 |
| $P_3$ | 4 | ~~1~~ 0 |
| $P_4$ | 0 | ~~6~~ ~~5~~ |
| $P_5$ | 2 | ~~3~~ 2 |

| $P_4$ | $P_1$ | $P_5$ | $P_3$ | $P_5$ | $P_4$ | $P_2$ |
|---|---|---|---|---|---|---|

0 2 3 4 5 7 11 16

eg:-

| P | AT | BT |
|---|---|---|
| $P_1$ | 0 | ~~7~~ 6 |
| $P_2$ | 1 | ~~5~~ 4 |
| $P_3$ | 2 | ~~3~~ ~~2~~ ~~1~~ 0 |
| $P_4$ | 3 | ~~1~~ 0 |
| $P_5$ | 4 | ~~2~~ 0 |
| $P_6$ | 5 | ~~1~~ |

| $P_1$ | $P_2$ | $P_3$ | $P_4$ | $P_2$ | $P_3$ | $P_6$ | $P_5$ | $P_2$ | $P_1$ |
|---|---|---|---|---|---|---|---|---|---|

0 1 2 3 4 5 6 7 9 13 19

## Premptive case in Priority Queue:

eg:

| P | AT | BT | P |
|---|---|---|---|
| $P_1$ | 0 | ~~10~~ ~~9~~ 0 | 2 |
| $P_2$ | 2 | ~~5~~ ~~4~~ 0 | 1 |
| $P_3$ | 3 | ~~2~~ 0 | 0 |
| $P_4$ | 5 | ~~20~~ 0 | 3 |

Less no. has more priority

| $P_1$ | $P_2$ | $P_3$ | $P_2$ | $P_1$ | $P_4$ |
|---|---|---|---|---|---|

0 2 3 5 9 17 37

eg:-

| P | P | AT | BT |
|---|---|---|---|
| $P_1$ | 10 | 0 | ~~5~~ 4 |
| $P_2$ | 20 | 1 | ~~4~~ 3 |
| $P_3$ | 20 | 2 | ~~2~~ ~~1~~ 0 |
| $P_4$ | 40 | 4 | ~~1~~ 0 |

More no. has more priority

| $P_1$ | $P_2$ | $P_3$ | $P_4$ | $P_3$ | $P_2$ | $P_1$ |
|---|---|---|---|---|---|---|

0 1 2 3 4 5 8 12

Eg:-

| P | AT | BT | PY |
|---|---|---|---|
| $P_1$ | 0 | ~~9~~ 0 | 5 |
| $P_2$ | 1 | ~~4~~ 0 | 3 |
| $P_3$ | 2 | ~~5~~ 0 | 1 |
| $P_4$ | 3 | ~~7~~ 0 | 2 |
| $P_5$ | 4 | ~~3~~ 0 | 4 |

| $P_1$ | $P_2$ | $P_3$ | $P_4$ | $P_2$ | $P_5$ | $P_1$ |
|---|---|---|---|---|---|---|

0 1 2 7 14 17 20 28

Eg:-

| P | AT | BT | PY |
|---|---|---|---|
| $P_1$ | 0 | ~~8~~ 0 | 3 |
| $P_2$ | 1 | ~~2~~ | 4 |
| $P_3$ | 3 | ~~4~~ | 4 |
| $P_4$ | 4 | ~~1~~ | 5 |
| $P_5$ | 5 | ~~6~~ ~~4~~ 0 | 2 |
| $P_6$ | 6 | ~~5~~ | 6 |
| $P_7$ | 10 | ~~10~~ | 1 |

| $P_1$ | $P_1$ | $P_1$ | $P_1$ | $P_5$ | $P_5$ | $P_7$ | $P_5$ | $P_1$ | $P_2$ | $P_3$ | $P_4$ | $P_6$ |
|---|---|---|---|---|---|---|---|---|---|---|---|---|

0 1 3 4 5 6 10 11 12 15 17 21 22 27

Eg:-

| P | AT | BT | PY |
|---|---|---|---|
| $P_1$ | 0 | ~~4~~ 0 | 2 |
| $P_2$ | 1 | ~~2~~ 0 | 4 |
| $P_3$ | 2 | ~~8~~ 0 | 6 |
| $P_4$ | 3 | ~~5~~ 0 | 10 |
| $P_5$ | 4 | ~~10~~ | 8 |
| $P_6$ | 5 | ~~4~~ 0 | 12 |
| $P_7$ | 6 | ~~6~~ 0 | 9 |

| $P_1$ | $P_2$ | $P_3$ | $P_5$ | $P_7$ | $P_4$ | $P_6$ |
|---|---|---|---|---|---|---|

0 4 6 9 10 16 21 25

ROUND ROBIN:

Eg:-

| P | AT | BT |
|---|---|---|
| $P_1$ | 0 | ~~5~~ ~~3~~ 1 |
| $P_2$ | 1 | ~~4~~ 2 |
| $P_3$ | 2 | ~~2~~ 0 |
| $P_4$ | 4 | ~~1~~ 0 |

T.Q = 2

RQ

| $P_1$ | $P_2$ | $P_3$ | $P_1$ | $P_4$ | $P_2$ | $P_1$ |
|---|---|---|---|---|---|---|

G.C

| $P_1$ | $P_2$ | $P_3$ | $P_1$ | $P_4$ | $P_2$ | $P_1$ |
|---|---|---|---|---|---|---|

0 2 4 6 8 7 10 12

Eg:-

| P | AT | BT |
|---|---|---|
| $P_1$ | 0 | ~~3~~ 0 |
| $P_2$ | 1 | ~~5~~ ~~3~~ 1 |
| $P_3$ | 2 | ~~2~~ 0 |
| $P_4$ | 3 | ~~5~~ ~~3~~ 1 |
| $P_5$ | 4 | ~~5~~ ~~3~~ 1 |

T.Q = 2

RQ

| $P_1$ | $P_2$ | $P_3$ | $P_1$ | $P_4$ | $P_5$ | $P_2$ | $P_4$ | $P_5$ | $P_2$ | $P_4$ | $P_5$ |
|---|---|---|---|---|---|---|---|---|---|---|---|

G.C

| $P_1$ | $P_2$ | $P_3$ | $P_1$ | $P_4$ | $P_5$ | $P_2$ | $P_4$ | $P_5$ | $P_2$ | $P_4$ | $P_5$ |
|---|---|---|---|---|---|---|---|---|---|---|---|

0 2 4 6 7 9 11 13 15 12 18 19 20

Eg:-

| P | AT | BT |
|---|---|---|
| $P_1$ | 0 | ~~4~~ 0 |
| $P_2$ | 1 | ~~5~~ 0 |
| $P_3$ | 2 | ~~2~~ 0 |
| $P_4$ | 3 | ~~1~~ 0 |
| $P_5$ | 4 | ~~6~~ 3 |
| $P_6$ | 6 | ~~3~~ 0 |

T.Q = 3

RQ:-

| $P_1$ | $P_2$ | $P_3$ | $P_4$ | $P_1$ | $P_5$ | $P_6$ | $P_2$ | $P_5$ |
|---|---|---|---|---|---|---|---|---|

G.C:

| $P_1$ | $P_2$ | $P_3$ | $P_4$ | $P_1$ | $P_5$ | $P_6$ | $P_2$ | $P_5$ |
|---|---|---|---|---|---|---|---|---|

0 3 6 8 9 10 13 16 18 21