Unit-2

Process Management:-

1) Introduction:- A process is an executing program. The CPU executes a large number of programs, while its main concern is the execution of user programs, the CPU is also needed for other system activities. These activities are called processes. A time-shared user program is a process. Process management is one of the function of operating system.

Process vs Program:-

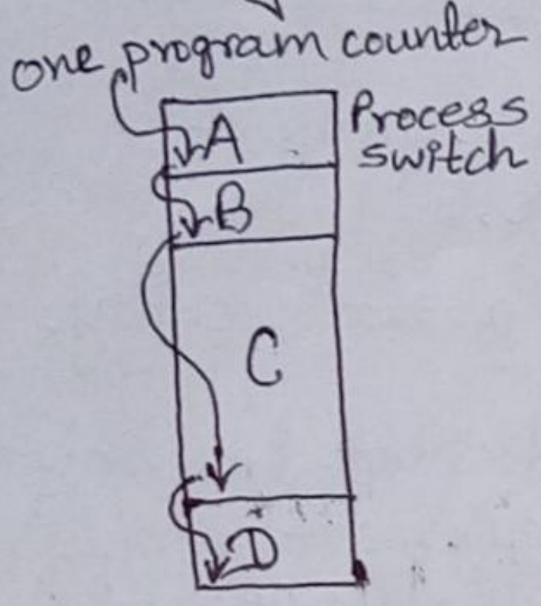
Process	Program.
i) Process is the activity.	i) Program is a group of instructions,
ii) Process is a dynamic entity.	ii) Process is a static entity.
iii) Process has a high resource requirement, it needs resources like CPU, memory, address, I/O during its lifetime.	iii) Program does not have any resource 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 its own control~~	
v) Process exists for a limited span of time as it gets terminated after the completition of task.	v) Program exists at a single place and continues to exist until it is deleted.

Multiprogramming:- In any multiprogramming system, the CPU switches from process to process quickly, running each for tens or hundreds of milliseconds. This rapid switching back and forth of the CPU is called multiprogramming. In multiprogramming systems, process are performed in a pseudo-parallelism as if each process has its own processor. In fact there is only one processor but it switches back and forth from process to process.

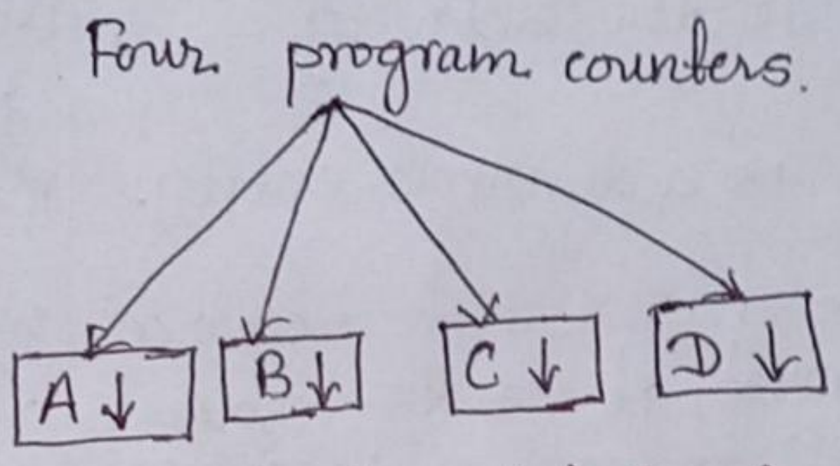
Keeping track of multiple parallel activities is hard for people to do. Therefore, operating system designers over the years have evolved a sequential process that makes parallelism easier to deal with.

Process Model:

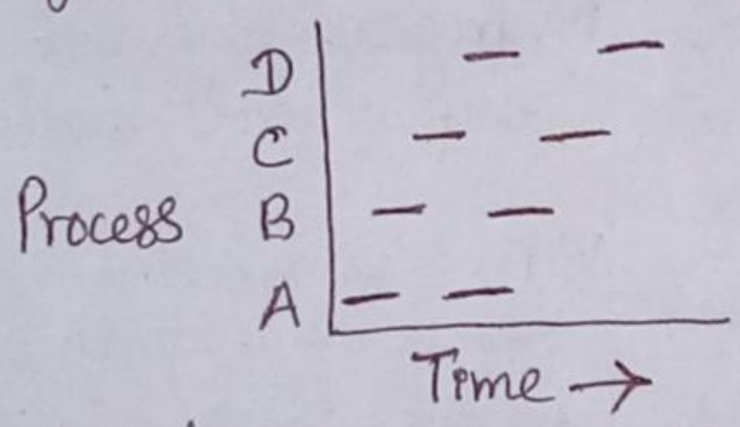
In this model all the runnable software on the computer, sometimes including the OS, is organized into a number of sequential processes. A process is just an instance of executing program, including the current values of program counter, registers and variables.



a) Multiprogramming four programs



b) Conceptual model for four independent, sequential processes.



c) Only one program is active at once.

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

Process states:

A process is an independent entity with its own input values, output values and internal state. One process may generate some outputs that other process uses as input. For example in the shell command.

cat file1 file2 file3 | grep tree

The first process running cat, concatenates and outputs three files. Depending on the relative speed of the two process, it may happen that grep is ready to run, but there is no input waiting for it. It must then block until some input is available. The process state may be in one of the following:

- New → The process is being created.
- Ready → The process is waiting to be assigned to a processor.
- Running → Instructions are being executed.
- Waiting/Suspended/Blocked → The process is waiting for some event to occur.
- Terminated → The process has finished execution.

The transition of the process states are shown in figure and their corresponding transition is described below:-

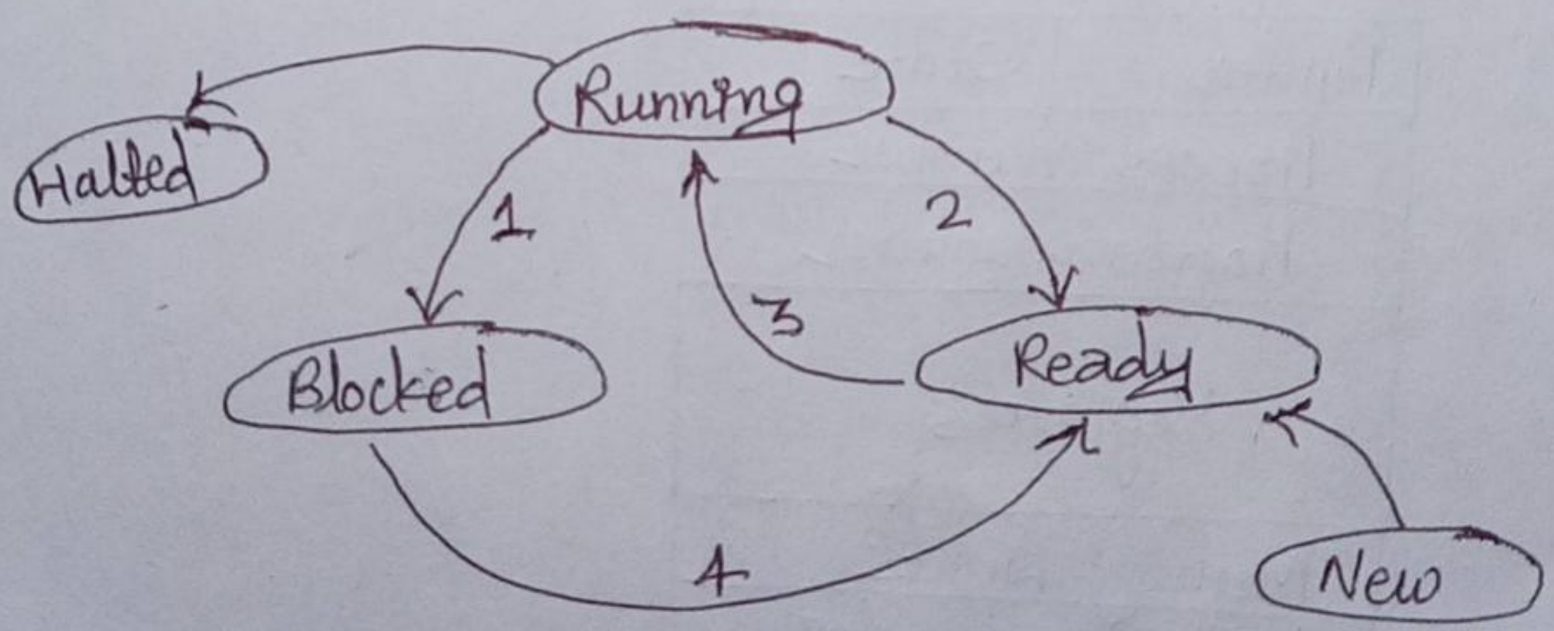


fig. Typical process states.

Process Control Block / Process Table (Implementation of Processes.)

To implement the process model, the operating systems ~~maintains~~ 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. The following is the information stored in a process table:

→ ~~Process state, each process~~.

→ Process state, which may be new, ready, running, waiting or halted.

→ Process number, each process is identified by its process number called process ID.

→ Program counter, which indicates the address of the next instruction to be executed for this process.

→ CPU registers, which vary in number and type, depending on the concrete microprocessor architecture.

→ Memory management information, which include 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 and so on.

→ Processor scheduling information, which includes process priority, pointers to scheduling queues and any other scheduling parameters.

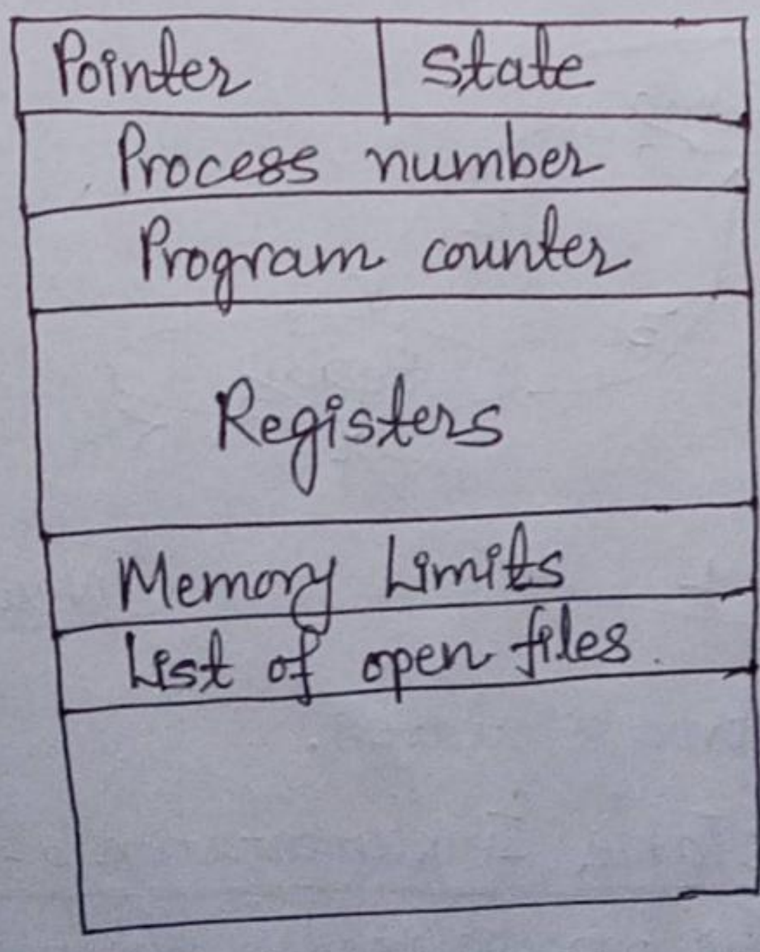


Fig. Process table structure.

2) Threads:

A thread is a path of execution within a process. A process can contain multiple threads. A thread is also called a lightweight processes (LWPs) which are independently scheduled parts of a single program. Each thread represents a seperate flow of control. Threads provide a way to improve application performance through parallelism. Threads represent a software approach to improve performance of OS by reducing the overhead thread. Threads have been successfully used in implementing network servers and web server. They also provide a suitable foundation for parallel execution of applications on shared memory multiprocessors.

Thread vs Process:-

Process	Thread.
i) Process is heavy weight or resource intensive.	i) Thread is light weight, taking lesser resources than a process.
ii) Process switching needs interaction with operating system.	ii) Thread switching does not need.
iii) In multiple processing environments each process executes the same code but has its own memory and file resources.	iii) All threads can share same set of open files, child processes.
iv) If one process is blocked, then no other processes can execute until the first process is unblocked.	iv) While one thread is blocked and waiting, second thread in the same task can run.
v) Multiple processes without using threads use more resources.	v) Multiple threaded processes use fewer resources.
vi) In multiple processes each process operates independently of the others.	vi) One thread can read write or change another thread's data.

Types of Threads:-

Threads are implemented in following two ways:-

1) User Level threads → In this case, the thread management kernel is not aware of the existence of threads. The thread library contains code for creating and destroying threads, for passing data and message between threads, for scheduling thread execution and for saving and restoring thread contexts. The application starts with a single thread.

Advantages

→ Thread switching does not require kernel mode privileges.
→ User level thread can run on any operating system.
→ Scheduling can be application specific in the user level thread.
→ User level threads are fast to create and manage.

Disadvantages

→ In a typical operating system, most system calls are blocking.
→ Multithreaded application cannot take advantage of multiprocessing.

2) Kernel Level threads → In this case, thread management is done by the kernel. There is no thread management code in the application area. Kernel threads are supported directly by the operating system. Any application can be programmed to be multithreaded. All of the threads within an application are supported within a single process.

Advantages

→ Kernel can simultaneously schedule multiple threads from the same process on multiple processes.
→ If one thread in a process is blocked, the kernel can schedule another thread of the same process.
→ Kernel routines themselves can be multithreaded.

Disadvantages

→ Kernel threads are generally slower to create and manage than the user threads.
→ Transfer of control from one thread to another within the same process requires a mode switch to the kernel.

User-Level Threads vs. Kernel-Level Threads:-

User-level threads	Kernel-level threads.
i) User-level threads are faster to create and manage.	i) Kernel-level threads are slower to create and manage.
ii) Implementation is by a thread library at the user level.	ii) Operating system supports creation of kernel threads.
iii) User-level thread is generic and can run on any operating system.	iii) Kernel-level thread is specific to the operating system.
iv) Multi-threaded applications cannot take advantage of multiprocessing.	iv) Kernel routines themselves can be multithreaded.

3). Inter Process Communication (IPC):-

Inter process communication (IPC) is a mechanism which allows processes to communicate with each other and synchronize their actions. This communication could involve a process letting another process know that some event has occured or the transfering of data from one process to another. This allows a specific program to handle many user requests at the same time.

Inter process communication describes three issues. The first one is how one process can pass information to another. ~~This allows a specific program to~~ The second one is making sure two or more processes do not get in each others way. The third concerns proper sequencing when dependencies are present.

Race Conditions:

A race condition is a situation that may occur inside a critical section. This happens when the result of multiple thread execution in critical section differs according to the order in which the threads execute.

In some operating system, processes that are working together may share some common storage that each one can read and write. The shared storage may be in main memory or it may be a shared file. The location of the shared memory does not change the nature of the communication or the problems that arise. Situation like this where two or more processes are reading or writing some shared data and the final result depends on who runs precisely when race conditions are called.

Race conditions in critical sections can be avoided if the critical section is treated as an atomic instruction. Also, proper thread synchronization using locks or atomic variables can prevent race conditions.

Critical Section:

Sometimes processes have to access shared memory or files, or doing other critical things that can lead to races. That part of the program where the shared memory is accessed is called the critical region or critical section.

If we could arrange matters such that no two processes were ever in their critical regions at the same time we could avoid races. We need four conditions to hold to have a good solution:-

i) No two processes may be simultaneously inside their critical regions.

ii) No assumptions may be made about speeds or the number of CPUs.

iii) No process running outside its critical region may block other processes.

iv) No process should have to wait forever to enter its critical region.

4). Implementing Mutual Exclusion:

@ Mutual Exclusion with Busy Waiting:

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

i) Disabling Interrupts → On a single-processor system, the simplest solution to have each process is disabling all interrupts just after entering its critical region and re-enable them just before leaving it. Once a process has disabled interrupts, it can examine and update the shared memory without fear that any process will interrupt.

This approach is generally unattractive because it is unwise to give user processes the power to turn off interrupts. What if one of them did it, and never turned them on again? that could be the end of system. Thus, disabling interrupts is often a useful technique within the operating system itself but is not appropriate for user processes.

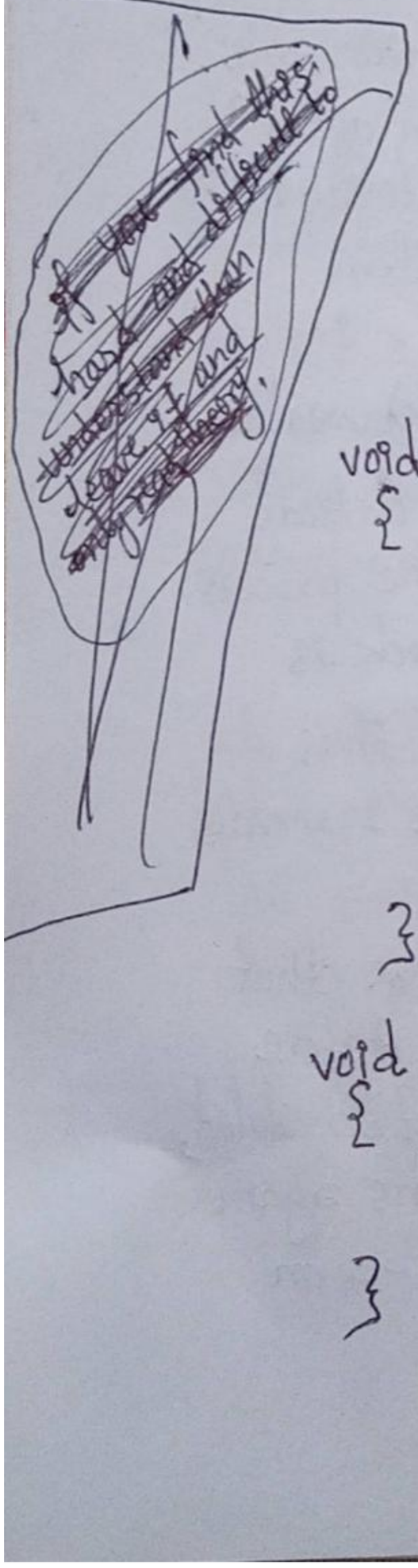
ii) Lock Variables: → Consider having a single, shared (lock) variable, initially set to 0. When a process wants to enter its critical region, it first tests the lock. If the lock is 0, the process sets it to 1 and enters the critical region. If the lock is already 1, the process just waits until it becomes 0. Thus 0 means that no processes in its critical region and 1 means that some process is in its critical region.

This idea also contains fatal flaw. Suppose that one process reads the lock and sees that it is 0. Before it can set the lock to 1, another process may get scheduled, runs and sets the lock to 1. When first process runs again it also sets the lock to 1. Now, two processes will be on their critical regions at the same time.

iii) Strict Alternation → The integer variable (turn) initially 0, keeps track of whose turn to enter critical region and updates shared memory. Initially first process finds value of turn 0 and enters it's critical region. Second process also finds it to be 0 and therefore sits in a tight loop continually testing turn ~~until~~ to see when it becomes 1.

Continuously testing a variable until some value appears is called busy waiting. It should usually be avoided since it wastes CPU time. A lock that uses by busy waiting is called a spin lock.

iv) Peterson's Solution → This algorithm consists of two procedures written in ANSI C, which means that function prototypes should be supplied for all the functions defined and used.




```
#define FALSE 0
#define TRUE 1
#define N 2  /* no. of processes */
    int turn;  /* whose turn is it */
int interested[N];  /* all values initially 0. (i.e, false) */

void enter_region(int process)  /* process is 0 or 1 */
{
    int other;  /* number of the other process */
    other = 1 - process;  /* the opposite of process */
    interested[process] = TRUE;  /* show that you are interested */
     turn = process;  /* set flag */
    while (turn = process && interested[other] = TRUE)
                                    /* null statement */
}

void leave_region (int process)  /* process, who is leaving */
{
    interested[process] = FALSE;  /* indicates departure from
                                     critical region */
}
```

Before entering its critical region, each process calls enter_region method (i.e, function) with its own process number 0 or 1 as parameter. This call will cause it to wait until it is safe to enter. After it has finished with the shared variables the process calls leave_region method to indicate that it is done and to allow other process to enter.

v) Test and Set Lock (The TSL Instruction) →

This is a proposal that requires a little help from hardware. It reads the contents of the memory word lock into register RX and then stores a non-zero value at the memory address lock.

To use the TSL instruction, we will use a shared variable, (lock) to coordinate access to shared memory. When lock is 0, any process may set it to 1 using the TSL instruction and then read or write the shared memory. When it is done, the process sets lock back to 0 using an ordinary move instruction.

⑥ Sleep and Wake up:

Both Peterson's solution and the solution using TSL are correct, but both have the defect of requiring busy waiting. These approaches have unexpected effects and waste CPU time. Sleep and Wakeup is one of the simplest inter process communication primitives that block instead of wasting CPU time.

Sleep is a system call that causes the caller to block, that is be suspended until another process wake it up. The wakeup call has one parameter, the process to be awakened. Alternatively, both sleep and wakeup each have one parameter, a memory address used to match up sleeps with wakeups.

⑦ Semaphore:

This was the situation in 1965, when E.W. Dijkstra (1965) suggested using an inter variable to count the number of wakeups saved for future use. In his proposal, a new ~~vari~~ variable type, called a semaphore was introduced.

A semaphore could have the value 0, indicating that no wakeups were saved, or some positive value if one or more wakeups were pending. Dijkstra proposed having two operations, down and up. The down operation on semaphore checks to see if the value is greater than 0. If so, it decrements the value and just continues.

If the value is 0, the process is put to sleep without completing the down for the moment. Checking the value, changing it and possibly going to sleep, is all done as a single indivisible atomic action. It is guaranteed that once a semaphore operation has started, no other process can access the semaphore until the operation has completed or blocked. This atomicity is absolutely essential to solve synchronization problems and avoiding race conditions.

(d) **Monitors:** 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 thread until that thread acquires the lock. This means that only one thread can execute within the monitor at a time. Any other threads must wait for the thread that is currently executing to give up control of the lock.

However a thread can actually suspend itself inside a monitor and then wait for an event to occur. If this happens, then 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.

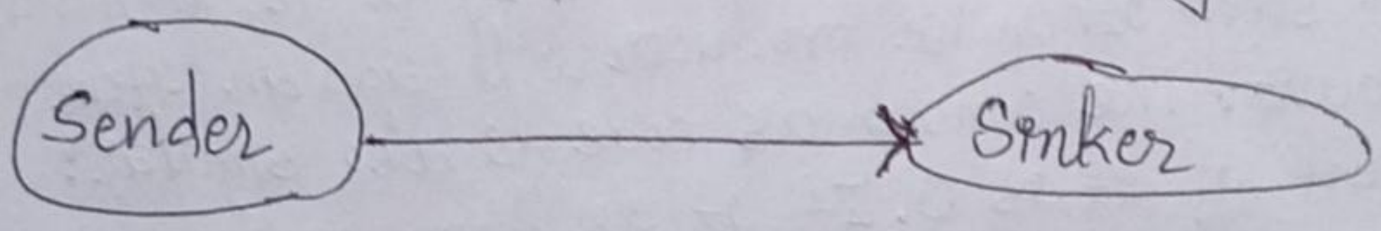
② Message Passing:

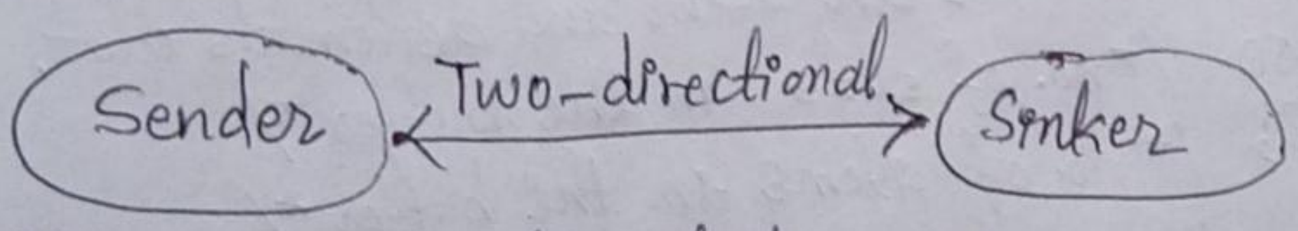
This method of inter process communication uses two primitives, send and receive, which like semaphores and unlike monitors, are system calls rather than language constructs. As such they can easily be put into library procedures, such as:

```
send(destination, &message);
```
and
```
receive(source, &message);
```

The former call sends a message to a given destination and the latter one receives a message from a given source. If no message is available, the receiver can block until one arrives. Alternatively, it can return immediately with an error code.

Message passing is commonly used in parallel programming systems like MPI (Message-Passing Interface) which is one of the well-known message-passing system. It is widely used for scientific computing.





The communication link in this scheme has following properties:

- A link is established automatically between every pair of processes that want to communicate.
- A link is associated with exactly two processes.
- Exactly one link exists between each pair of processes.

5) Classical IPC Problems:-

● These problems are used for process synchronization.

@Producer-Consumer Problem: (Bounded-Buffer problem)

In producer-consumer problem two processes share a common, fixed-size buffer. The producer puts information into the buffer and the consumer takes it out. Trouble arises when the producer wants to put a new item in the buffer, but it is already full. The solution is the producer should go to sleep, and to be awakened when the consumer has removed one or more items. Similarly, if the consumer wants to remove an item from the buffer and sees that the buffer is empty, it goes to sleep until the producer puts something in the buffer and wakes it up.

To keep the track of the number of items in the buffer we will need a variable count. If the maximum number of items the buffer can hold is N, then producers code will first test to see if count is N. If it is, then the producer will go to sleep otherwise producer will add an item and increments count. The consumer's code is also similar: first it test count if it is 0. If it is, then goes to sleep. If it is nonzero, it removes an item and decrements count.

```
#define N 100 /*no. of slots in the buffer*/
int count=0; /*no. of items in the buffer*/
void producer(void){
    int item;
    while(TRUE){
        item=produce_item(); /*generate next item*/
        if(count==N) sleep(); /*if buffer is full go to sleep*/
        insert_item(item); /*put item in buffer*/
        count=count+1;
        if(count==1) ~~wakep~~ wakeup(consumer); /*was buffer empty?*/
    }
}
```

```
void consumer(void){
    int item;
    while(TRUE){
        if(count==0) sleep();
        item=remove_item();
        count=count-1;
        if(count==N-1) wakeup(producer);
        consume_item(item);
    }
}
```

(b) Sleeping Barber problem:

In this problem there is a barber shop with one barber, one barber chair, and n chairs waiting for customers if there are any to sit on the chair.

→ If there is no customer, then barber sleeps in his own chair.

→ When a customer arrives, he has to wake up the barber.

→ If there are many customers and the barber is cutting a customer's hair, then the remaining customers either wait if there are empty chairs in waiting room or they leave if no chairs are empty.

The solution to this problem includes three semaphores. First is for the customer which counts no. of customers present in the waiting room. Second the barber, 0 or 1 is used to tell whether the barber is idle or working and third mutex is used to provide the mutual exclusion required for process to execute.

When the customer arrives, he acquires the mutex for entering critical region, if another customer enters thereafter, the second one will not be able to anything until the first one has released the mutex.

If the chair is available then the customer sits in the waiting room and increments the variable and also increases the customer's semaphore this wakes up the barber if he is sleeping. At this point, customer and barber both

are awake and the barber is ready to give that person a haircut. When the haircut is over, the customer exits the procedure and if there are no customers in waiting room barber sleeps.

Algorithm:

```
Semaphore customers=0;
Semaphore barber=0;
Mutex seats=1;
int freeSeats=N;

Barber { while (true){ down(customers); /*waits for customer*/
                       down(seats); /*mutex to protect no. of seats*/
                       freeSeats++;  /*a chair gets free*/
         up(barber); /*bring barber for haircut*/
         up(seats); /*release the mutex on the chair*/
         /*barber is now cutting hair*/
                     }
}

Customer { while(true){
             if(freeSeats>0){
                             freeSeats--; /*sitting down*/
                             up(customers); /*notify the barber*/
                             up(seats); /*release the lock*/
                             down(barber); /*wait in waiting room if
                                             barber is busy*/
                             /*Customer is now having haircut*/
                            }
             else{ up(seats); /*release the lock*/
                   /*customer leaves*/
                 }
           }
```

© Dinning Philosopher Problem:

refer to book theory image of this theory is difficult to understand

In this problem five philosophers are seated around a circle table for their lunch. Each philosopher has a plate of spaghetti (a kind of food). the spaghetti is so slippery that a philosopher needs two forks to eat it. Between each pair of plates is one fork. The philosophers alternate between thinking and eating.

When philosopher gets hungry, she tries to acquire her left and right fork one at a time, in either other. If successful in acquiring two forks, she eats for a while then puts down the forks and continues to think.

Solution to Dinning Philosopher problem

Attempt 1: When a philosopher is hungry, she picks up her left fork and waits for right fork. When she gets it, she eats for a while and then puts both forks back to the table.

The problem with this attempt is that, if all five philosophers take their left forks simultaneously then deadlock will happen.

Attempt 2: After taking the left fork, checks the right fork. If it is not available, the philosopher puts down the left one, waits for some time and then repeats whole problem.

Problem is that if all five philosophers take their left fork simultaneously then starvation will occur. A situation in which all the programs continue to run indefinitely but fail to make any progress is called starvation.

Final Attempt:- Among the many attempts using semaphore for each philosopher moves only in eating state if neither neighbour is eating is the perfect solution to the dinning philosopher problem.

Algorithm:

```
#define N 5 /*no. of philosophers*/
#define LEFT (i+N-1)%N /*no. of i's left neighbor*/
#define RIGHT (i+1)%N /*no. of i's right neighbor*/
#define THINKING 0 /*philosopher is thinking*/
#define HUNGRY 1 /*philosopher is trying to get forks*/
#define EATING 2 /*philosopher is eating*/
typedef int semaphore; /*semaphores are a special kind of int*/
int state[N]; /*array to keep track of everyone's state*/
semaphore mutex=1; /*mutual exclusion for critical regions*/
semaphore s[N]; /*one semaphore per philosopher*/
void philosopher(int i): /*i: philosopher number from 0 to N-1*/
{ while(TRUE){ /*repeat forever*/
        think(); /*philosopher is thinking*/
        take_forks(i) /*acquire two forks or block*/
        eat(); /*eating*/
        put_forks(i); /*put both forks back on table*/
        }
}
void take_forks(int i){
                down(&mutex); /*enter critical region*/
                state[i]=HUNGRY;
                test[i]; /*try to acquire 2 forks*/
                up(&mutex); /*exit critical region*/
                down(&s[i]); /*block if forks were not acquired*/
}
void put_forks(i){
                down(&mutex);
                state[i]=THINKING
                test(LEFT);
                test(RIGHT);
                up(&mutex);
}
void test(i){ if(state[i]==HUNGRY && state[LEFT]!=EATING && state[RIGHT]
                !=EATING)
        { state[i]=EATING;      up(&s[i]); }
}.
```

6.) 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 schedular and the algorithm it uses is called the scheduling algorithm.

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

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

Process Behaviour →

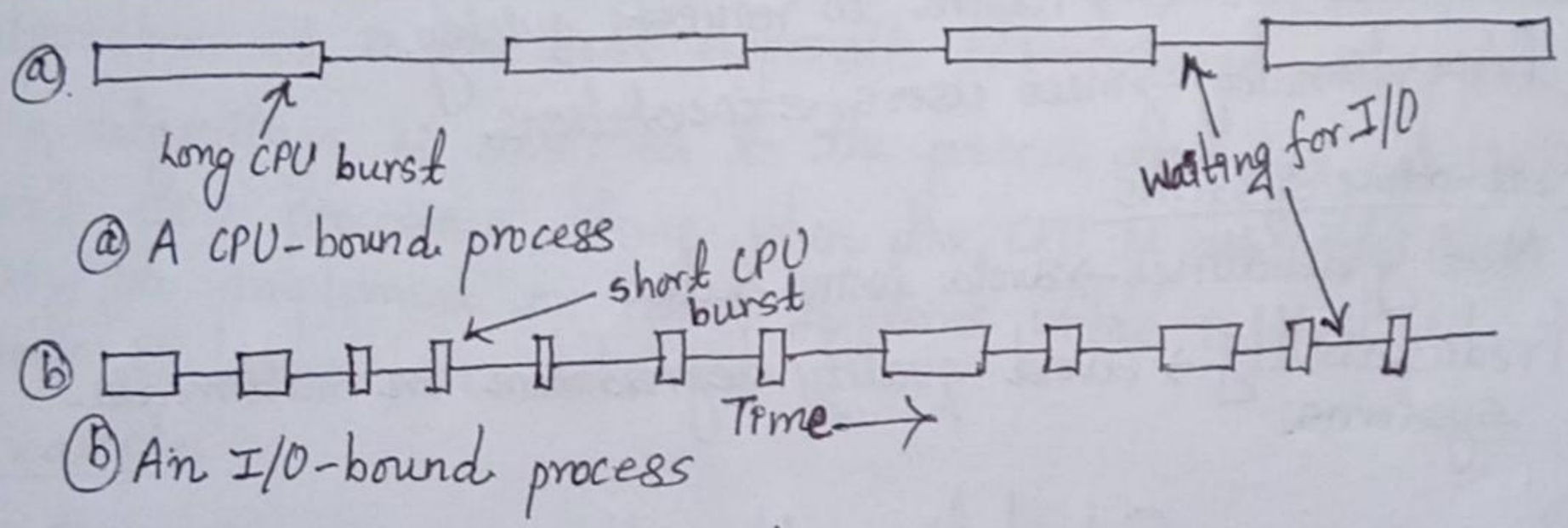


Fig. Bursts of CPU usage alternate with periods of waiting for I/O.

Typically the CPU runs for a while without stopping, then a system call is made to read from a file or write to a file. When the system call completes, the CPU computes again until it needs more data or has to write more data and so on.

Some processes such as fig@ spend most of their time on computing, while others such as fig⑥ spend most of their time waiting for I/O. The former are called compute-bound; the latter are called I/O-bound.

Scheduling Algorithm Goals:-

In order to design a scheduling algorithm, it is necessary to have some idea of what a good algorithm should do. Some goals depend on the environment (batch, interactive, or real time), but there are also some that are desirable in all cases. Some goals are listed below:

i) All systems

Fairness → giving each process a fair share of the CPU.
Policy enforcement → seeing that stated policy is carried out.
Balance → keeping all parts of system busy.

ii) Batch systems

Throughput → maximize jobs per hour.
Turnaround time → minimize time between submission and termination.
CPU utilization → keep the CPU busy all the time.

iii) Interactive systems

Response time → respond to requests quickly.
Proportionality → meet users expectations.

iv) Real-time systems

Meeting deadlines → avoid losing data.
Predictability → avoid quality degradation in multimedia systems.

Scheduling Algorithms:

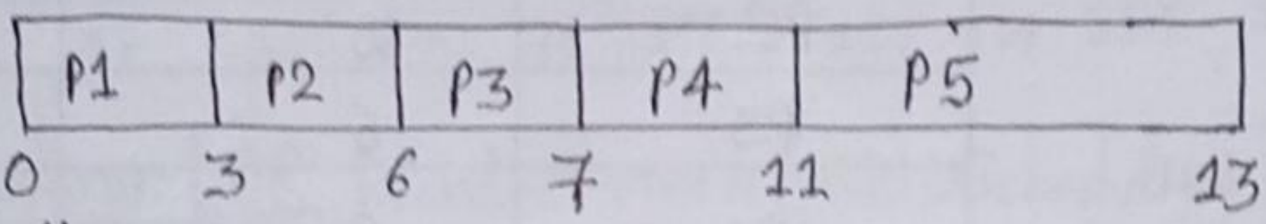
@ Batch System Scheduling:

1) First Come First Serve (FCFS):→ This is non-preemptive scheduling algorithm. This is the simplest scheduling algorithm. Jobs are scheduled in the order they are received. Implementation of this technique is easy and is based on FIFO queue. It has poor performance as average wait time is high.

Example:

Process	Arrival Time	Processing Time (In milliseconds)
P1	0	3
P2	2	3
P3	3	1
P4	5	4
P5	8	2

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



Note:- If all the processes arrive at the time 0, then the order of scheduling will be P3, P5, P1, P2 and P4.

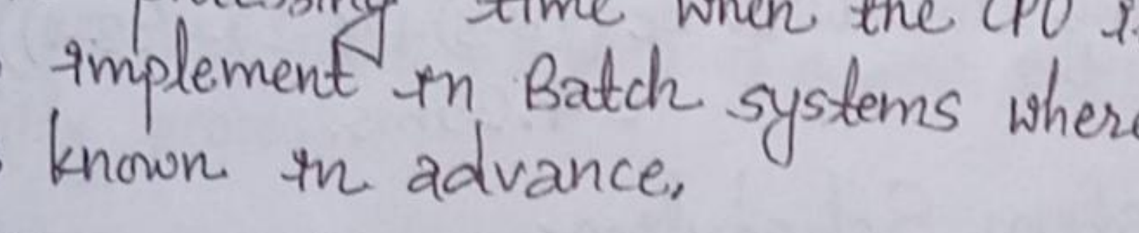
⇒ Average waiting time = (0+3+6+7+11)/5 = 5.4ms

ii) Shortest-Job First (SJF) → This is also non-preemptive scheduling algorithm. It is the best approach to minimize waiting time. This algorithm is assigned to the process that has smallest next CPU processing time when the CPU is available. It is easy to implement in Batch systems where required CPU time is known in advance.

Example:

Process	Processing Time (ms)
P1	6
P2	8
P3	7
P4	3

Using SJF, the Gantt chart will be:



Average waiting time = (0+3+9+16)/4 = 7ms.

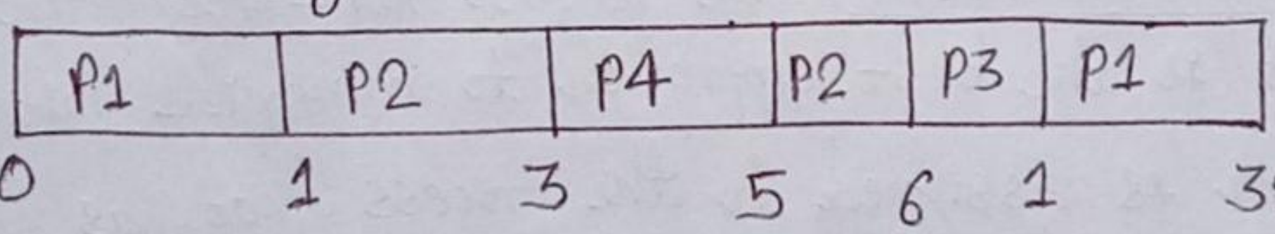
iii) Shortest Remaining Time Next → This is a preemptive scheduling algorithm. In this algorithm, process with the smallest amount of time remaning until completion is selected to execute. This method is advantageous because short processes are handeled very quickly. When a new process is added the algorithm only compare the currently executing process with the new process, ignoring all other processes currently waiting to execute.

Example:

~~The Gantt~~

Process	Arrival time	Processing Time (Burst Time)
P1	21	0
P2	3	1
P3	6	2
P4	2	3

The Gantt chart for preemptive shortest remaining time next scheduling will be.



The average waiting time will be ((5-3)+(6-2)+(12-1))/4
= 4.25ms.

(b) Interactive System Scheduling:

i) Round-Robin (RR) Scheduling:- It is one of the oldest, simplest and most widely used algorithm. Each process is assigned a time-interval called its quantum, during which it is allowed to run. If the process is still running at the end of the quantum, the CPU is preempted (i.e, stopped) and given to another process. If the process has blocked or finished before the quantum has elasped, the CPU switching is done when the process blocks. Round-Robin is easy to implement.

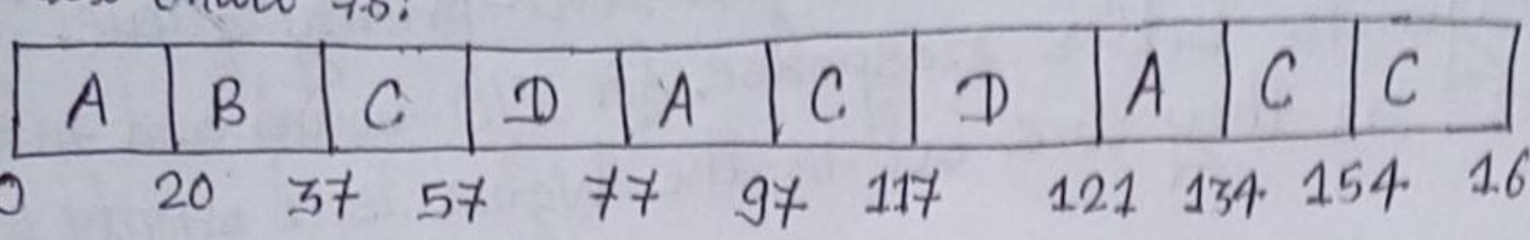
To implement the RR scheduling, Queue data structure is used. A new process is added to the tail of the Queue. The CPU schedular picks the first process from the Queue,

allocate processor for a specified time Quantum. After that the CPU schedular will select the next process in the ready Queue.

Example:

Process	Burst Time (Processing Time)
A	53
B	17
C	68
D	24

The Gantt chart is:



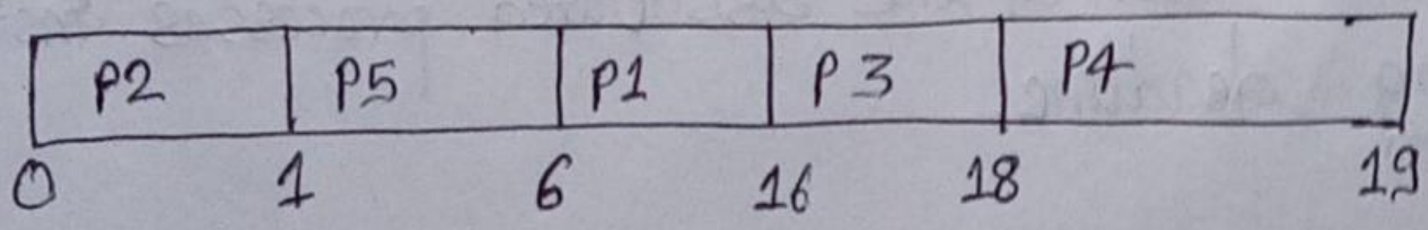
ii) Priority Scheduling: [OR Event-Driven (ED) Scheduling]

A priority is associated with each process and the schedular always picks up the highest priority process for execution from the ready queue. Equal priority processes are scheduled FCFS. The level of priority may be determined on the basis of resource requirements, process characteristics and its runtime behaviour. A major problem with a priority based scheduling is indefinite blocking of a lost priority process by a high priority process.

Example:

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

Using priority scheduling the Gantt chart is:



iii) Multiple Queues:- In this system there are multiple queues with a different priority level set for each queue. Processes in the highest priority level use less CPU time than processes in the next-highest priority level. Processes may move between the queues. If a process uses its entire quantum, it will be moved to the tail of the next-lower-priority-level queue while if the process blocks before using its entire quantum it is moved to the next-higher-level-priority queue. Its advantage is better response for I/O-bound and interactive processes as they are set in the higher level priority queue.

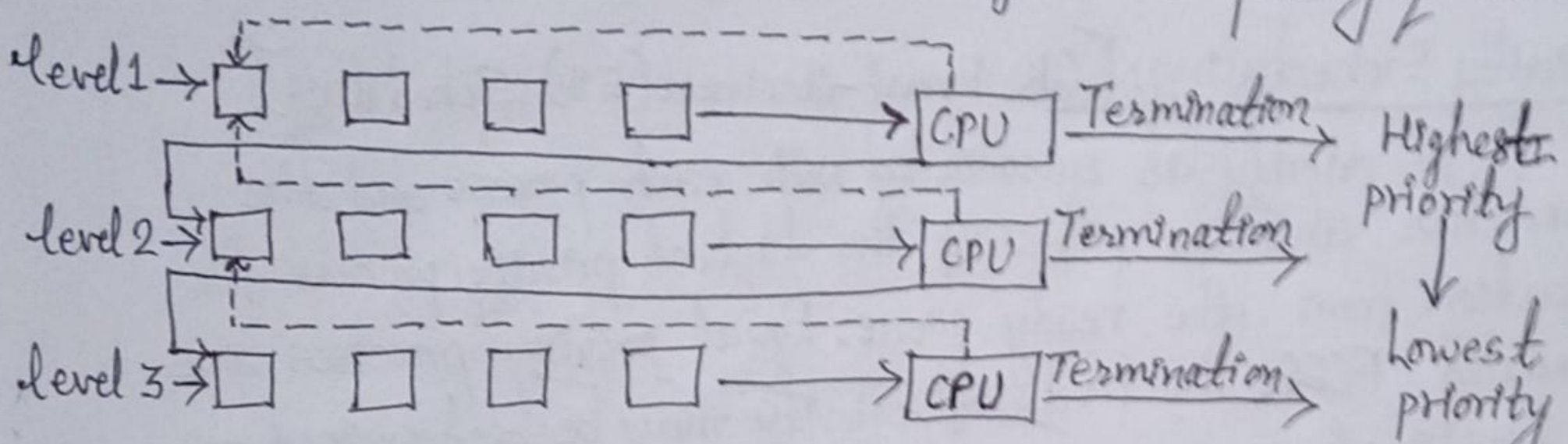


Fig. Multiple Queues

⊛. Overview of Real-Time System Scheduling:

Systems whose correctness depends on their temporal aspects as well as their functional aspects are called real-time systems. Predictability on timing constraints is its key property. These are the systems that carry real-time tasks. These tasks need to be performed immediately with a certain degree of urgency. In real-time systems, the scheduler is considered as the most important component which is typically a short term task scheduler. The main focus of this scheduler is to reduce the response time associated with each of the associated processes instead of handling deadline.

Real-time scheduling is playing an important role in cyber-physical systems (CPSs). Examples of CPSs range from small systems, such as medical equipment and automobiles, to large systems like national power grid.