Process Management

(Processes & threads)

→ How a program is developed.

→ ① We write the program in some high level languages (C, C++ | Java ---)

② But computers understand only binary codes, so program has to be converted to binary code.

We can use compiler to compile the program. (which helps in converting high level language prog into machine understandable format (binary))

③ So its not enough to have only Binary code for a program to execute.

It has to to be loaded into the memory. So a program needs some resources of Comp. (OS allocates resources to program.)

Then program begins execution.

→ A program which is written remains as a passive entity (idle without doing anything) untill it ~~becom~~ starts / begins execution.

But at the moment it starts execution we call that program as a process.

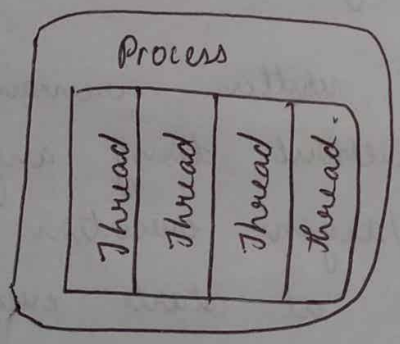
Process → a program that ~~can~~ is in execution is called a process.

In older computers it is possible for only 1 process & a program to execute at a time,
but in today's comp we can execute multiple processes, programs at same time, & even 1 single program may have many processes associated with it.

→ Threads → it is a ~~bis~~ basic unit of process that is in execution.
(A thread is a unit of execution within a process. A process can have anywhere just from thread to many threads).

→

Earlier	Now
One process had only one thread.	→ Single process can have many threads (/units of exec)
→ One process /prog can execute at a time	→ many processes / programs can execute at 1 time



→

Process State

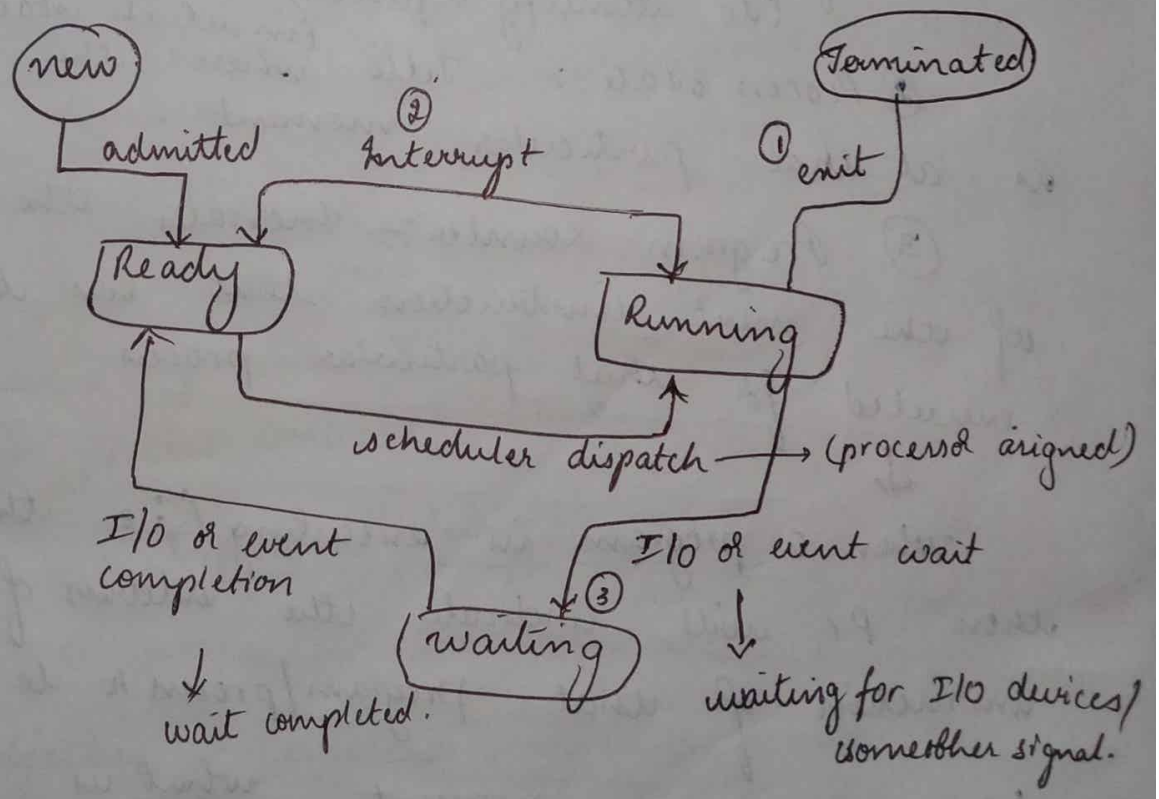
As a process executes it changes its states
→ The state of a process is defined in part by the current activity of that process.

(what the process currently doing)
↳ state of that process.

Each process may be in one of the following states

---

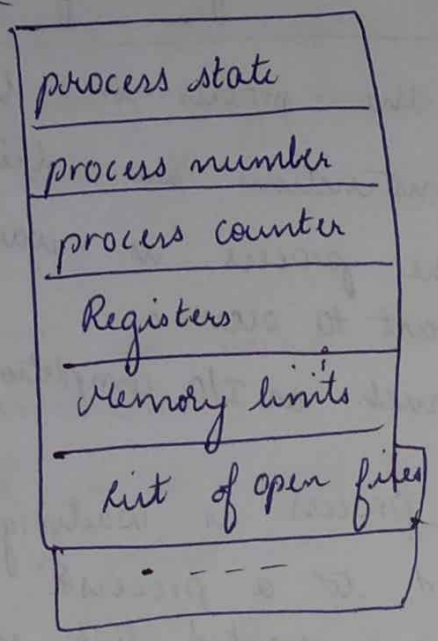
① new → the process is being created
② Running → Instructions are being executed.
③ Waiting → The process is waiting for an event to occur.
(such as I/O completion / reception of signal) (waiting for signal to be received)
④ Ready → The process is waiting to be assigned to a processor
(process is created but not yet started running) → waiting to be assigned.
⑤ Terminated → The process has finished execution.



Process Control Block

Each process is represented in the OS by a process control block
(PCB) → task control block.

Eg:-



what we have in a PCB

① Process id:- Process num/id shows the unique id/num of a particular process.
(to identify a process)

② Process state :- Tells (in which state) where the process is at that particular moment.

③ Program counter:- Indicates the address of the next instruction that has to be executed for that particular process

↓

when a program is executing (i.e the process) then P.C will indicate the address of next instruction of that program/process to be executed
(At a particular moment what is the address of next line to be executed)

④ CPU registers:- Tells us the registers that are being used by a particular process.

(diff kind of registers like index reg, stack pointers, general purpose registers ---)

⑤ CPU scheduling info :- CSI has priority of processes & has pointer to scheduling queue & other scheduling parameters.

→ determines order of execution of processes.

→ ⑥ Memory management info :- represents the memory that is used by a particular process. (different aspects of memory & info)

→ ⑦ Accounting info :- Keeps an account of certain things like resources that are being used by particular process (for exec) (CPU, time, memory ---)

⑧ Input / Output Status info :- represents the input output devices that are being assigned to a particular process.

These above all represent different aspects of a particular process. & they are in a block known as process controlled block.

Process Scheduling (PS)

→ PS

→ The objective of multiprocessing is to have some process running at all times to maximize CPU utilization.

→ The objective of time sharing is to switch the CPU among processes so frequently that users

can interact with each program while it is running.

→ How are these 2 objectives accomplished!?

To meet these objectives the process scheduler selects an available process (possibly from a set of several available processes) for prog execution on CPU

→ what happens if a process is selected by Process scheduler? Will it be given comp time to complete the execution? what if another process with high priority comes in middle? all such qns answered by P.S

For a single processor system, there will never be more than 1 running process.
If there are more no. of processes, the rest will have to wait untill CPU is free & can be rescheduled.

To help this process scheduling we have Scheduling Queues.

→ 2 types

Job queue

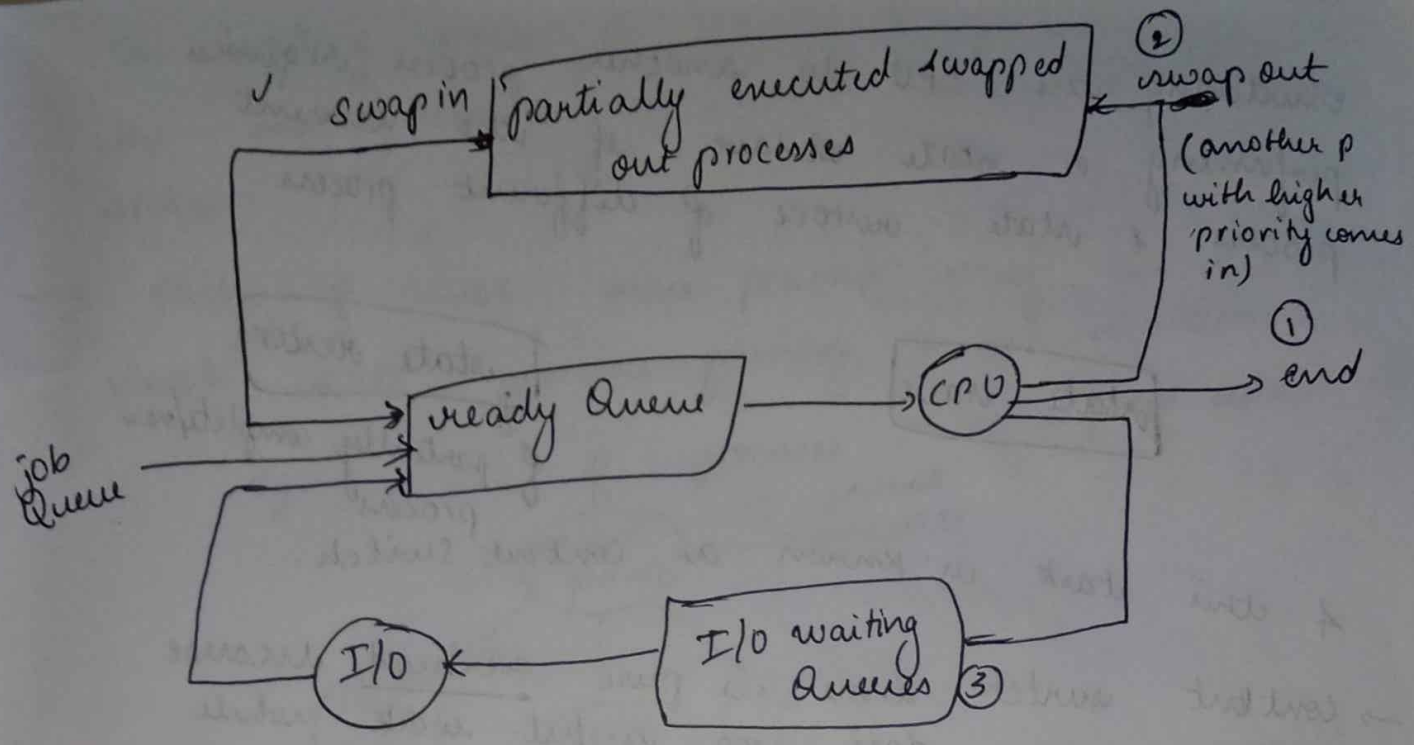
As a process enters the system it is put into a job queue which consists of all processes in the system.

Ready Queue

The processes that reside in main memory & are ready & waiting to execute are kept on a list called ready Queue.

↓

Once a process in ready Queue gets access to execution then what happens!?



Content Switch

→ Interrupts cause the OS to change CPU from its current task & run a kernel routine

→ such operations happen frequently on general purpose systems.

→ when an interrupt occurs, the system needs to save the current content of the process currently running on CPU so that it can restore that content when its processing is done, essentially suspending the process ~~when~~ & then resuming it.

→ The content is represented in PCB of process
↓
process control block

consist of current state, info of process (as when process resumes it has to know where did it stop at particular moment)

switching the CPU to another process requires performing a state save of the current process & state restore of different process.

state save

state restore of partially complete/new process.

& this task is known as Context Switch.

→ Context switch time is pure overhead because the system does no useful work while switching.

→ Its speed varies from machine to machine, depending on memory speed, the no. of registers that must be copies & existance of special instructions (such as single instruction to load/restore all registers)

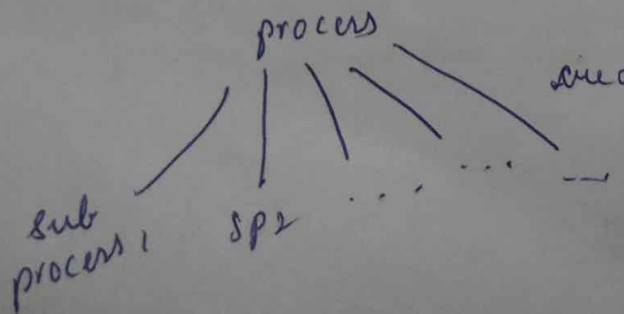
→ Typical speeds are few milliseconds.

→ when switching of processes is going on no process is being executed (i.e no useful work done)

Operations on Processes

(Process creation)

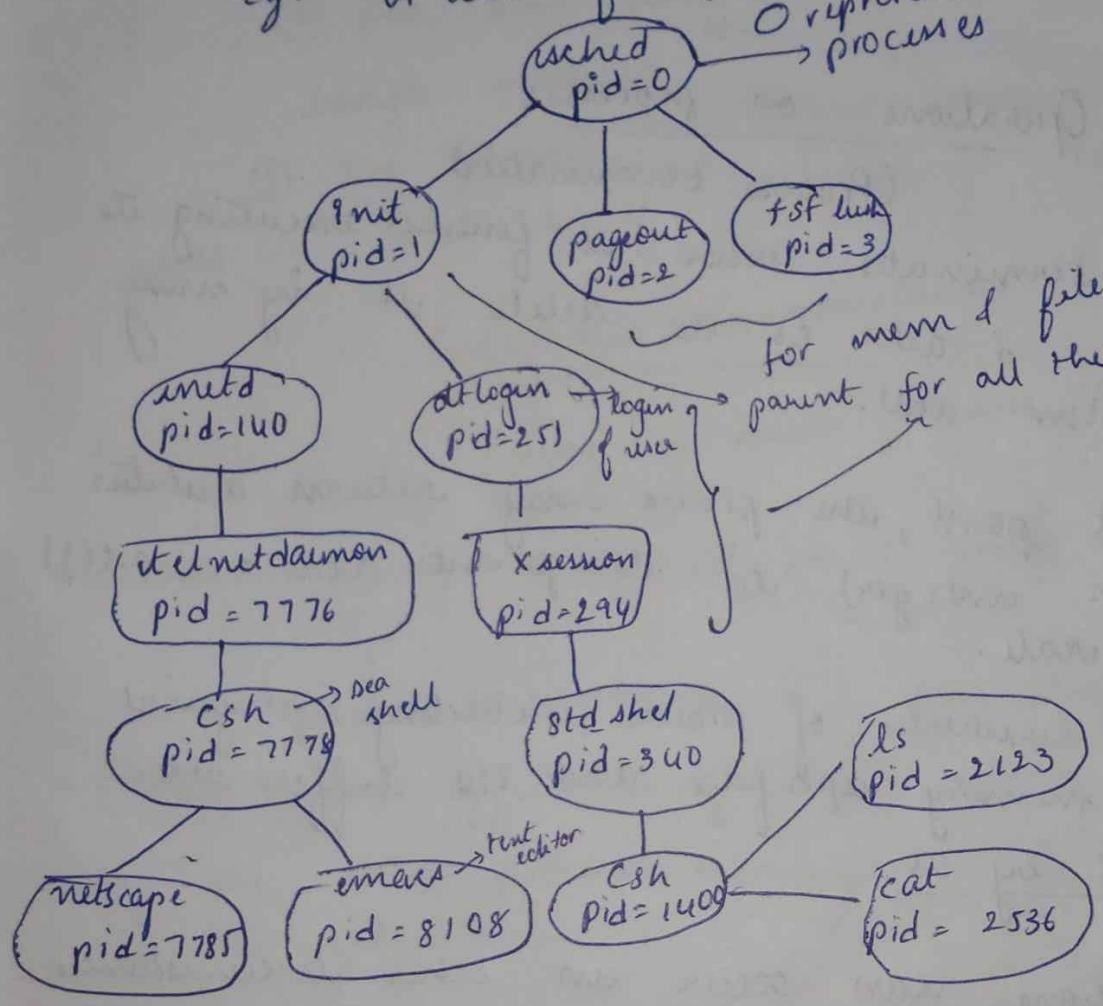
→ A process may create several new processes via a create-process system call during execution



The creating process is called a parent process & the new processes are called the children of that process.

→ Each of these new processes may in turn create other processes forming a tree structure.

Eg:- A tree of processes on a typical solaris sys



when a process creates a new process, 2 possibilities exist in terms of execution.

① The parent continues to execute concurrently with its children

② The parent waits untill some/ all of its children have terminated.

Resource allocation

children can have all resources from parent

children can have subset of resources from parent

There are 2 possibilities in terms of address space of the new process:-

① child process is a duplicate of parent process (it has same prgm as that of parent)

② The child process has a new prgm loaded into it

Operations on processes

(Process termination)

A process terminates when it finishes executing its final stmt & asks OS to delete it by using exit() system call.

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

→ All the resources of process including physical & virtual memory openfiles and I/O buffers are deallocated by OS.

Terminations can occur in other circumstances as well

↓

→ ① A process can cause termination of a 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 could arbitrarily kill each other)

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

① The child has exceeded the usage of some of the resources that it has been allocated.

(To determine this parent must have a mechanism to inspect state of its children)

② The task assigned to the child is no longer required.

③ The parent is exiting & OS doesnot allow child to continue if its parent terminates.

Interprocess Communication

↓

way in which processes communicate with eachother

Processes executing concurrently in OS may be either

independent processes
(They cant affect / be affected by the other processes executing in the system)

→ Eg:- Any process that shares data with processes

cooperating processes
↓
(These process can be affected / can effect other processes executing in the sys)
Eg:- processes that share data with other processes

Interprocess Communication is req by cooperating processes ↓

(because they are going to be affected by each other & need to communicate with each other)

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

① Information sharing:-

(Several 2/more users may be interested in single piece of info (like shared file))

↓

So this processes need to communi-- & cooperate with each other to provide info sharing.

② Computational speed up.

Communication Required { task / subtask | | ... } we can speed up our system

③ Modularity

↓

designing OS by seperating into different modules.

Communication req { modules → → → } put together

④ Convinience:-

↓

If processes can comm-- with each other then it becomes convinient for user

How does IPC takes place?

IPC

Cooperating processes require an IPC mechanism which will allow them to exchange data & info.

2 fundamental models of IPC

① shared memory

② message passing.

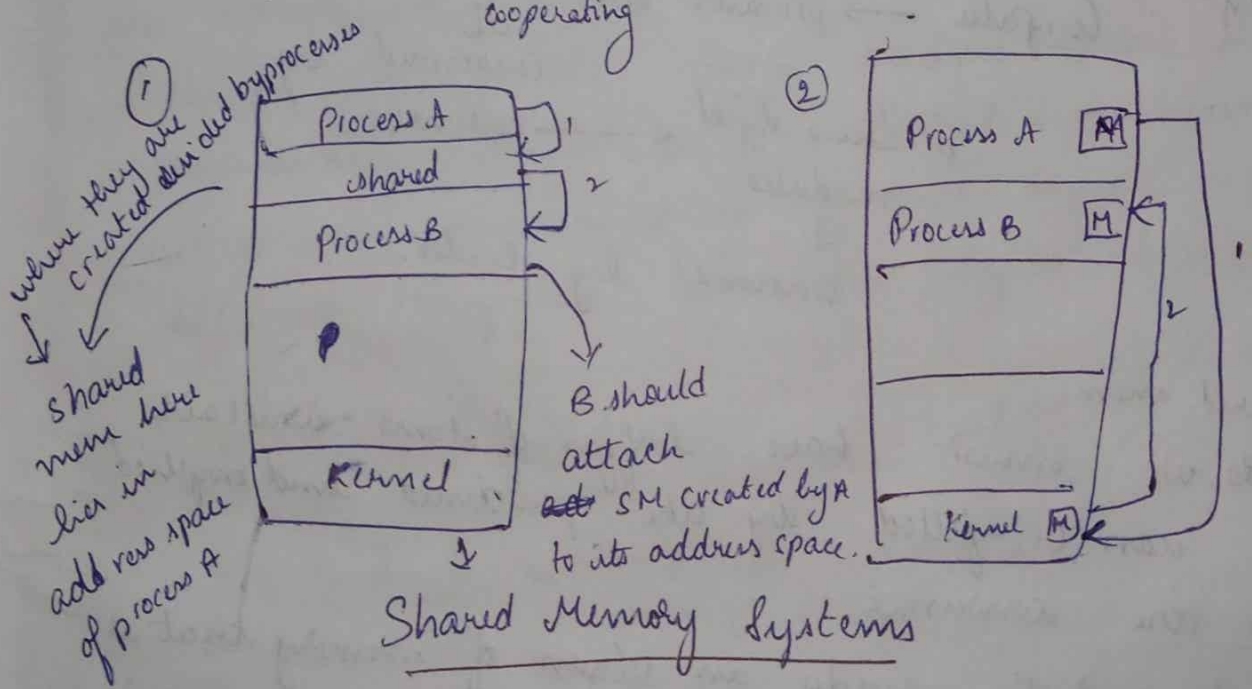
Shared memory

→ In shared memory model, a region of memory that is shared by cooperating processes is established. Processes can then exchange info by reading & writing data to shared region.

Message passing

↓

Communication takes place by means of messages exchange b/w cooperating processes



Shared Memory Systems

Interprocess Commu.. using shared mem requires communicating processes to establish a region of shared memory.

→ Typically a shared mem region resides in the address space of the process creating a shared mem segment.

Other processes that wish to communicate using this shared mem segment must attach it to their address space.

→ Normally OS tries to prevent this shared accessing another processes memory.

→ Shared mem requires that 2 / more processes agree to remove restrictions.

Producer Consumer problem

A producer process produces info that is consumed by a consumer process.

Eg:- Compiler → produces assembly code
↓
consumed by
assembler → produces object modules
↓
consumed by loader.

one of soln → shared mem

So we must have buffer of items available that can be filled by the produced and emptied by the consumer.

This buffer resides in region of memory that is shared by the producer & consumer process.

∴ producer & Consumer must be synchronized so that the consumer doesnt try to consume an item that is not yet been produced.

Buffer (2)

unbounded buffer

~~prod~~

Places no practical limit on size of buffer. The consumer may have to wait for (when buffer is empty) all items are consumed new items, but the producer can always produce new items.

bounded buffer.

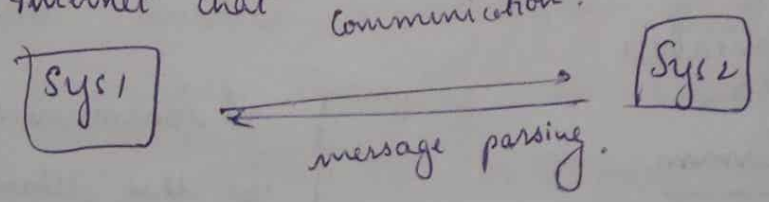
→ Assumes a fixed buffersize. In this, the consumer must wait if buffer is empty & producer must wait if buffer is full.

Message passing

Message passing provides a mechanism to allow processes to communicate & try to synchronize their actions without sharing the same address space & is particularly useful in a distributed environment where the communicating processes may reside on different computers connected by a network.

↓

Eg :- Internet chat



A message passing facility provides atleast 2 operations

① send message ② receive mesg.

Messages sent by a process can be fixed / variable size.

① Fixed size

→ The system level implementation is straight forward

But makes the task of programming more difficult

② Variable size

→ Requires more complex system level implementation

→ But the programming becomes simpler.

Process (P) — Communicate → Process (Q)

For them to communicate a communication link must exist b/w them.

This link can be implemented in variety of ways. There are several methods for logically implementing a link & send/recieve mesg operations like,

* Direct / indirect Communication
* Synchronous / Asynchronous "
* Automatic / explicit buffering

} Several issues related with features like
- naming
- Synchronization
- Buffering.

→ Direct / Indirect Communication

Naming :- Processes that wants to communicate must have a way to refer each other. They can use either direct / indirect Communication

Direct Comm

Each process that wants to communicate must explicitly name the recipient / sender of the communication

send (P, message) – Send mesg to P

recieve (Q, message) – Recieve mesg from Q

→ A communication link in this scheme has the following properties.

- A link is established automatically b/w each pair of process that want to communicate. The processes need to know each others identity to Communicate.

- A link is associated with exactly 2 processes.
- Between each pair of processes, there exists exactly 1 link

↓

This scheme exhibits symmetry in addressing, i.e both sender & receiver process must name the other to communicate

Another variant of direct Communication

Here only the sender names the recipient & recipient is not required to name the sender.

send (P, msg) → send msg to P

recieve (id, msg) – Recieve msg from any process. the variable id is set to name of process with which communication has taken place.

↓

This scheme exhibits assymetry in addressing

Disadvantages in both schemes (sym, assym ..)

is the limitted modularity of the resulting process definitions. Changing the identifier of a process may necessaitate examining all other process definitions.

Indirect Communication

The messages are sent to & recieved from mailbones / ports.

→ A mail box can be viewed abstractly as an object into which messages can be placed by processes

& from which messages can be removed.

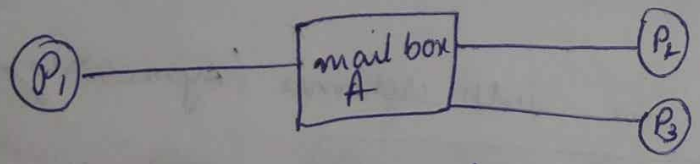
→ Each mail box has a unique identification.

→ 2 processes can communicate only if the processes have a shared mailbox.

→ Properties

→ A communication link in this scheme has the following properties.

- A link is established b/w a pair of processes only if both the members of the pair have a shared mailbox.
- A link may be associated with more than 2 processes.
- Between each pair of communicating processes, there may be a number of different links, with each link corresponding to 1 mailbox.



Process P_1 sends msg to A. while P_2, P_3 execute a recieve() from A.

which process will recieve the msg from P_1?

Answer depends on which method we choose.

- Alow a link to be associated with 2 processes at most
- Allow at most 1 process at a time to execute a recieve operation at a time.
- Allow system to select arbitrarily which process to recieve the msgs.

The system may define an algo for selecting which process will receive mesg. (round robin) The system may identify receiver to the sender.

→ A mailbox may be owned either by a process or by the operating system.

if mailbox. owned by [process] → so no ambiguity who is owner but once process terminates mailbox disappears.

mailbox owned by [OS] → so mailbox has existance of its own.

Synchronous / Asynchronous Communication

Synchronization & Asynchronous

Communication b/w processes takes place through calls to send() & receive() primitives. There are different design options for implementing each primitive.

Message passing may be either blocking & non blocking (also called as synchronous & asynchronous)

→ Blocking Send :- The sending process is blocked untill the mesg is recieved by recieving ~~process~~ process) by mailbox.

Non blocking send :- The sending process sends the mesg & resumes operation.

Blocking receive :- The receiver blocks untill a mesg is available.

Non blocking receive :- ~~Non blocking use~~

The receiver retrieves either a valid mesg / null.

Automatic / Explicit buffering

When communication is direct / indirect, messages exchanged by communicating process reside in a temporary queues (buffers)

Basically, such Queues can be implemented in 3 ways.

① Zero Capacity :- The Queue has a max length of 0, thus the link cant have any messages waiting in it. In this case, the sender must block untill the recipient receives the mesg.

(acts just like a link)

② Bounded capacity buffer :- The Queue has finite length n, thus at most n messages can reside in it. If the queue is not full when a new mesg is sent, the mesg placed in the queue & the sender can continue execution without waiting. The links capacity is finite, however if the link is full, then the sender must block untill space is available in the Queue.

③ Unbounded capacity buffer :- The length of buffer is potentially infinite; thus any no. of messages can wait in it. The sender never gets blocked.

Sockets

↓

(Communication b/w processes mainly in client server based systems)

→ Socket → end point for Communication.

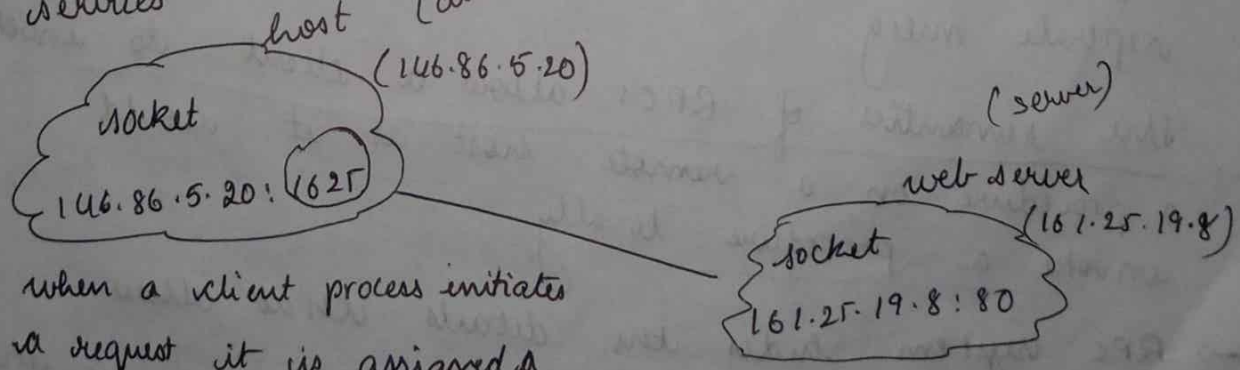
→ A pair of process communicating over a network employ a pair of sockets. (one for each process)

→ A socket is identified by an IP address concatenated with a port no.

→ The server ~~receives~~ waits for incoming client requests by listening to a specified port. Once a request is received, the server accepts a connection from the client socket to complete the connection.

→ Servers implementing specific services (such as telnet, ftp & http) listen to well known ports.
(a telnet server listens to port 23 & ftp server listens to port 21 & and web or http server listens to port 80)

→ All ports below 1024 are considered well known, we can use them to implement standard services.



when a client process initiates a request it is assigned a port by the host Comp. This port is some arbitrary no > 1024

The packets travelling b/w hosts are delivered to the appropriate process based on destination port no.

Remote Procedure Calls (RPC)

RPC is a protocol that one program can use to requ a service from a program located in another ~~program~~ comp on a network without having to understand the network's details.

→ It is similar in many respects to IPC mechanism.

→ However because we are dealing with an environment in which the processes are executing on seperate system, we must use a message based communication scheme to provide remote service.

→ In contrast to the IPC facility, the messages exchanged in RPC are well structured & are thus no longer just packets of data.

→ Each mesg is addressed to an RPC daemon listening to a port on the remote system & each contains an identifier of the function to execute & the parameters to pass to that function.

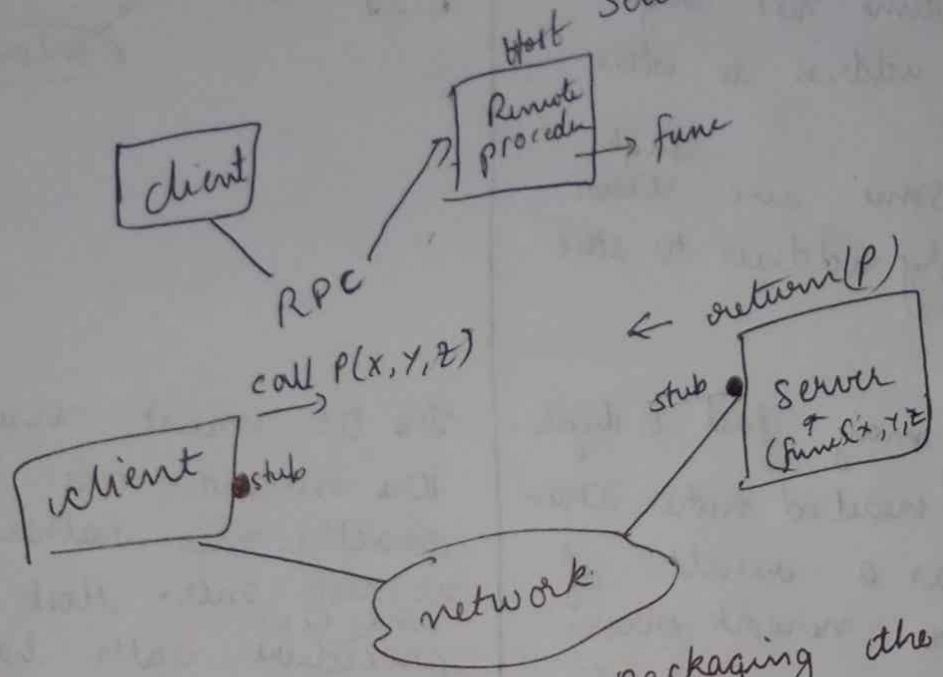
→ The func is then executed as requested & any output is sent back to the requester in seperate mesg

→ The semantics of RPC's allow a client to invoke a procedure on a remote host as it would invoke a procedure locally

→ RPC system hides the details that allow communication to take place by providing a stub on client side.

→ Typically, a seperate stub exists for each seperate remote procedure.

When a client invokes a remote procedure, the RPC system calls the appropriate stub, passing it the parameters provided to the remote procedure. This stub locates the port on the server & marshalls the parameters.



Parameter marshelling involves packaging the parameter into a form that can be transmitted over a network.

→ The stub then transmits a mesg to the server using mesg passing.

→ A similar stub on (other) server side receives this mesg & invokes the procedure on the server.

→ If necessary, return values are passed back to the client using the same technique.

Issues in RPC & How they are resolved

~~Issues~~

Issues

① Diff in data representation on client & server machines

Eg:- For 32 bit Sys

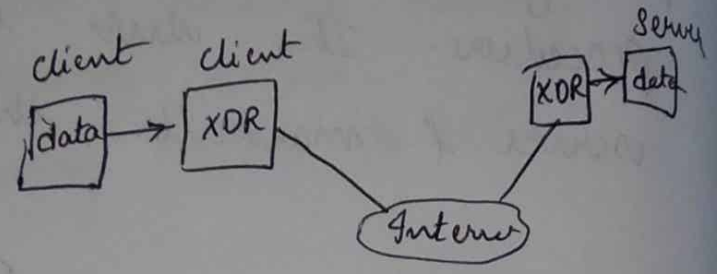
Some systems use high memory address to store MSB.

while Some use ~~lowest~~ high memory address to store LSB.

② RPC's may fail or duplicated & executed more than once as a result of common network errors. (while Local procedures fail at only extreme conditions)

Resolving

RPC systems define machine independent representation here XDR → external data repres...



The OS must ensure that the messages are acted on exactly once, rather than at most once (may be 0). Most local procedure calls have the exact once functionality but it is more difficult to implement.