Operating System

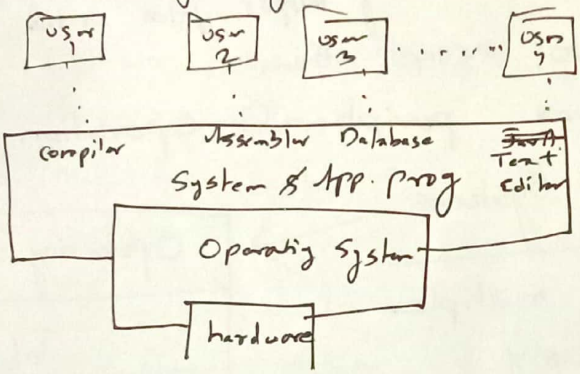
An operating system is an integrated set of programs whose major function is to - manage resources (like CPU, memory, Disks), control input and output, schedule jobs, handle errors, provide security act as an interface b/w the user and hardware.

An OS is application neutral and service specific.

Topics:

Need of OS, Computer System Components;

Abstract view of System



OS views:

- Resource allocator
- Control program - Controls execution of user program & operation of I/O devices
- Kernel - The program that executes at all times (everything else is an application with respect to the Kernel).

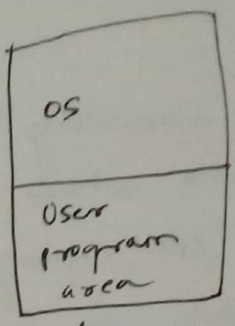
Early Systems - Bare Machine (1950s)

Hardware - expensive Human - cheap

- Structure
  - large machines run from console
  - Single user system
  - Card readers or punched cards, tape drives.
- Inefficient use of expensive resources.
  - low CPU

Batch Systems (1960s).

- Reduce setup time by batching jobs with similar requirements.
- Supervisor / Operator Control



Memory layout for batch system.

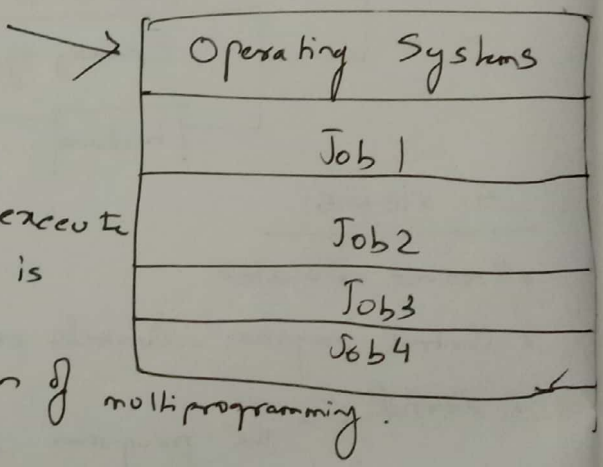
Turnaround time - It is the time to submission to finish of a program. Utilization of CPU is called throughput.

Batch System Issues: (Solns to speed up I/O).

- Offline processing
- spooling → use disk (random access devices) as large storage for reading as many input files as possible and storing output files until output devices are ready to accept them.

Spooling - Simultaneous peripheral operation Online.

Multiprogramming Systems:



- Use interrupts to run multiple program simultaneously
- When a program performs I/O execute another program till interrupt is recieved.

Time sharing: logical extension of multiprogramming.

Personal Computing Systems:

- Single user systems, portable.

Parallel Systems

Multiprocessor systems with more than one CPU in close communication

- Improved throughput, economical, increased reliability.
- Kinds:
  - vector & pipelined
  - Symetric & assymetric multiprocessing.
  - Distributed memory vs shared memory.
- Programming models
  - Tightly coupled vs loosely coupled, message-based vs shared variable.

Distributed system:

- Distribute computation among many processors.
- loosely coupled
  - no shared memory, various communication lines.
- client/server architecture.
- Advantages.
  - resources sharing
  - computation speed up
  - reliability
  - communication (email)
- Applications digital libraries, digital multimedia.

Real time systems:

- Correct system functions depends on timeliness.
- Sensor & actuators.
- Hard real-time systems-
  - failure if response time too long.
  - Used for dedicated application.
- Soft real time systems-
  - less accurate if response time is too long.
  - Useful in applications such as multimedia, virtual reality.

Computer Startup

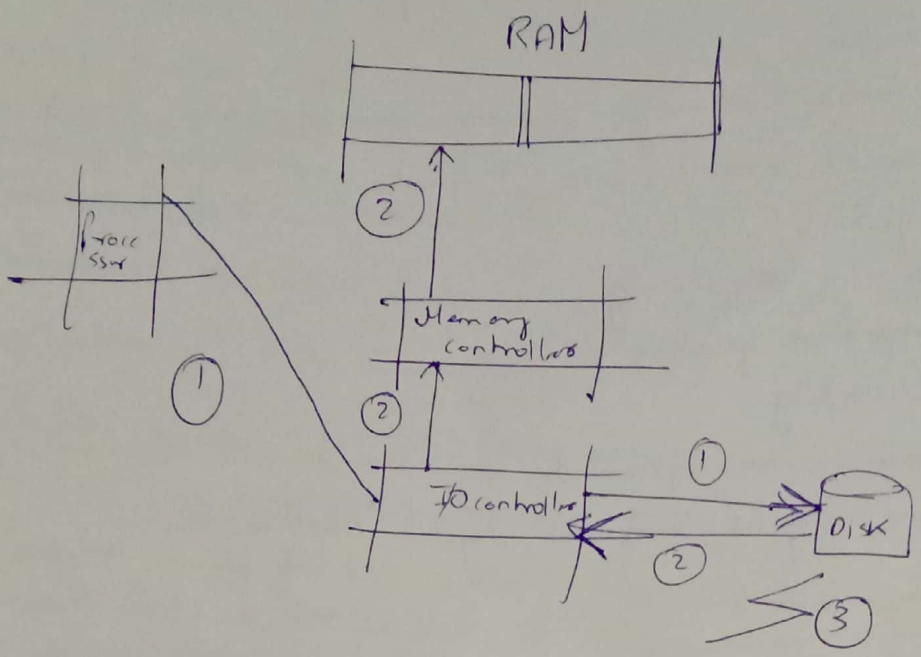
- bootstrap program is loaded at power up.
- Typically stored in ROM or EPROM generally known as firmware
- Initializes all aspects of system from CPU register to device controller to memory contents.
- loads operating system kernel and starts execution.
- OS starts executing first process, such as 'init' & waits for some event to occur.

Interrupts:

OS preserves the state of the CPU -

- Trap: Software generated interrupt - caused either by an error or a user request.

Direct Memory Access (DMA)



① A processor sends an I/O request to the I/O controller, which sends the request to the disk. The processor continues executing instructions.

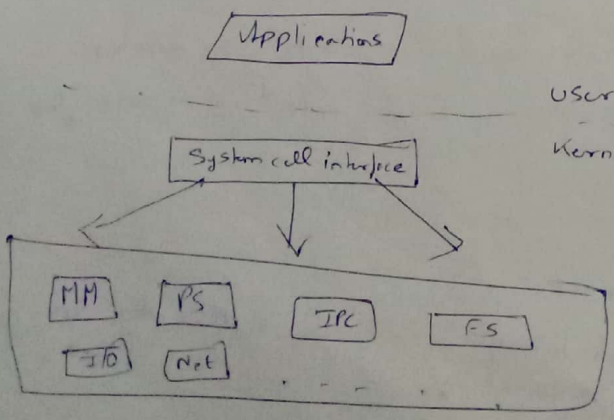
② The disk sends data to the I/O controller, the data is placed at the memory address specified by the DMA command.

③ The disk sends an interrupt to the processors to indicate that the I/O is done.

Operating System Architectures:

- Monolithic
- layered Architecture
- MicroKernel architecture.

Monolithic Architecture



MM = Memory Manager.
PS = Processor Scheduler.
IPC = Inter process communication.
F.S = File system
I/O = Input/Output manager
Net = Network manager

In monolithic OS - Every component contained in Kernel. Any component can directly communicate with any other using function call.

highly efficient.

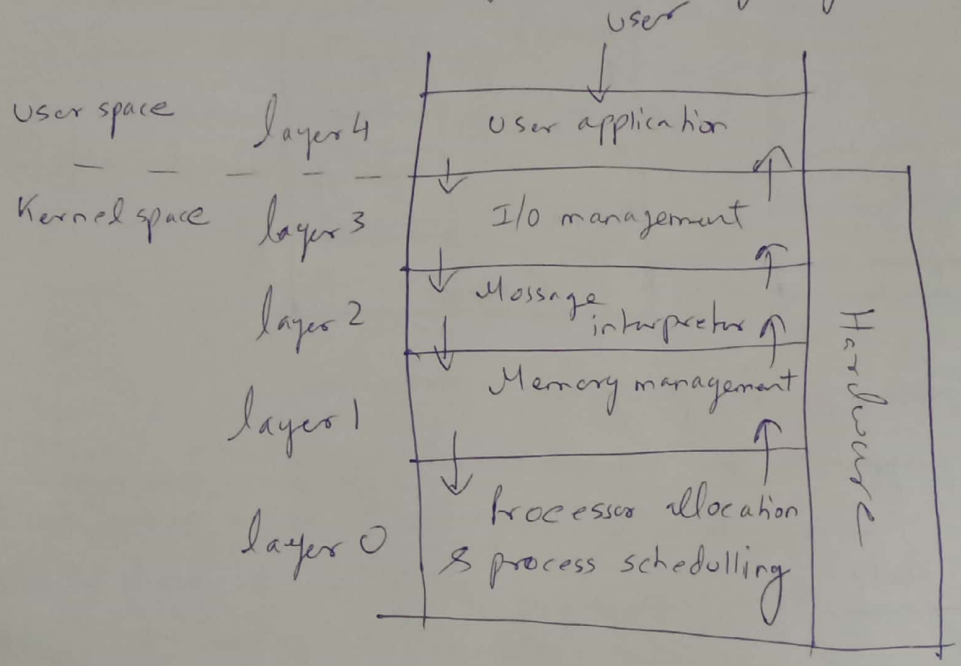
- Disadvantages is difficulty determining source of bugs & other errors.

Eg: linux, unix

MS DOS Mac OS till 8.6.

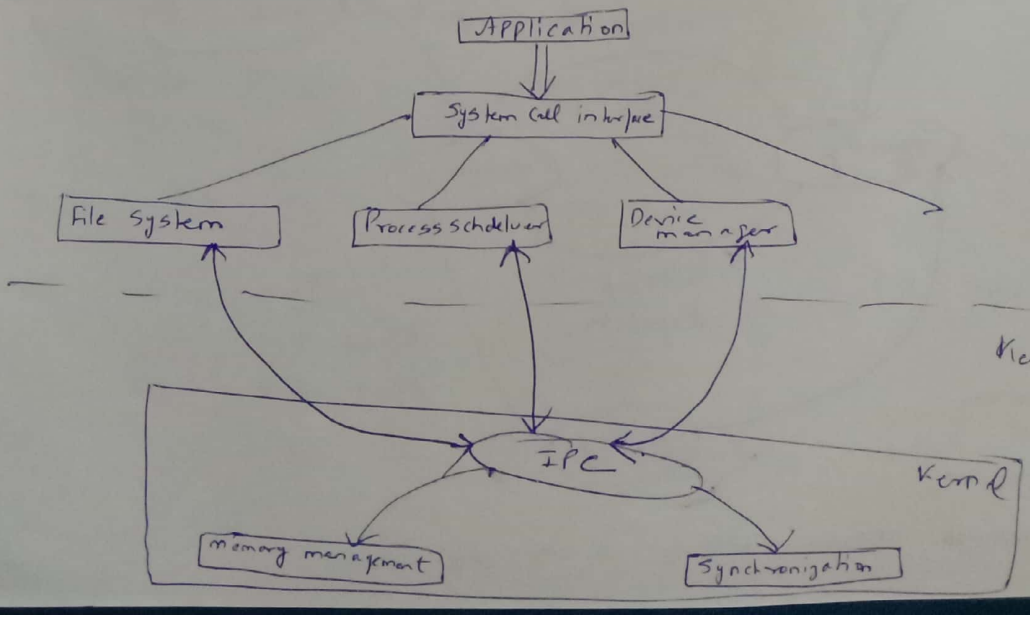
layered Architecture

layers of the operating system



Micro Kernel Architecture:

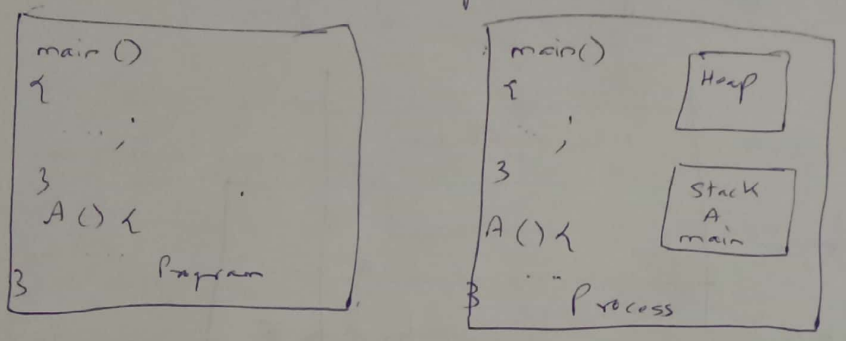
Eg- MACH
NT
EROS
MINIX



Concept of Process & Scheduling Algorithms:

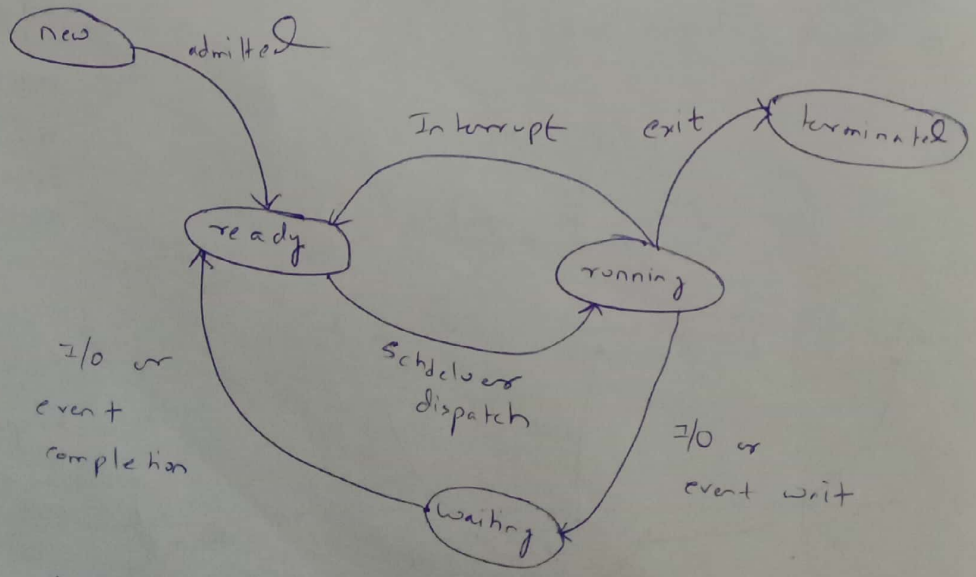
- An os executes a variety of programs.
  - batch systems - jobs
  - time shared systems - user programs or tasks.
  - job & program used interchangeably.
- Process is an entity.
  - process execution proceeds in a sequential fashion.
  - Each process has its own address space, which typically consists of a text region, data region & stack region.

Process = ? Program



- A process is a program in execution.
- A program is an inanimate entity.

Process state:



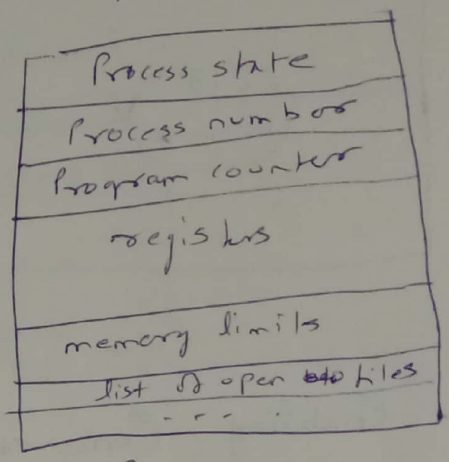
A process changes states as its executes.

Process state

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

Process control Block (PCB):

- contains information associated with each process.
- Process state - eg. new, ready, running etc.
- Process Number - Process ID
- Program Counter - add - of next instruction to be executed.
- CPU registers - general purpose registers stack pointers etc.
- CPU scheduling ~~information~~ information - process priority, Pointer.



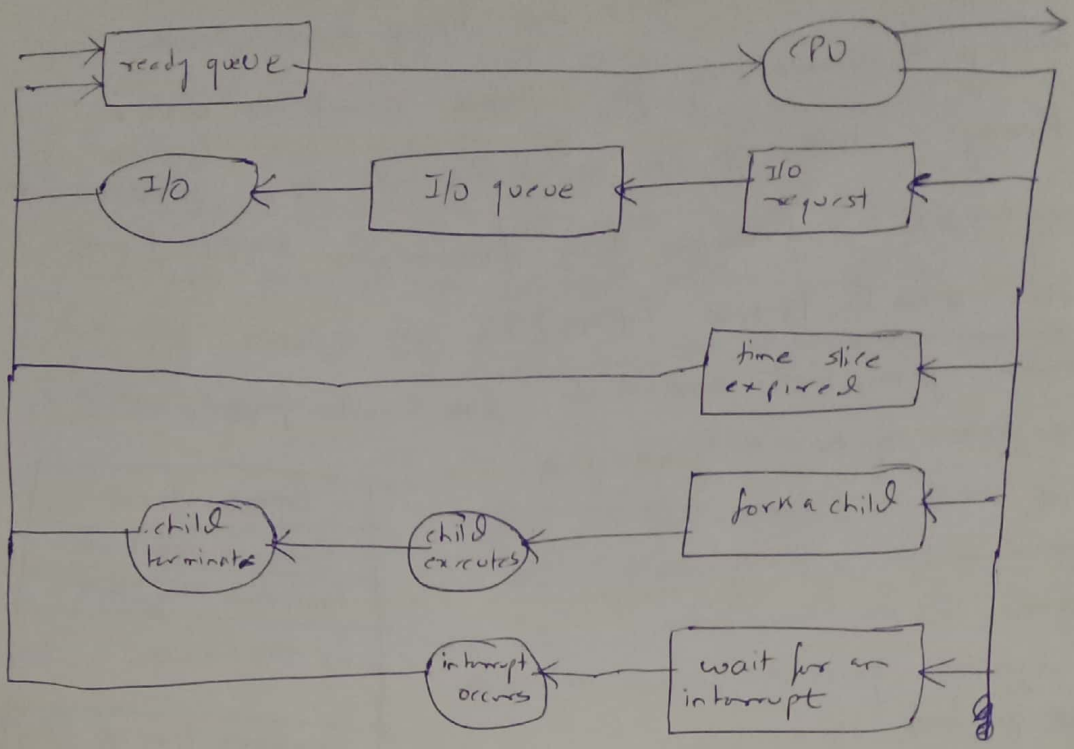
Process control block

Process scheduling Queues.

- Job queue - set of all processes in the system.
- Ready queue - set of all processes residing in main memory, ready & waiting to execute.
- Device queue - set of processes waiting for an I/O device.
- Process migration b/w the various queues.
- Queue structures - typically linked list circular list etc.

Queuing - Diagram of Process Scheduling

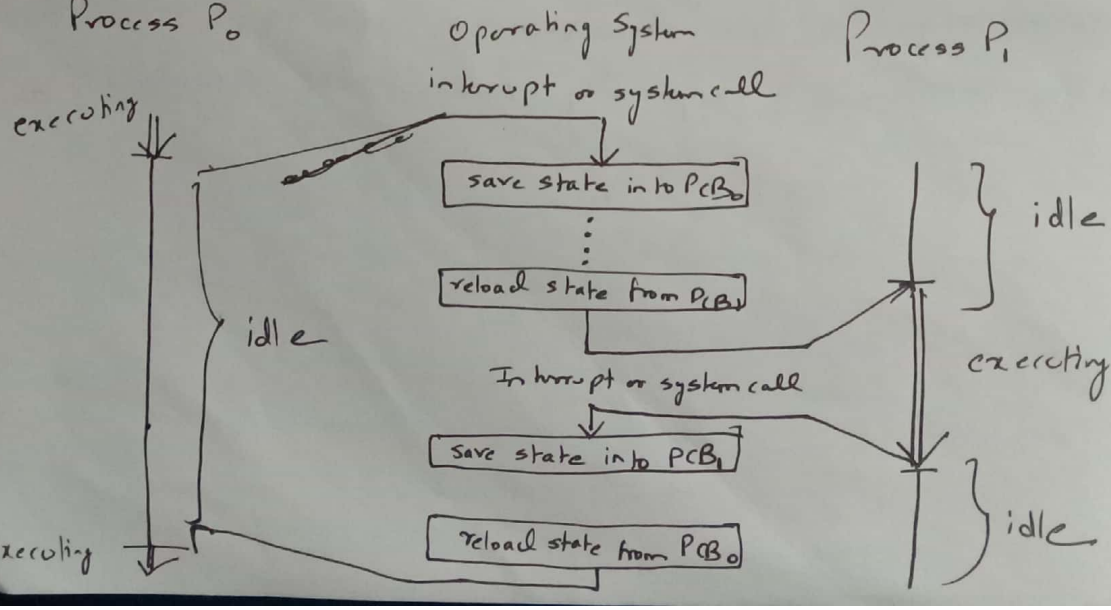
Process (PCB) moves from queue to queue.



Enabling Concurrency: Context Switch:

- Task that switches CPU from one process to another process.
  - the CPU must save the PCB state of the old process and load the saved PCB state of the new process.
- Context-switch time is overhead
  - System does no useful work while switching
- Time for context switch is dependent on hardware support (1 - 1000 microseconds).

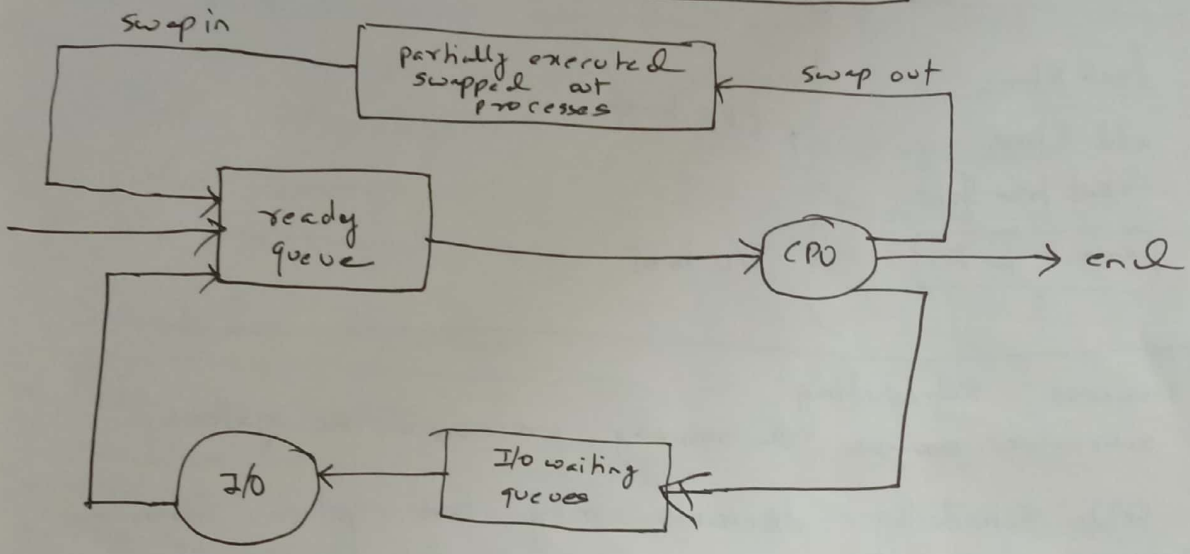
CPU switch from process to process:



Schedulers:

- long-term scheduler (or job scheduler)
  - selects which processes should be brought into the ready queue.
  - invoked very infrequently (seconds, minutes), may be slow.
  - controls the degree of multiprogramming.
- Short term scheduler (or CPU scheduler)
  - selects which process should execute next & allocate CPU.
  - invoked very frequently (milliseconds) must very fast.
- Medium term scheduler
  - swaps out process temporarily.
  - balances load for better throughput.

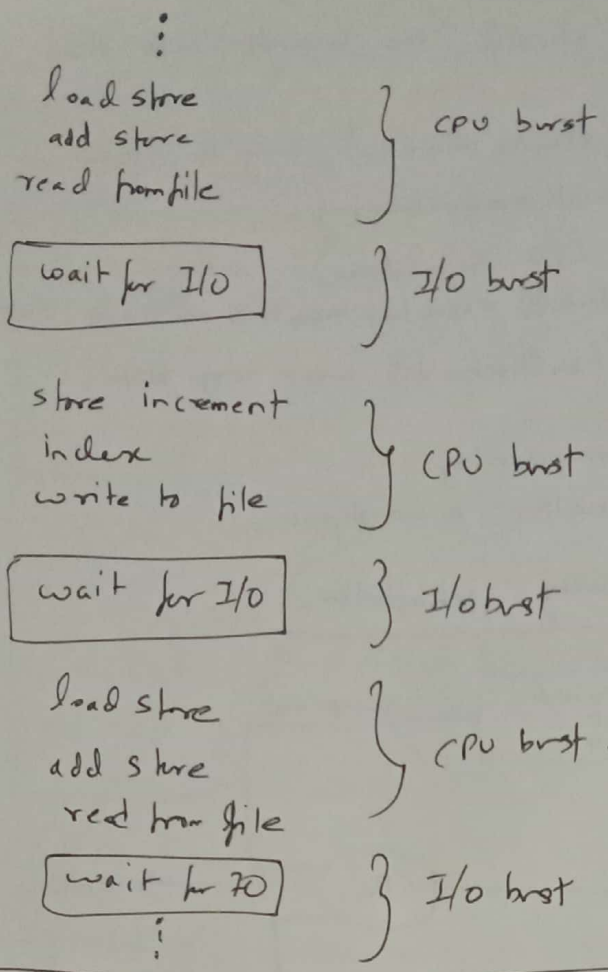
Medium Term (Time-sharing scheduler)



Process Profiles:

- I/O bound process -
  - spends more time in I/O, short CPU bursts, CPU underutilized.
- CPU bound process -
  - spends more time doing computations; few very long CPU bursts, I/O underutilized.
- The right job mix:
  - long term scheduler - admits jobs to keep load balanced between I/O & CPU bound processes.
  - Medium term scheduler - ensures the right mix (by sometimes swapping out jobs & resuming them later).

CPU - I/O Burst cycle



Process scheduling:

- scheduler manages the running processes in the system.

CPU scheduling decisions may take place when a process:

1. Switches from running to waiting state
2. Switches from running to ready state
3. Switches from waiting to ready.
4. Terminates.

- Scheduling under 1 and 4 is non preemptive
- All other scheduling is preemptive. (2 & 3)

scheduling are two types - preemptive & non preemptive.

Dispatcher

- Dispatcher module gives a control of the CPU to the process selected by the short term scheduler; this involves.
- switching context.
- jumping to the proper location in the user program to restart that program.
- Dispatch latency - time it takes for the dispatcher to stop one process & start another running.

Scheduling Criteria:

- CPU utilization - keep the CPU as busy as possible.
- Throughput - # of processes that complete their execution per time unit.
- Turnaround time - amount of time to execute a particular process
- Waiting time - amount of time a process has been waiting in the ready queue.
- Response time - amount of time it takes from when a request was submitted until the first response is produced, not output (for time-sharing environment).

Each scheduling algorithm must be evaluated from how it optimizes the following variables:

- Max CPU utilization
- Max thoroughput
- Min turnaround time.
- Min waiting time
- Min response time.

Scheduling Algorithms:

- Preemptive Scheduling.
  - Scheduler suspends a running process
  - Allows other processes to run without each process having to complete.
  - Must be careful with real time
  - Must be careful to avoid race conditions.
  - Multiuser freindly.
- Non-preemptive scheduling.
  - Run each process to completion
  - Not efficient for I/O bound applications
  - Easy to understand and implement
  - No race condition issues
  - Must prevent starvation
  - No way to guarantee turnaround.

CPU Scheduling Algorithm:

1. FCFS - In this algorithm the processes which come first will execute first. This is non-preemptive type algorithm. Ex -

Process	Arrival time (ms)	Service/burst time (ms)
P 1	0	8
P 2	1	4
P 3	2	9
P 4	3	5

find out from the given - the average waiting time and total turnaround time (tat).

Ans: waiting time for

$P_1 = 0$

$P_2 = 8 - 1 = 7$

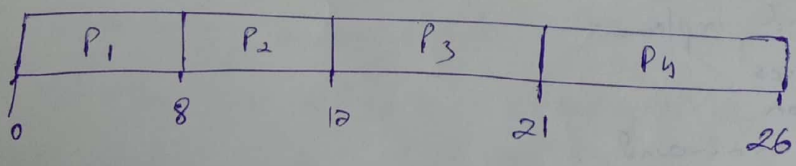
$P_3 = 12 - 2 = 10$

$P_4 = 21 - 3 = 18$

Avg. waiting time = $\frac{0 + 7 + 10 + 18}{4}$ time

= 8.75 ms.

FCFS



TAT,

$P_1 = 8$

$P_2 = 12 - 1 = 11$

$P_3 = 021 - 2 = 19$

$P_4 = 26 - 3 = 23$

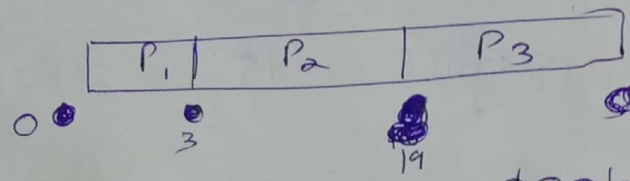
Avg. TAT = $\frac{8 + 11 + 19 + 23}{4}$ = 15.25 ms.

Q: There are

Process	Burst time (ms)
P_1	3
P_2	16
P_3	5

all the processes arrived at the 0th time.

Gantt chart:



$P_1 = 3 - 0 = 3$

$P_2 = 16 + 3 = 19$

$P_3 = 5 + 19 = 24$.

Avg. tat time $= \frac{3 + 19 + 24}{3} = \frac{46}{3} = 15.33$.

Avg. waiting time.

$P_1 = 0$

$P_2 = 3$

$P_3 = 19$

Avg waiting time $= \frac{0 + 3 + 19}{3} = \frac{22}{3} = 7.33$

Algorithm - 2 :

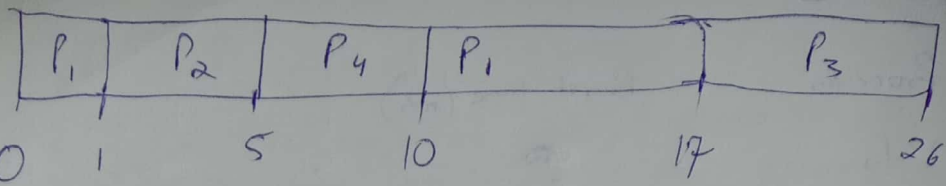
(SJF) - shortest Job first :

In this algorithm the process which has the shortest burst time will execute first. this algorithm is either preemptive or non-preemptive type.

(i). preemptive SJF :

Ex-

Process	Arrival time	Service time / Burst time
P1	0	8
P2	1	4
P3	2	9
P4	3	5



WT of

$P_1 = 10 - 1$ (0 ⇒ arrival time; 10 ⇒ execution time) $= 9$

$P_2 = 1 - 1$

$= 0$. ⇒ arrival time.

$P_3 = 17 - 2$ ⇒ arrival time $= 15$.

$P_4 = 5 - 3 = 2$

↓ arrival time.

Avg. WT $= \frac{9 + 0 + 15 + 2}{4}$

$= 6.5$ m/s.

T.T.

$P_1 = 17 - 0 = 17$

$P_2 = 1 = 1$

$P_3 = 17 - 2 = 15$

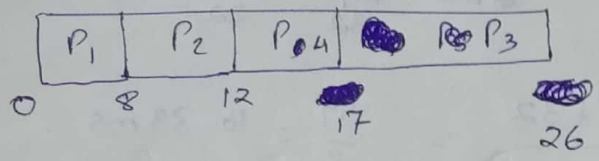
$P_4 = 5 - 3 = 2$

Avg. tat $= \frac{17 + 0 + 15 + 2}{4}$

$= \frac{18 + 17}{4} = \frac{35}{4} = 8.$ ms.

Non-Preemptive SJF:

Process	Arrival time	Burst time	CT	TAT	WT
P_1	0	8	8	8	0
P_2	1	4	12	11	7
P_3	2	9	26	24	15
P_4	3	5	17	14	9



Avg WT = $\frac{31}{4} = 7.75$ ms

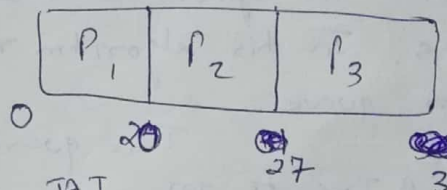
Avg TAT = $\frac{57}{4} = 14.25$ ms Ans:

FCFS

Process	Arrival time	Burst time (ms)	CT,	TAT,	WT,
P_1	0	20	20	20	0
P_2	0	7	27	27	20
P_3	0	5	32	32	27

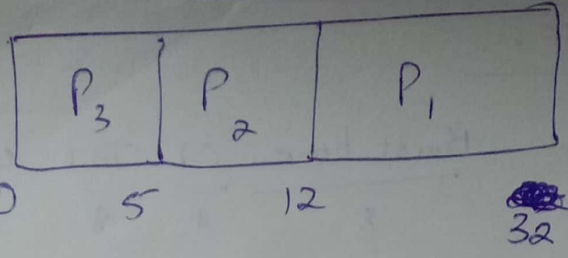
(i) $P_1\ P_2\ P_3$ (ii) $P_3\ P_2\ P_1$

(i).



Avg TAT = $\frac{20+27+32}{3} = \frac{79}{3} = 26.33$.

Avg WAT = $\frac{0+20+27}{3} = \frac{47}{3} = 15.67$ ms



Process	Arrival Time	Burst Time	CT	TAT	WT
P_3	0	5	5	5	0
P_2	0	7	12	12	5
P_1	0	20	32	32	12

$$\text{Avg TAT} = \frac{12+5+32}{3} = \frac{49}{3} = 16.33 \text{ ms}.$$

$$\text{Avg WT} = \frac{0+5+12}{3} = \frac{17}{3} = 5.67 \text{ ms}.$$

SJF

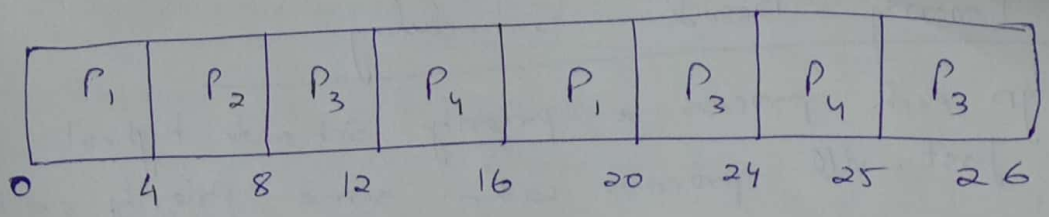
The disadvantage of this algorithm is the problem to know the length of the time for which the CPU is needed by a process. A prediction algorithm may be used to predict the amt of time for which CPU may be required by a process.

Round Robin Algorithm (RR)

The RR scheduling algorithm is designed specially for time sharing system. It is similar to FCFS algorithm, but preemption is added to switch between processes. A small unit of time called time slice or time quantum is defined where the range is kept b/w 10 to 100 ms. In this algorithm ready queue is assumed to be circular queue.

Ex Time quantum = 4 ms.

Process	Process Arrival Time	Service Time	CT	TAT
P_1	0	8	20	20
P_2	1	4	8	7
P_3	2	9	26	24
P_4	3	5	25	22



W.T for

$P_1 = 16 - 4 - 0$

$= 12$ ms.

$P_2 = 4 - 1$

$= 3$ ms

$P_3 = 25 - (2+4+4)$

$= 15$ ms

$P_4 = 24 - (3+4)$

$= 17$ ms.

Avg wt $= \frac{12+3+15+17}{4}$

$= \frac{47}{4} = 11.75$ m/sec.

TAT for

$P_1 = 20$ ms

$P_2 = 7$ ms.

$P_3 = 26 - 2 = 24$ ms

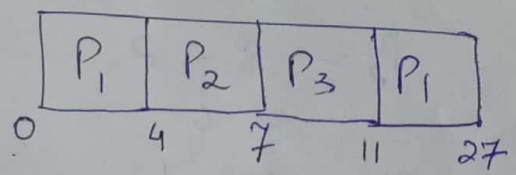
$P_4 = 25 - 3 = 22$ ms

Avg TAT $= \frac{20+7+24+22}{4}$

$= \frac{73}{4} = 18.25$ ms.

Q: Time quantum = 4 ms

Process	Arrival time	Burst time	CT	TAT	WT
P_1	0	20	27	27	7
P_2	0	3	7	7	4
P_3	0	4	11	11	7



Avg WT $= \frac{7+4+7}{3} = \frac{18}{3} = 6$ ms

Avg TAT $= \frac{27+7+11}{3} = \frac{45}{3} = 15$ ms.

Priority Based Scheduling:

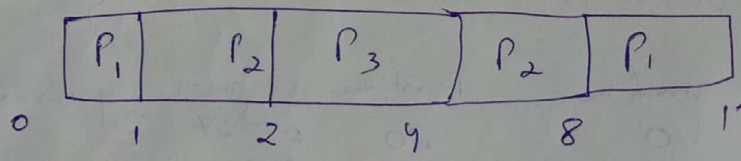
- Assign each process a priority. Schedule highest priority first. All processes within same priority are FCFS.
- Priority may be determined by user or by some default mechanism. The system may determine the priority based on memory requirements, time limits or other resource usage.
- Starvation occurs if a low priority process never runs.
  solution: build aging of low priority processes.

Priority scheduling is also of two types - preemptive & non-preemptive priority scheduling.

for priority 1 > 2 > 3

Process	Burst time	Priority	Arrival time
P_1	10	3	0
P_2	5	2	1
P_3	2	1	2

Preemptive Priority scheduling:



C.T →

$P_1 = 7$ ms

$P_2 = 4 - (1+1)$ ms

$= 2$ ms

$P_3 = 2 - 2$

$= 0$ ms.

Avg WT $= \frac{7+2+0}{3}$

$= \frac{1}{3} = 3$ ms.

TAT

$P_1 = 17 - 0$

$= 17$

$P_2 = 8 - 2$

$= 6$

$P_3 = 4 - 1$

$= 3$

Avg TAT $= \frac{17+6+3}{3}$

$= \frac{26}{3} = 8.6$ ms

3)26(8.
24
2.

Highest - Response - Ratio - Next scheduling (HRRN)

- Improve upon SJF scheduling.
- It calculates dynamic priority according to the formula

$$\text{priority} = \frac{\text{waiting time} + \text{service time}}{\text{service time}}$$

- Considers how long process has been waiting.
- Prevents indefinite postponement.
- still non-preemptive.

Selfish Round Robin Scheduling Algorithm:

- A variant of round robin scheduling algorithm.
- In selfish RR, there is a maximum limit on the number of processes that can be placed in the round robin queue (including the process being executed by the CPU).
- After that maximum is reached, newly entering processes are placed on a holding queue.
- Processes in the holding queue do not get any time slice of the CPU
- when a process in the round-robin queue completes and leaves the system, the oldest process in the holding queue is allowed to enter the round robin queue.
- It has 2 queues -
  - Holding & Active queues.
  - Process enters into holding queue
  - Increases priority as process ages.
  - moves to active queue when it reaches priority of other processes in active queue.
  - RR scheduling will occur in active queue.

Assignment:

Compare & discuss the FCFS, SJF, priority, round robin, HRNN, Selfish RR.

Process Synchronization

Two types of process - Dependent & Independent

problem: concurrent execution of the dependent process why?

Producer

```
while (true){
  // produce an item & put in nextproduce
  while (count == Buffer_size)
    ; // do nothing
  buffer[in] = nextproduced;
  in = (in+1) % Buffer_size;
  count++;
}
```

Consumer

```
while (true){
  while (count == 0)
    ; // do nothing
  nextconsumed = buffer[out];
  out = (out+1) % Buffer_size;
  count--;
  // consume the item in nextconsumed
```

count++ is implemented as

I_1 = register1 = count

I_2 = register1 = register1 + 1

I_3 = count = register1

---

count-- is implement as

I_4 = register2 = count

I_5 = register2 = register2 - 1

I_6 = count = register2

I_1 I_2 I_4 I_5 I_3 I_6

register1 = 5

register1 = 5+1

register2 = 5

register2 = 5-1

count = 6

count = 4

A situation where several processes access and manipulate the same data concurrently and the outcome of the execution depends on the particular order in which the access takes place, is called race condition.

Critical Section Problem:

Each process has a segment of code, called critical section, in which the process may be changing there common variables, updating a table, writing a file and so on.

Solution to critical section Problem:

1. Mutual exclusion - If process P_i is executing in its critical section, then no other processes can be executing in their critical section.
2. Progress - If no process is executing in its critical section and there exist some process that wish to enter there critical section, then the selection of processes that will enter the critical section next cannot be postponed indefinitely.
3. Bounded waiting - A bound must exist on the number of times that other processes are allowed to enter their critical sections after the process has made a request to enter its critical section and before that request is granted.

Two Process solutions:

Algorithm 1 : I finished with it, now you've it.

This algorithm applies to a set of two processes (P_0, P_1). Both are co-operating thorough a shared integer variable "Turn".

- Turn == 0, P_0 executes in CS
- Turn == 1, P_1 executes in CS

```
int turn = 0; // initial value of turn can be set
                to 0 or 1
```

```
P0 : do
   { while (turn == 1);
   // keep looping as long as
     turn equals 1
     <CS>
     turn = 1; // enable P1 to enter
                 CS
     < remainder section > } while (1);
```

```
P1 : do { while (turn == 0)
// keep looping as long as turn
equals 0
   <CS>
turn = 0; /enables P0 to enter CS
   <remainder section>
   } while (1);
```

Now let us see wether it satisfies all the three requirements of a CS solution :

(1) mutual Exclusion : At a time the value of the turn will be either 0 or 1. when turn = 0, then P_1 cannot enter its critical section & when turn = 1, then P_0 cannot enter into its critical section. Thus, at a time one co-operating process can enter its critical section. So requirement of mutual exclusion is satisfied.

(2) Progress : Consider a situation that initially turn is set to zero but P_1 intends to enter its critical section earlier than P_0. It cannot till P_0 enters first, exits and changes the value of turn to 1, only then P_1 can enter. So the requirement of progress is not satisfied.

(3) Bounded waiting : Yes bounded waiting is preserved, as it causes strict alternation.

Algorithm 2 : let me have it, after you finish it.

```
typedef enum boolean (false, true);
boolean flag[2]; // initial both set to false
P0 → do { flag[0] = true; // intend to enter CS
    while (flag[1]); // keep looping as long as flag[1] is true
    <CS>
    flag[0] = false;
    <remainder section>
      while (1);
    }
P1 : → do { flag[1] = true; // intend to enter CS
    while (flag[0]); // keep looping as long as flag[0] is true
    <CS>
    flag[1] = false; // exiting from CS
    <remainder section>
      while (1);
    }
```

Algo - 3 (Peterson's Algorithm)

I need it, but first you have, if you also need

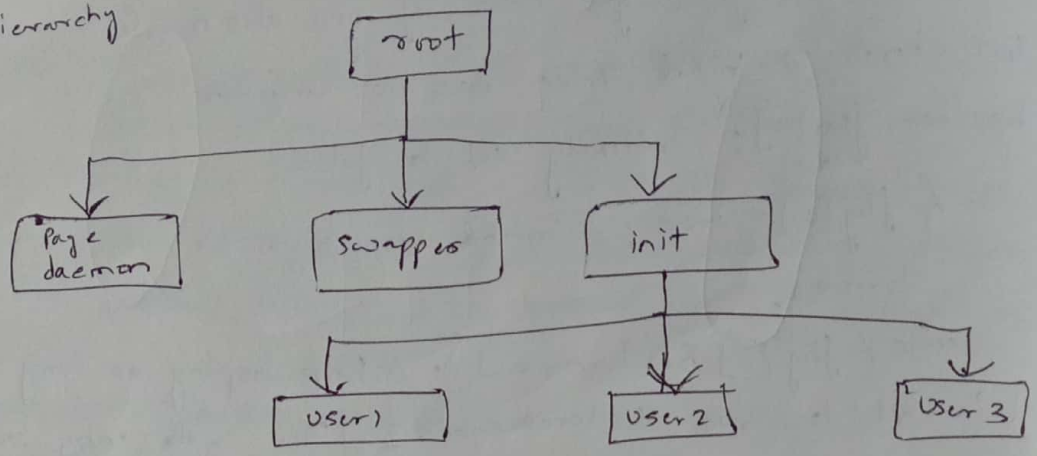
```
int turn; // initial value does not matter.
boolean flag[2]; Initially set to false
P0: do { flag[0] = true; // Intend to enter CS
         turn = 1;
         while (flag[1] && turn == 1); // Keep looping as long as
flag[1] is set to true & turns equals 1. This is the entry section.
         <CS>
         flag[0] = false; // exit from CS
         <RS>
}
P1: do { flag[1] = true; // Intend to enter CS
         turn = 0;
         while (flag[0] && turn == 0); // Keep looping as long
                                       as flag[0] is set to true & turn equal 0.
         <CS>                          This is the entry section.
         flag[1] = false; // exit from CS
         <RS>
}
```

Process Creation:

1) Processes is created & deleted dynamically.
2) Process which creates another process is called a parent process, the created process is called child process.
3) Therefor the result is a tree of processes.
4) Resources required when creating process. Ex - CPU time, files, memory, I/O devices etc.
5) Resource sharing - Ex - (i) Parent and children share all resources
(ii) Children share subset of parents resources.

Process Creation

Fig: Unix process hierarchy



6) Execution - (i) Parent & child execute concurrently.
(ii) Parents wait until child has terminated.

7) Address space: (1) Child process is duplicate of parent process
(2) Child process has a program loaded into it.

Process Termination:

1) Process executes last statement and ask the operating system to delete it (exit).
- Output data from child to parent
- Process resources are deallocated by OS.

2) Parent may terminate execution of the child processes.
- Child has ~~execut~~ exceeded allocated resources.
- Task assign to child is no longer required
- Parent is exiting. OS does not allow child to continue if parent terminates and therefore cascading termination of processes.

Co-operating Process:

1. Concurrent Processes can be independent or co-operating process. Independent processes cannot affect or be affected by the execution of another processes. On the other hand, the co-operating processes can effect or be effected by the execution of another process.
2. The main advantages of process co-operation are information sharing, computation speed up, modularity and convenience. for ex- editing, printing and compiling.
3. The concurrent execution requires process communication & process synchronization

Inter-process Communication (IPC):

Inter process communication can be done via shared memory and via messaging system. The shared memory can be accomplished by mapping addresses to common DRAM using read & write memory operation. further, in restoring messaging system, processes communicate without to shared variable. Send() and reciever() messages are used. Messaging system and shared memory are not mutually exclusive.

Co-operating Processes via message passing:

The IPC facility provides two operations -

(i) Send (message)

(ii) Reciever (message)

If processes P & Q wish to communicate, they need to establish a communication link b/w them and exchange messages via send or reciever function.

Direct Communication:

(1) Sender & Reciever processes must name each other explicitly

(a) Send (P, message) → Send a message to process P.

(b) receive (Q, message) → recieve a message from process Q.

(2) Properties of communication link -

(a) links are established automatically.

(b) A link is associated with exactly one pair communicating process.

(c) link may be unidirectional, usually bidirectional

Indirect Communication:

(1) Messages are directed to and recieved from mail-boxes (also called ports) -

(a) unique id for every mail box

(b) Processes can communicate only if they share a mail box.

(i) Send (A, message) → Send messages to mailbox A.

(ii) Reciore (A, messge) → reciere messages from mail box A.

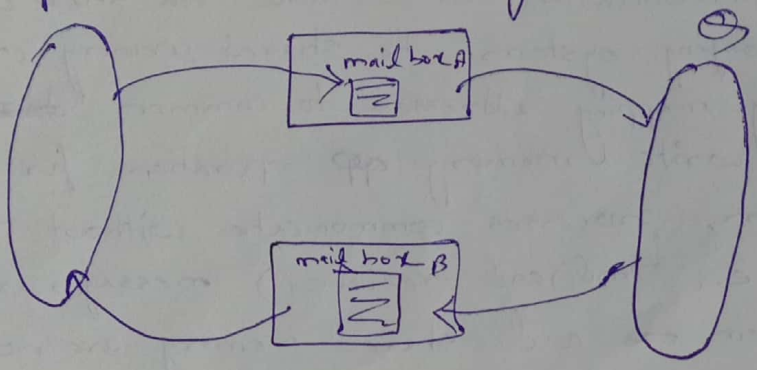
(2) Properties of communication link -

(a) link established only if processes share a common mail box.

(b) link can be associated with many sub processes.

(c) links may be unidirectional or bidirectional.

Indirect communication using mailboxes.



Q: What is the main issue associated with the mail box while using the indirect communication?

Semaphore

Semaphore is a synchronization tool that does not require busy waiting.
It is a operating system tool. Basically, semaphores are two types -

(1) binary semaphore
(2) Counting semaphore

The binary semaphore has only two value 0 and 1 while the value of counting semaphore varies unlimited domain value.
Semaphore S is an integer variable.
There are two standard operation which only modifies S

(i). wait () → P()
(ii) signal() → V()

```
wait (S){
    while (S<=0)
        ; //no-op
    S--
}

signal (S){
    S++;
}
```

The wait() and signal() should be atomic (indivisible).
Binary Semaphore is also known as mutex locks.

```
Semaphore S; // initialized to 1.
P1 →  wait(S);
      Critical section;
      signal(S);
```

Semaphore Implementation:

Must gurantee that no two processes can execute wait() and signal() on the same semaphore at the same time. Thus, the implementation becomes the critical section problem where the wait() & signal() code are placed in the critical section.

Dining Philosopher:

Consider 5 Philosopher who spend their lives thinking & eating. The philosopher share a circular table surrounded by five chairs, each belonging to one philosopher. In the center of the table there is a bowl of rice, and the table is laid with five single spoon. When a philosopher thinks, she does not interact with her colleagues. from time to time, philosopher gets hungry and tries to pick up two spoons that are closest to her (left & right).

A philosopher may pick up only one chopstick at a time. Obviously, she cannot pick up a chopstick that is already in the hand of the neighbour.

When a hungry philosopher has both her spoon at the same time, she eats without releasing her spoon. When she is finished eating, she puts down both of her chopsticks and starts thinking again.

One simple solution is to represent each chopstick with a semaphore. A philosopher tries to grab a chopstick by executing a wait() on that semaphore. She releases her chopstick by executing the signal() operation on the same semaphore. Thus, the shared data are -

```
Semaphore chopstick[5];
```

where all the element of the chopstick are initialized to 1.

```
do {
    wait ( chopstick [i]);
    wait ( chopstick [(i+1) % 5];
    .....
    // eat
    signal ( chopstick [i]);
    signal ( chopstick [(i+1) % 5]);
    // think
} while ( True);
```

Fig → The structure of philospher i

Although this solution gurantees that no two neighbours are eating simultaneously. Suppose that all five philospheres become hungry simultaneously and each grabs her left chopstick. All the elements of chopstick will now be equal to 0. When each philospheres tries to grab her right chopstick, she will be delayed forever.

Several possible remedies to the deadlock problem are —

(i) Allow atmost four philospheres to be silting simultaneously at the table

(ii) Allow a phliospher to pick up her chopsticks only if both chopsticks are availaible.

(iii) And odd philospher picks up first her left chopstick and then her right chopstick whereas an even philospher picks up her right chopsticks and then her left chopstick.

(Bounded philospher - ~~reader~~ producer/consumer).

Deadlock

In a multiprogramming environment, several processes may compete for a finite number of resources. A process request resources, if the resources are not available at that time, the process enters a waiting state. Sometimes, a waiting process is never again able to change state, because the resources it has requested are held by other waiting processes. This situation is called deadlock.

Process utilizes in the following sequences –

i) Request
ii) Use
iii) Release

Necessary Conditions for deadlock:

(1) Mutual Exclusion: Atleast one resource must be held in a non-sharable mode, that is only one process can use the resource at a time.

(2) Hold and wait: A process must be holding atleast one resource and waiting to acquire additional resources that are currently being held by other processes.

(3) No-preemption: Resources cannot be preempted, that is a resource can be released only volontarily by the process holding it.

(4) Circular wait: A set $\{P_0, P_1, \ldots P_n\}$ of waiting processes must exist such that P_0 is waiting for a resource which is held by P_1, P_1 is waiting for a resource held by P_2, ..., P_{n-1} is waiting for resources held by P_n, and P_n is waiting for a resource P_0.

Safe State

1) When a process request an available resource, system must decide if immediate allocation leaves the system in a safe state.

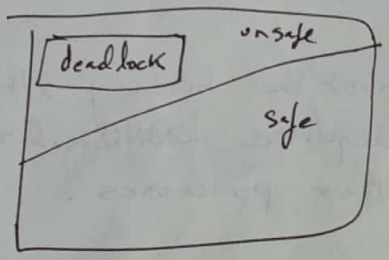
2) System is in safe state if their exist a sequence $[P_1, P_2, \ldots P_n]$ of all the processes in the system such that for each process P_i, the resources that P_i can still request can be satisfied by currently available resources + resources held by all the P_j, with $j < i$.

3) (a) That is if P_i resource needs are not immediately available, then P_i can wait until all P_j have finished.

(b) When P_j is finished, P_i can obtain needed resources, execute, return allocated resources & terminate.

(c) When P_i terminates, P_{i+1} can obtain its immediate resources, and so on

Safe, Unsafe, Deadlock state



Data structure for Banker's Algorithm:

Let n = number of process and m = number of resources type.

1) Available: Vector of length m if available. If available $[j] = K$, then K instances of resource type R_j are available.

2) Max: An $m \times n$ matrix defines the maximum demand of each process. If $max[i,j] = K$, then process P_i may request atmost K instance of resource type R_j.

3) Allocation: An $m \times n$ matrix defines the number of resources of each type currently allocated to each process. If $allocation[i,j] = K$, then process P_i is currently allocated to K instance of resource type R_j

4) Need: An n x m matrix indicates the remaining resource process need of each process.

If need [i, j] = k, then process P_i may need k more instances of resource type R_j to complete its task.

Need [i, j] = Max [i, j] - Allocation [i, j].

Safety Algorithm:

- There are four steps for safety algorithm:

1) let work and finish be vectors of length m and n respectively, initialize work = available.

Finish [i] = false for i = 0, 1 .... n-1,

2) find an index i such that both ~~finish~~ i

(a) finish [i] == false

(b) Nreq_i >= work

If no such i exist then go to step 4.

(3) work = work + Allocation_i ;

Finish [i] = true

go to step 2.

4. If finish i == true for all i, then the system is in safe state.

These algorithm may require an order of $m \times n^2$ operations to determine wether a state is safe.

Resource Request algorithm:

~~These algorithm~~ Next we describe the algorithm for determining wether requests can be safely ~~granted~~ or not.

let request_i be the request vector for process P_i.

If ~~request[i]~~ $request_i[j] == K$, then process P_i wants K instances of resource type R_j.

When a request for resources is made by process P_i, the following actions are taken -

1) If $request_i <= need_i$ go to step 2, otherwise raise an error condition since process had exceeded its maximum claim.

2) If $request_i <= available$, go to step 3, otherwise P_i must wait since the resources are not available.

3) Pretend to allocate requested resources to P_i by modifying the state as follows available = available - $Request_i$.

$Allocation_i = Allocation_i + request_i$

$Need_i = Need_i - request_i$

If the resulting resource allocation state is saved, the transaction is completed, and process P_i is allocated its resources.

Example of Banker's Algorithm:

- There are 5 processes from P_0 to P_4
  3 resource type:
  A (10 instances), B (5 instances), and C (7 instances).
- Snapshot at time T_0:

	Allocation	Max	Available	Need
	A B C	A B C	A B C	A B C
P_0	0 1 0	7 5 3	3 3 2	7 4 3
P_1	2 0 0	3 2 2		1 2 2
P_2	3 0 2	9 0 2		6 0 0
P_3	2 1 1	2 2 2		0 1 1
P_4	0 0 2	4 3 3		4 3 1