Operating System

Date Jan, 14 2023
M T W T F S S

OS:

→ Interface b/w hardware and users
(or)

→ OS is a collection of programs (software) that helps users to use the computer hardware.
(OR)

→ This OS gives you a particular beautiful view of a ugly hardware
(OS)

→ Software abstracting hardware
→ Set of utilities to simplify application development /execution
→ Control program
→ Acts like a government

Services of operating system

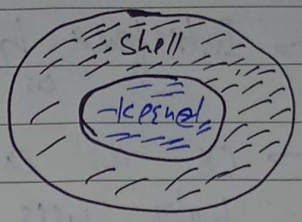
→ Users Interface
→ Program execution (important service in OS)
→ I/O operation
→ File-System manipulation (manipulate → handle or control)
→ communication (Inter-process communication)
→ Error detection
→ Resource allocate (Memory allocate for programs in Ram
→ Accounting
→ Protection and security

Goals of operating system

① Convenience (users Friendly) ② Efficiency (use resource effectively best way)
③ portability (Portable for every hardware) ④ Reliability ⑤ Scalability (updat or enhance features without unistalling) ⑥ Robustness (handle unexpected erro

Date 20
M T W T F S S

Parts of OS



Kernel: core

All the functionalities of OS (OS) functions of all operations are written in kernel

Shell: frontend

kind of interface that use the functionalities that are in kernel

Shell → interface → graphical user interface (GUI) | command line Interpreter (CLI)

System call

A system call is a way for programs to interact with OS

for any operations ~~OS~~ shell call particular function is called system call having special rights

Privileged operations:

that only OS can do

unPrivileged operation:

that user or process can do

Dual mode of operation: (used to implement protection)

2 modes

1) user mode (mode bit = 1)
2) kernel / system / supervisor / Privileged mode (mode bit = 0)

Date Jan, 15 2023
M T W T F S S

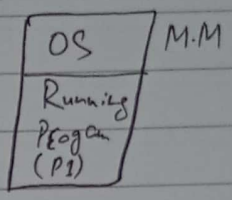
Types of operating systems

① Uni-programming OS

OS allows only one process to reside in main memory (RAM)

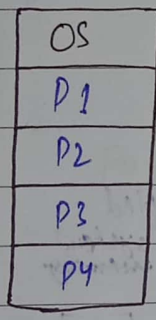
→ single process can't keep CPU and I/O devices busy simultaneously

→ Not a good CPU utilization.



② Multi-Programming OS

OS allows multiple processes to reside in memory



→ Better CPU utilization than uni-prog OS

→ Degree of multiprogramming
No of running programs (Processes) in M.M.

→ As degree of multiprogramming increases, CPU utilization also increases (but upto certain limit).

Types

Preemptive
Running process can be forcefully taken out from CPU
→ this type not present in uni-prog os

Non Preemptive
A process runs on CPU until it want.
→ Either process terminate or goes for I/O operations

Date 20
M T W T F S S

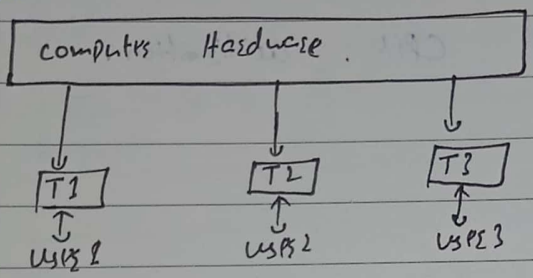
③ Multi-tasking OS/Time-sharing OS → enables users to execute multiple computer tasks at the same time.

Extension of multiprogramming OS in which processes execute in round-robin fashion (fixed time for each process)

④ Multi-Users OS

This OS allows to access single system simultaneously.

Unix, linux → multiuser
Windows → not

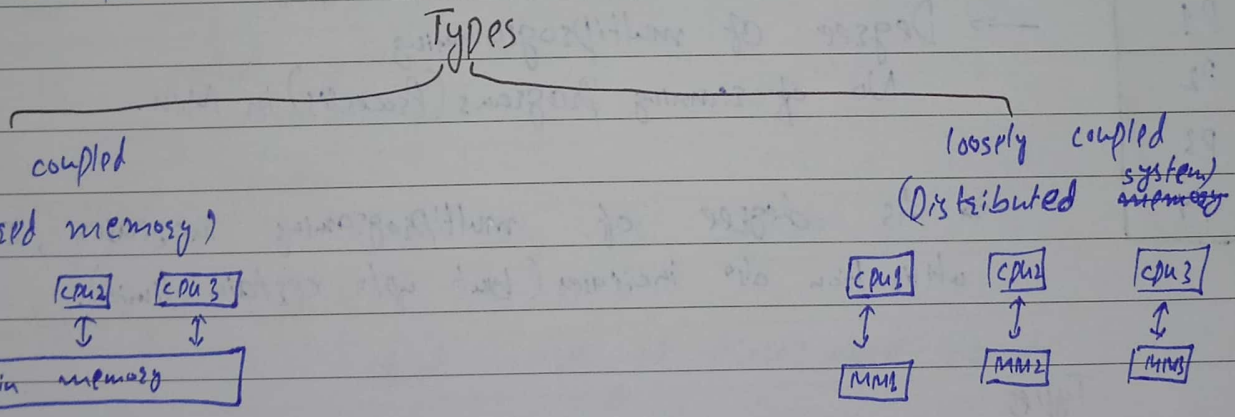


T → keyboard, mouse, display etc

⑤ Multi-processing OS

This OS is used in computer systems with multiple CPUs

Example: Windows NT, 2000, XP and Unix.



⑥ Embedded OS

An OS for embedded computer systems

→ Designed for specific purpose, to increase functionality and reliability for achieving a specific task.

→ user interaction with OS is minimum.

Date 20
M T W T F S S

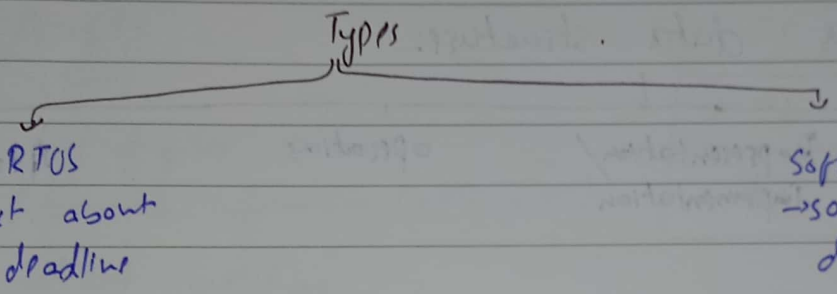
⑦ Real Time OS

Real time OS (RTOS) are used in environments where a large number of events, mostly external to the computer system, must be accepted and processed in a short time or within certain deadlines.

ex

OS used for socket launching

→ Every process have deadline



⑧ Hand-held Device OS

OS used in hand-held devices

Points

→ The process that are residing in main memory and are ready and waiting to execute are kept on a list is called a ready queue.

Process Management

Date Jan, 16 2023
M T W T F S S

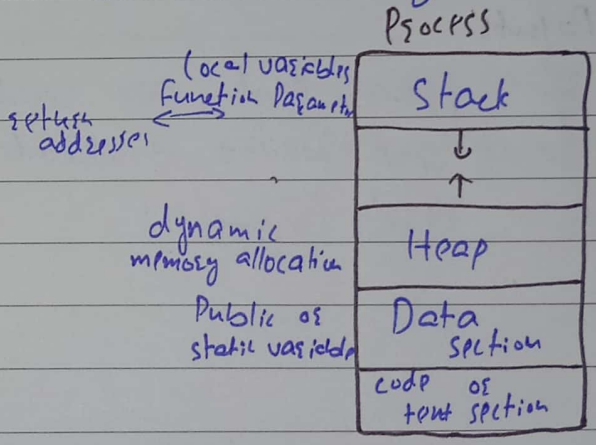
Process:
→ Program under execution
Process = program + runtime activity
→ An instance of program
→ schedulable/Dispatchable unit (CPU)
→ Unit of execution (CPU)
→ Locus of control (OS)

Process as data structure:

Definition — Representation/Implementation — operations — Attributes

Definition
code of only program
↳ a series of actions which are carried out in order to achieve a particular result

Representation of process:
How process stored in memory
every process stored in 4 section in memory



→ Heap and stack are not of fix size

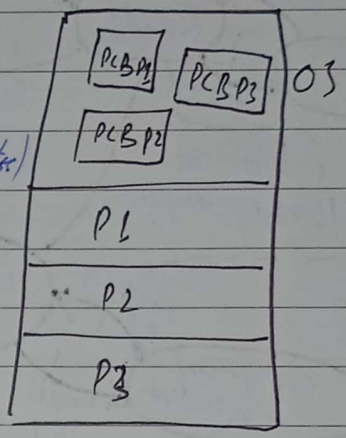
Operations on process:
→ Create (resource allocation)
→ Schedule, Run
→ Wait/Block
→ Suspend, resume
→ Terminate (resource deallocation)

Date 20
M T W T F S S

Attributes of a process

- PID (Process ID)
- PC (Program counter)
- GPR (General Purpose registers)
- list of devices
- Type
- Size
- Memory limits (Base + limit register)
- Priority
- state
- list of files
- Accounting information (CPU cycles)

OS stores attributes of process in data structure and that collection is called process control block (PCB) also known as process discriptor



Context switching:

Stop running process and store its content in its PCB, and load the context of new process in CPU, and run new process
(context load) (context save)

PCB stored in OS protected area

Context:

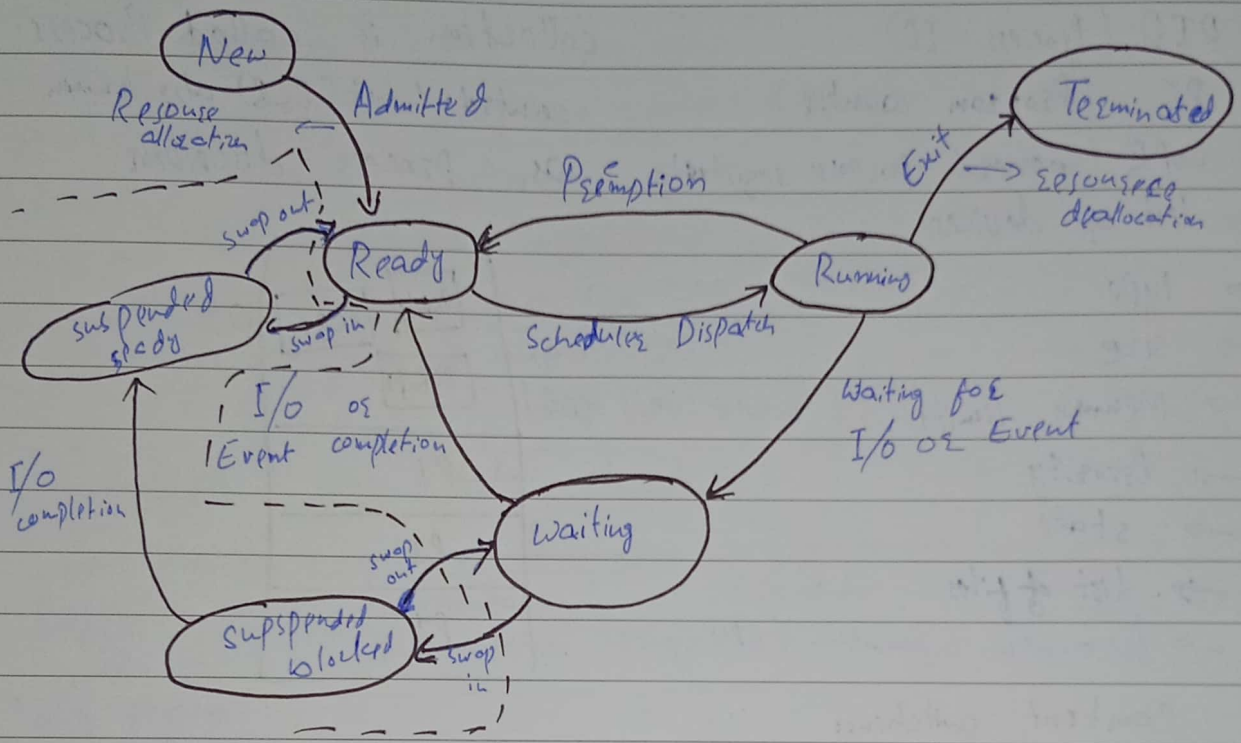
The content of PCB of a process are collectively known as 'context' of that process

Dispatcher: function

context switch is done by dispatcher.

Date 20
M T W T F S S

Process State:



When you open an app then before running process or task that process from new to ready state

Ready:

All process that are going to run in CPU stay in ready state

only 2 transitions running to Termination and running to waiting done process itself

1 transition running to ready (Preemption) done by forcefully (OS).

all others transitions done by OS.

New:

All installed process are in known to be in new state ~~run on CPU~~

Running:

Blocked All process which are waiting for any IO or event

New: If refers to a new process that has been created but has not yet been approved for execution by OS.

Date 20
M T W T F S S

Q: n process admitted, m CPU m<n

State	min process	Max process
Running	0	m
Ready	0	n
Blocked	0	n

Non preemptive process

There is not any preemption transion or, There is not any way to go Running to Ready state.

CPU bound process:

A process which stay more time with CPU or If the process is intensive in terms of CPU operation
ex: Antivirus, IDE

IO bound

If the process in intensive in terms of IO operations
ex:

Printer, Playing movie

IO bound process also use CPU b/c process come in I/O state from running state which is CPU

A process which has just terminated but has to relinquish its resources is called Zombie process

Date 20
M T W T F S S

Process Scheduling

Scheduling is needed b/c we can utilize each and every thing properly
or for better resource utilization

Scheduling Queues

Queues are in OS

→ Job Queue ⇒ kept all processes which are in new state.
→ Ready Queue ⇒ all processes which are in ready state
→ Device Queue ⇒ all process which are waiting for a specific device.

All queues store PCB's of process

Types of schedulers:

1.) Long-Term Scheduler (Job) → (Resource allocation)
2.) Short-Term Scheduler (CPU)
3.) Mid-Term Scheduler (Medium-Term)

} These are three functions in OS

Job scheduler / Long-term scheduler takes process from new state to ready state. The process initiated in two ways by user or by OS (some background processes).

Short-Term Scheduler decides which process has to run on cpu from all those process that are ready.

→

we need two different long-term and short-term schedulers b/c of their frequency of usage.
Short term scheduler used frequently (more).

Date 20
M T W T F S S

Mid-term ~~sheschodules~~:

When you don't have space in main memory to open an app and you want to open it and click on it at time Mid-term shedules takes place it check which process/es are inactive for short time from long time take that one or more process from main memory to hard disk. This process is called swap out. process When your working is complete and you want to back that process which moved to disk and you open it, it will shows you loading and this is called swap in and your process is start from those where you left.

swap out and swap in both is called swapping/~~polling~~ /swapping done by Medium term schedules

swapping is also called rolling if swapping is based on process priority

Swap space:

Area of hard disk which only access by OS and where sp swap out process takes place in hard disk

when any process swap out its state will become suspended

from two states process can swap out from ready or waiting state if process swap out from Ready state it state become suspended ready and if process swap out from waiting state its state become suspended (~~Ready~~) blocked.

When process is swap out its PCB is still in OS

Date _____ 20___
M T W T F S S

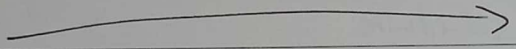
When short term scheduler selects which process has to move running state then disptches takes place to context switching

CPU Scheduling

Function → Make a selection

Goal:

- → Minimize Wait time and Turn-around time
- → Maximize CPU utilization (Throughput)
- → Fairness (all type of process scheduled).



Long-term scheduler controls max degree of multiprogramming if there is few process in main memory long – term scheduler scheduled new process if there is any, if there is maximum process in main memory then long-term scheduler doesn't scheduled any process.

Mid-term scheduler reduces the degree of multiprogramming.

Long term and Mid-term scheduler both controls the degree of multiprogramming

Date 3 Feb 2023
M T W T F S S

Scheduling Times

1. Arrival Time (AT) → The arrival time of a process is when a process is ready to be executed
2. Burst Time (BT) → The total amount of time required by the CPU to execute the whole process.
3. Completion Time (CT) → The time at which process completes its execution
4. Turnaround Time (TAT) → Time from arrival to completion
5. Waiting Time (WT) → WT = TAT - BT
6. Response Time (RT) → Amount of time from arrival time and the time at which process first gets the CPU.
7. Scheduling length (L) → max(CT) - min(AT)
8. Throughput → no of processes executed per unit of time. (No of process / Total time)

Every process has no I/O operation (assumption in all algorithms).

Algorithms

① First come first serve:

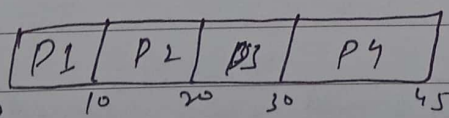
Criteria:
→ Arrival time
if two process come at same time
Tie breaker → Process with smaller id first

Type:
Non-preemptive.

Convoy effect:
If largest process (process) takes place first then short processes has to wait more. (slows down system performance)

In non-preemptive scheduling waiting time is equal to response time.

Gantt chart (always start from 0)



Lec 7

Date Feb, 3 2023
M T W T F S S

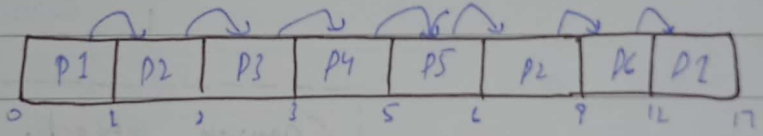
(2) Shortest Job First (SJF)

Criteria: Burst Time (smallest BT process first)
Tie breaker: FCFS

Type: non Preemptive.

(3) Shortest Remaining Time first (SRTF)

Criteria: BT
Tie breakes: FCFS
Type: Pre-emptive



no of context switching = 7

Problems with SJF and SRTF:

1. Stasvation (indefinite waiting for longest process).
2. No Faisness (if now short process coming then ongoing long will only wait).
3. Practical implementation is not possible (b/c OS dont what is burst time of process)

L8

Date Feb 4, 2023
M T W T F S S

④ HRRN (Highest Response Ratio Next)

Objective: Not only favours short Jobs but decreases the WT of longer Jobs.

Criteria: Response Ratio

→ Tie breakes: BT

Type:

Non-Preemptive

$$RR = \frac{w+s}{s}$$

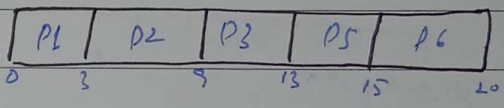
$RR \propto w$

$RR \propto \frac{1}{s}$

w → Wait time

s → service/Burst time

Process	Arrival time	Burst time
P1	0	3
P2	2	6
P3	4	4
P4	6	5
P5	8	2



At time 9:

$RR(P3) = \frac{5+4}{4} = 2.25$ $RR(P4) = \frac{(3+5)}{5} = 1.3$ $RR(P5) = \frac{1+2}{2} = 1.5$

⑤ Priority based Scheduling

Criteria: Priority

Tie-breakes: FCFS

Type → Non-preemptive

→ Preemptive

Priority → of process

→ static → fixed

→ Dynamic → may be increase or decrease by OS

Disadvantage:

→ Starvation: If high priority process keep arriving then low priority process may wait till indefinite time.

Solution of starvation: Aging → After a predefined time period increase priority of all waiting process by 1

→ Only applicable for dynamic priority system.

L9

Date 20
M T W T F S S

Round Robin:

Criteria: AT + Q → Time Quantum

Tie-breaker → Smallest Process ID first

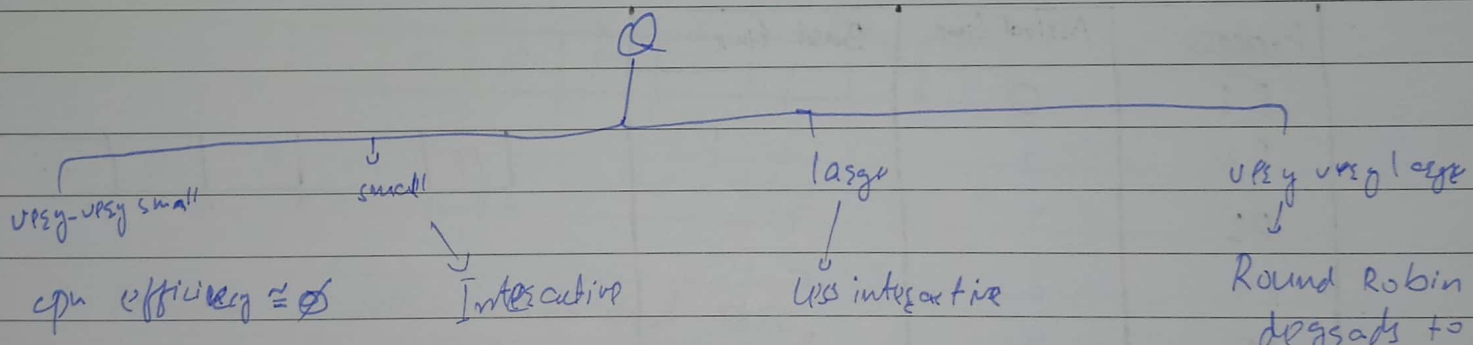
Type: Pre-emptive

Round-robin is not give very less waiting time

Objective: Provides interactiveness
give fairness

No of context switches = Σ (each process burst times / Quantum time)

What should be the Quantum value:



cpu spends more time in context switching and very less time in process execution.

Advantages of RR

→ fairness
→ interactiveness for time-sharing system
→ don't depend on burst time

Date 20
M T W T F S S

System call:

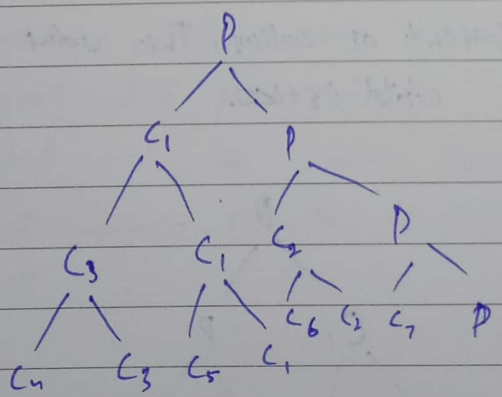
→ to access devices

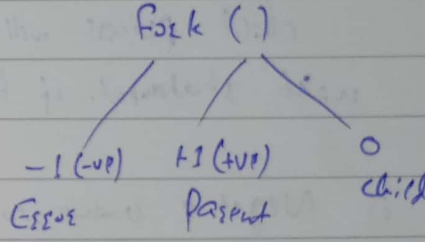
File Related => open, create file etc
Device Related => Reposition, ioctl etc
Information => get Pid, attribute
Process Control => load, execute, fork, abort etc
Communication => pipe(), shmget()
delete/create connections etc

In some OS we don't use directly system calls, we use API's for this OS (local libraries) then it will use system call.

① fork()
↳ to create a child process (which is done of parent but with distinct ID.

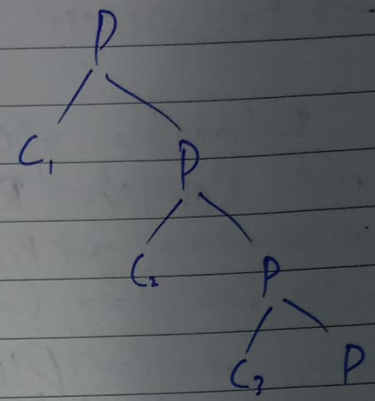
→ Program
fork
fork();
fork();
Printf("hello");
}





Program

```
main{
if (fork() && fork())
    fork( )
Printf("Hello");
  return 0;
}
```



How System Call works

Request of Process from user space ⇒ Interrupt request for kernel ⇒ Control transfer to kernel ⇒ kernel checks if operation can be done

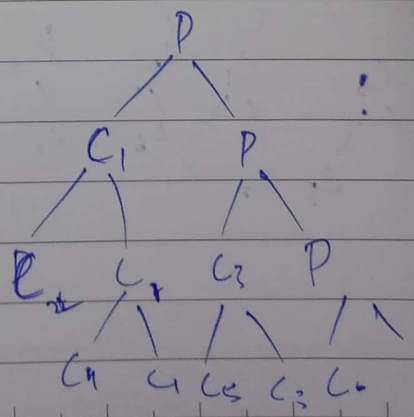
⇓

kernel performs operation ⇐ kernel returns the output to user process

Child process will start from immidately next statement of fork() call.

i) Negative value => creation of child process is unsuccessful
ii) Zero => Returned to the newly created child process.
iii) Positive value => Returned to parent or caller. The value contains Process ID of newly created child process.

```
if (fork() || fork())
    fork();
Printf("1");
return 0;
}
```



Silberschatz, Robert love. linux system programming

Date ____ 20__
M T W T F S S

Booting:

—> Boot loader:

A Program that loads the bigger program. (e.g OS)
OS ka run hona sa phly kr sari chzn khilti gi boot kar process.

Boot -> OS to running

Process in Linux:

The OS creates one task on startup
init: the parent of all tasks —> Pid is always 1
launchd: replacement for init on mac

Deamon -> Process that run in background that haven't any UI

grep -r " " . —> to find something
-r: whole tree
" ": any string
.: dot

Creation Process:

"Create a copy on write"
when child process is created by fork() # both parent child process map to same memory till the process has to read but as when process has to write OS map that memory seperate for both

Date 20
M T W T F S S

Wait()

Block calling process until the child process terminates. If child process has already terminated, the wait call returns immediately.

Waitpid();

Options available to block calling process for particular child process with the first one.

pid_t waits(int * status)

wait call returns value that passed in exit() in child process

Zombie:

Interval b/w child terminating and the parent calling wait → the child is said to be zombie.

ORphan

Process termination

Call to any exec function from a process with more than one thread shall result in all threads being terminated.

New process Inherit

① Process ID ② PPID ③ Process group ID ④ Session membership
⑤ Real group ID ⑥ Real user ID ⑦ Supplementary group IDs
⑧ Current working directory ⑨ Root directory ⑩ Time left until alarm clock signal.

Date 20
M T W T F S S

Exec

Replace the code with another code (binary code) executable file

→ Fork, exec is declare in <unistd.h>

→ Program is same.

If an exec execs if returns → else it returns nothing

① execl int execl (const char *path, const char *arg, ..., NULL);

① Take the path of executable binary file
② Take arguments (can take more than one argument)
③ Takes NULL

If an

② execlp

int execlp (const char * file, const char *arg, ..., NULL)

Parameters

① Take executable binary file
② Take arguments
③ Takes NULL

③ execvp

int execvp (const char *file, char *const argv[]);

Parameters:

② Takes executable file
Takes arguments (we can pass all arguments in NULL-Terminated array argv)

Date 20
M T W T F S S

④ execv

int execv(const char *path, char * const argv[]);

→ In execlp char *file → used to construct a pathname that identifies the new process image file. If the file contains a slash character, the file argument is used as the pathname for the file.

arg[0], .. arg[n]

Pointers to NULL terminated characters strings. These strings constitute the argument list available to the new process image. Must terminate this list with NULL pointer. arg[0] must point to filename that is associated with the process being started can't be NULL.

you can also use execlp to run unix system commands such as ls, cp, mkdir etc

New program Interits.

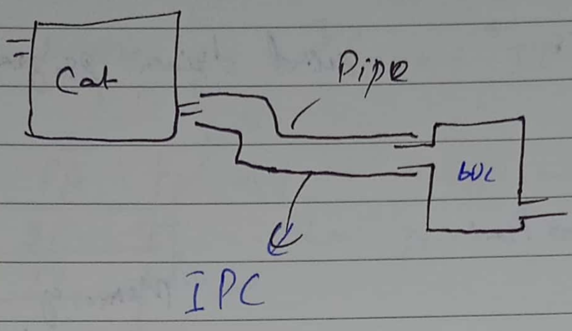
① Process group ID ② Open files ③ Access groups
④ Working directory ⑤ Root directory ⑥ Resource usage and limits. ⑦ Times ⑧ File mode mask ⑨ Signal mask.

Date ______ 20__
M T W T F S S

Inter-Process-Communication (IPC).

| word count |
| wc -c (count char) |
| wc -l (= lines) |
| wc -w (= word) |

Two communicate two process.



OS cannot decide that process is stuck or it is working when process is not responding (Process signal specify nahi karta)

Cascading:
The propogation of effect from one location to other location

If OS ka signal job process accept karna chor deta hai to apka screen pr show hota process not responding

Signals: ps aux → it will show all running process ID
ps aux | grep process-name → it show particular PID

→ Inform process of asynchronous events
→ Process may specify signal "handlers" → Piece of code that ha...
→ Process can poke each other (if they are owned by the same user.
direct kill not any time to inform ← kill -9, PID
→ Sending a signal kill → Send a signal
kill (int pid, int signal number) → 9 → to terminate
→ Detecting a signal signal → receive a signal
signal (signal_number, function).

Date ______ 20__
M T W T F S S

```
int main( ){
                constant number          handler that have to run
    Signal ( SIGINT, handler);
    while (1);
}

                 event handler                event driven program
void handler (int sig ){

    "Signal" , sig ; -> print

}
```

-> context switch is slow

$P_1 \leftrightarrow P_2$

p   c

Memory will not duplicate (on another memory) when fork is called b/c copy on write is used

Cooperating process

Independent:

Process cannot affect or be affected by execution of other process (no any communication with any other process

Cooperating:

Process can affect or be affected by execution of other process

Advantages of process cooperation

-> Information sharing
-> Computation speed up
-> Modularity
-> Convenience

Danger:

-> Data corruption, Deadlocks, increased complexity
-> Requires process to synchronize their processing

most than two process spread only over within

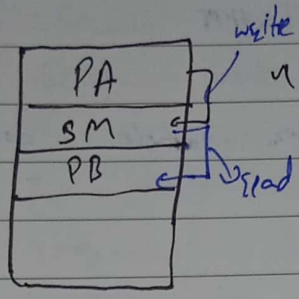
Date 20
M T W T F S S

Two Types of Model

1 Model
2 IPC

→ shared memory
→ message passing

→ shared Memory → fastest IPC communication mechanism



if one process want to give data to another process then process writes that data in SM.

Produces process write on SM and consu process read from SM

(5) Basic steps to use shared memory

(1) Request a memory segment that can be shared by processes to OS

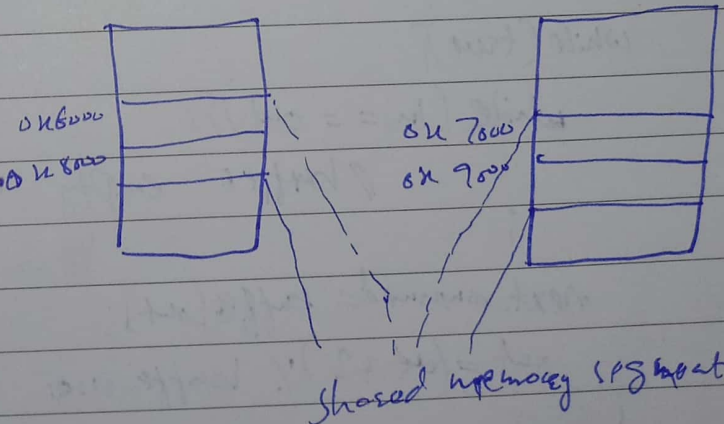
(2) Associate a part of that memory or whole memory with the address of the calling process

OS Maps memory segm in the address spac of several processes t read and write in the memory segment w/o calling OS functions

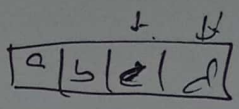
* Shared memory region in two or more process, there is guarantee that the regions will be placed at same base address

forever

(3) Shared memory region resides in address space of process that initiates communication



Two process don't write at same location simultaneously



Date 20
M T W T F S S

Producer – Consumer example

Producer: Produces information that will be consumed by consumer
Consumer: Consumer information that produced by the producer

Two version of producers
① Unbounded buffer → unlimited buffer
② Bounded buffer → limited buffer

Producer can produce one item while consumer is consuming another item.

```
while (true){
    while (((in + 1) % Buffer_size) == out){
        // wait do nothing
    }
    buffer[in] = next_produced
    in = (in+1) % buffer_size;
}
```

Scheme allows at most buffer_size - 1 items in the buffer at the same time

```
item next_consumed

while (true){
    while (in == out){
        // buffer empty wait
    }
    next_consumed = buffer[out];
    out = (out + 1) % buffer_size;
}
```

Date 20
M T W T F S S

Inter-related process communication is performed using Pipes or Named pipes. Unrelated process communication performed using Named pipes or by IPC techniques (shared memory and message queue)

Problem with pipes, FIFO and message queues → information exchange b/w two process goes through kernel it must go
→ Server reads from input files
→ Server writes this data in a message using pipe, fifo or message queue.
→ The clients reads the data from IPC channel, again requesting data to be copied from the kernel's IPC buffer to the clients buffer.
→ finally ~~data~~ 'data' is copied from the ~~the~~ clients buffer

total four copies of data (2 Read, 2 write)

Message-Passing model
→ cooperating process communicate by exchanging messages exchange messages with one another.
→ Implement easily than shared memory.
→ useful in distributed environments (Network based usage)
→ implement using system calls and is more time consuming
ex: chatting

Two operation
1) Send(message) 2) Receive message

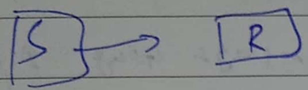
Logical Message Passing model
1) naming
2) Synchronization
3) Buffering

Date ______ 20__
M T W T F S S

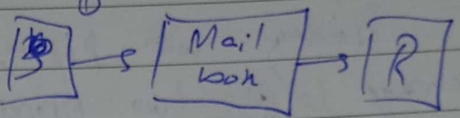
① Naming → Dono ko aik dosre ko identify (Refrence) ko pata hona chaiye

Direct communication

→ sender reciever ko direct message bhejta hai beech mai koi third party nahi hai.



Indirect communication



→ Each mail box has unique id

Indirect Communication

OS working

→ Create Mailbox (M.B)

→ Put a message in M.B

. or take message from M.B

. and send to reciever

→ Mailbox delete karega

Send() → M.B → Receive()

Send(M, Msg)

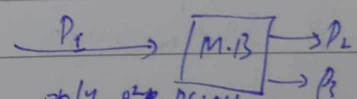
Receive(M, Msg)

Properties of communicate link

→ A link is established b/w a pair of processes only if both shared Mailbox.

→ A link may be associated with more than two processes

Problem



only one process can receive at a time

solutions create separate link

Solution notified which process can see message

solution: one wait one receive after one second receive

→ Each pair of process may have several communication link

Direct communication

Symmetric

Send(P, msg) — Receive

Receive(Q, msg) — sender

Identify

Asymmetric

Send(P, msg) — Receive

Receive(ID, msg)

Address of the process which send the message

Asymmetric network has multiple routes for incoming and outgoing network traffic.

A Link associated with only one pair of communicating process.

. link may be unidirectional but is usually bidirectional

disAdv: Process must know the name or ID of the processes

→ from sender side synchronization ensures that message doesn't send until receive is ready
→ from receiver side requests a message from sender process ensures that sender process is ready to send message receiver doesn't request until sender is ready

receiver is ready to receive

Date 20
M T W T F S S

Synchronization

Communication take place through calls to send( ) and receive functions.

→ Different design to implement each functions include message passing by blocking or unblocking also known as synchronous and asynchronous

① Blocking send ② unblocking send ③ blocking receive
④ unblocking receive

Blocking Send: sender → block until the receiver is ready
Sending process is blocked until the message is received by receiving process or mailbox

Unblocking =: Senders can send the message and continue receive kaygaa.

Blocking Receive: Receive block until until senders sent the message
Receive process blocks until a message is available

Unblocking Receive:
Receiver process receives either message or NULL.

Date 20
M T W T F S S

Buffering:

Message exchanged via any type of communication resides in a temporary queue.

Three types of Queue.

① Zero capacity. (Message passing system without buffering
Maximum length 0. Senders wait until
receipt receives the message.

② Bounded capacity
finite size of queue

③ Unbounde capacity
Infinite size of queue.

⟹ Communication in client/server system

Sockets:

Used for communication in client server system

→ Socket is the endpoint of the communication process

A pair of processes 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 number

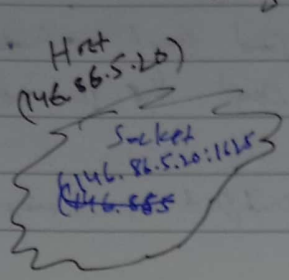
All ports below 1024 are considered well known

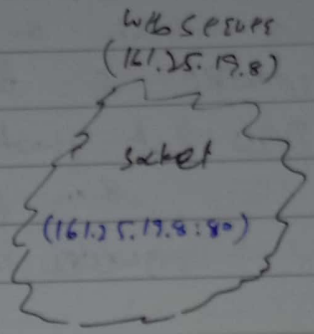
FTP = 21 Total port = 2^{16}

Telnet = 23

http or Web server = 80

Communication using sockets





- When a client process initiates a request for connection, it is assigned a port by host computer

Packets travelling b/w the hosts are delivered to the appropriate process on the destination port number

Sockets can be
- Connection oriented (TCP) socket
- Connectionless (UDP) socket

In Java socket can also be Multicast Socket class data can be sent to multiple recipients

Considered low level modes of communication
→ shares unstructured stream of bytes

Remote Procedure call

Processes that are residing in different system connected over a network wants to connect.

→ RPC is a protocol that one program can use to request a service from a program located in another computer on a network, without having to understand the network's details.

→ We must use a message based communication scheme to provide a remote scheme

→ messages exchanged in RPC communication are well structured and are thus no longer just packets of data

Stubs:→Client side proxy for the actual procedure on the server

Date 20
M T W T F S S

→ Each message is addressed to an RPC daemon (Program which is always listening) listening to a port on the remote system, and each contains an identifier of the function to execute and the parameters to pass that function.
then function is executed output is sent to requester in separate message

How client to invoke a procedure on a remote host.
→ RPC system hides the details that allow communication to take place by providing a stub on the client side.
→ Typically a separate stub exists for each separate remote procedure.
→ when the 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 and marshals the parameters.
→ Parameter marshalling involves packaging the parameters into a form that can be transmitted over a network.
→ stub then transmits the message to the server using message passing
→ A similar stub on the server side receives this message and invokes a procedure on the server.
→ If necessary, return values are passed back to the client the same technique.

Date 20
M T W T F S S

Issues	Solution:
① Differences in data representation on the client and server machines ① little endian ② big endian not occur in IPC	RPC system define a machine-Independent representation of data. One such representation is known as External data representation. (XDR) client side Parameter marshalling involves (changing machine dependent data into XDR
② RPC calls can fail, or duplicated and executed more than once as a result of common network failures.	OS must ensure that messages are acted on exactly once Rather than at most once. Most local procedure calls have the "exactly once" functionality but it is more difficult to implement → fixed port addresses → Rendezvous mechanisms or match makes (store port No and this mes

PIPE → IPC

getpagesize → How many bytes have been written

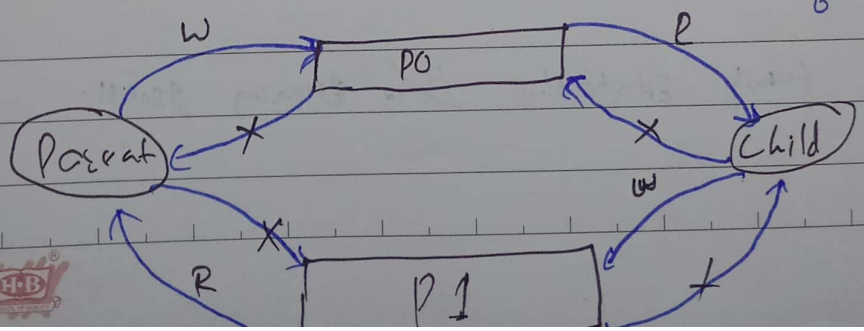
write() P0 ⇒ PIPE ⇒ P1 Read()

one way

=> child and parent process communication
=> FIFO working
=> write 512 bytes
=> Read 1 byte

file descri

#include<unistd.h>
int Pipe(int fds[2])
o fd[0] & fd[1]
Read write
o Return '0' - success
'-1' -> fail
Perror to check



Date 20
M T W T F S S

Ordinary pipes:

→ UNIX/LINUX allow communication b/w related processes

→ Communication is in standard producer–consumer style
→ Unidirectional
→ In windows ordinary pipes

two way
→ use two ordinary pipes one for P, one for C

Named pipes:

→ more powerful than ordinary pipes
→ communication b/w related or unrelated process
→ Several process can share pipe
→ bidirectional
→ exist after communication finished.

Present in both UNIX and Windows More powerful than UNIX

UNIX	Windows
→ FIFO	
→ Bidirectional but half duplex	Bidirectional and full duplex
→ Same machine communication	Intermachine communication is also allowed
→ Byte oriented data transfer (Msg treated as a continuous stream of bytes	Byte oriented or message oriented transmission (msg exchange in discrete unit)

pstree → see family relationship b/w running process.

Threads

Date 20
M T W T F S S

Component of process
or
lightweight process (or) subpart of process
(or) flow of execution through the process code

Threads are independent of other threads

→ Provide a way to improve application performance through parallelism.

kernels are generally multithreaded

Shared Among Threads	Unique for each Thread
Code Section	Thread ID
Data Section	Register Set
OS Resources	Stack
Open files and signals	Program counter
Address space	

Advantages

1. Responsiveness → may allow continued execution if part of process is blocked
2. Faster context switch
3. Resource sharing
4. Economy → cheaper than process creation
5. Communication
6. Utilization of multiprocessor architecture → Scalability

Types of thread

1. User level thread → user process manage it
2. Kernel level thread → OS kernel manage it

} thread will manage by the process

Date 20
M T W T F S S

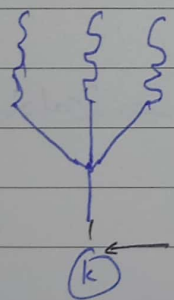
If multithreading implemented is the kernel (OS) — OS aware of it

If users processes implemented the multithreading then OS will not aware of it only user process aware it

User Thread	Kernel Multithread
① Created without kernel intervention	kernel itself is multithreaded
② Context switch is very fast	Context switch is slow
③ If one thread is blocked, OS blocks entire process	Individual thread can be blocked
④ Generic can run on any OS	Specific to OS
⑤ Faster to create and manage	Slower to create and manage
a mangement done by users level threads library	

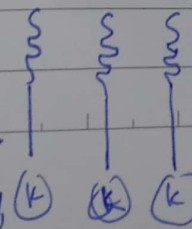
Models

① Many to one



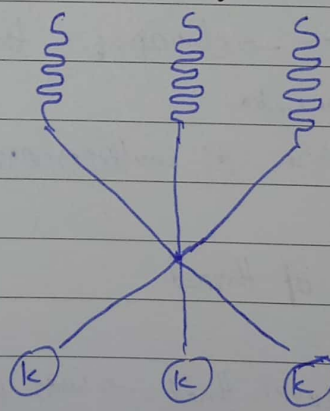
Suffers from lack of concurrency
disadvantage
→ Blocking
→ Not parallel b/c only one may be in kernel at a time

② one to one



① Linux
suffered from overhead of OS among kernel level threads

Many to Many



k. level ≤ users level threads

→ Solaris

gcc inputfile -o outputfile -lpthread

Multithreading

- Execution model that allows a single process to have multiple code segments (threads) that run concurrently within the context of that process

→ In multiple processes each process operates independently of the others
→ One thread can read, write, or change another threads data

Process model has two concepts
(i) Resource grouping
(ii) Execution

Multicore or Multiprocessor

Programmers must accurately use threads
→ Divide independent problems and assign to thread
→ keep a balance in dividing activities, do not over burden one or more threads
→ split data too, so threads can work properly.
→ Data should be split properly, so that dependency among threads remain minimum
→ Testing and debugging of the multithreaded applications is difficult

One thread can read, write or change another thread's data

Types of parallelism:

Data parallelism:

dividing large blocks data sets into smaller chunks that can be process independently by different cores simultaneously perform same operation on each . commonly used on scientific computing.

Date 20
M T W T F S S

Task parallelism:

Processing different tasks parallel on different cores. Each core work on seperate tasks and results are combined to obtained final output. commonly used in webservers, database servers and game engines.

Cpu has cores as well as hardware threads

Creating a new thread

① int pthread_create(pthread_t* thread, NULL, void*, void*)

it will return 0 if it create successfully

→ A → is pointed of thread ID. it should be different for all threads

→ B → points to a pthread_attr_t structure whose contents are used to thread creation time to determine attributes for the new thread. NULL means use default attr.

→ C → is address of function which we are going to use as thread

D → is argument to function. Only one is allowed

② pthread_join(pthread_t thread, void**)

is used to in main program to wait for the end of particular thread

A → thread ID of particular thread

B → used to catch return value from thread

pthread_t → datatype

#include<pthread.h> void indicates that pointer does not have specific type of data

void* a(void *arg) → this can take any type of data and it is pointer to memory location where data is placed and into function we have to cast the data to appropriate data type

e.g. ex: int *pts = (int *) arg;

int pthread_exit(void *retval);

Types

Synchronization

Two parties decided something and doing something that was decided (Two parties should inform each other every thing).

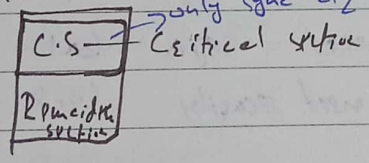
Problems without synchronization

→ Inconsistency
→ Loss of data
→ Deadlock

Critical section

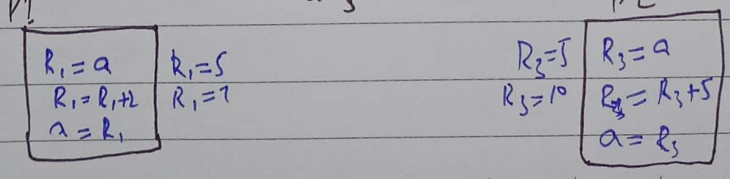
Critical section is a code of segment where the shared variable can be accessed (communication occurs).

There is no synchronization eg. for each instruction execution



Race condition:

A Race condition is an undesirable situation, it occurs when the final result of concurrent processes depends on the sequence in which the process complete their execution



result depends on which finish last

Date 20
M T W T F S S

Requirements of Critical Section Problem Solution

1. Mutual Exclusion: -> If one process is using C.S then other process cannot use that C.S
2. Progress: -> If no any process in C.S and atleast one process wants to enter in C.S then it should be allowed
3. Bounded waiting: -> waiting of process should be bounded

Solution 1: using lock

Pre-empt after each instruction

```
Boolean lock = false;
P0                                  P1
while(true){                        while(true){
    while(lock);                        while(lock);
    lock = true;                        lock = true;
    CS                                  CS
    lock = false;                       lock = false;
    RS;                                 RS;
}                                   }
```

i) Does not satisfy mutual exclusion
if a process preempted after while(lock);
ii) Progress is s

Fail b/c you provide security to shared region but using with shared variable and that variable also need security (lock variable also use synchronization)

Date 20
M T W T F S S

Solution2: Using Turn

```
int turn = 0;

while(true){
  while(turn!=0);
    CS
  turn=1;
  RS;
}
```

→ Mutual Exclusion is satisfied
→ Progress is not satisfied.
→ Bounded waiting also satisfied

```
while(true){
→ while(turn!=1);
  CS
  turn = 0
→ RS;
}
```

2 Process will execute only in strict alternate manner

Solution3: Petessonis Solution

```
Boolean flag[2];
int turn;
  while(true){
    flag[0] = true;
    turn = 1;
  while(flag[1] && turn==1)
    CS
  flag[0] = false
   RS;
} }
```

checking for bounded waiting

P0 Enters in CS
P1 waits outside CS
P0 comes out

flag[0] = f t f t
flag[1] = f t
turn = t ϕ ?

flag[0] = 0
flag[1] = 0

```
while(true){
  flag[1] = true;
  turn = 0;
 while(flag[0] && turn == 0)
   CS
  flag[1] = false
  RS;
}
```

Problem:

- Petesson solution isn't practical b/c can not solve CS problem for more than two processes at same time.

Date 20
M T W T F S S

Bakery Algorithm

→ Processes numbered 0 to N-1

→ num is an array N integers (initially 0)

— Each entry corresponds to a process

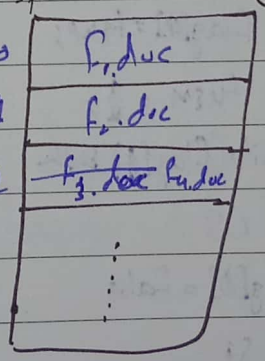
```
lock(i){
    num[i] = Max(num[0], num[i], ..., num[N-1]) + 1
    for(P=0 ; P<N ; ++P){
        while(Num[P] != 0 && num[P] < num[i]);
    }
}

Critical section
unlock(i){
    num[i] = 0;
}
```

Print spooler

value of in

1) Load R_i, m[in]
2) Store SD[R_i], "File Name"
3) inc R_i
4) Store m[in], R_i



Problem loss of data

Date 20
M T W T F S S

$(a, b) < (c, d)$ if $a < c$ and $b < d$

```
do{
    choosing[i] = true;
    numbers[i] = max(numbers[0], ..., numbers[n-1]) + 1;
    choosing[i] = false

    for (j=0; j<n; j++){
        while (choosing[j]);
        while ((numbers[j] != 0) && ((numbers[j], j) < (numbers[i], i)));
    }

        CS                      shared data
numbers[i] = 0;                 choosing[0...n-1] of Boolean
                                numbers[0...n-1] of int
        RS
} while (1);
```

DO will moves waits for any process

Date 20
M T W T F S S

Synchronization Hardware → instruction support by CPU ⇒ can be be provide to use synchronization. → Preveliged instruction

1) Test And Set( )
2) Swap( )

Implementation of semaphore with waiting queue

```
typedef struct{
    int value;
    struct process x list;
}semaphore;
```

value → shows how many process are currently in blocked state

```
wait(semaphore x S)
S -> value--;
    if(S->value<0){
        add proces to s.list;
        block();
    }
}
signal(Semaphore x S){
    S-> value++
    if(S-> value<=0){
        remove a process p from S->list
        wakeup(p);
    }
}
```

block() operation suspends the process that invokes it

Symmetric solution:
Same code for all process & otherwise solution is Asymmetric

→ Deadlock
→ starvation

Remedy for busy waiting: Rather than engaging in busy waiting the process can block itself, block operation: Places a process in waiting queue associated with the semaphore, and the state of the process is switched to the waiting state. Then control transfers to CPU scheduler for select another process.
Wakeup() process that is blocked, should be restarted when some other process executes wakeup() signal(). process become waiting → Ready state

Synchronization tools → Atomic function can pre-emption b/w

1. Semaphore
2. Monitor

(1) Semaphore

It is synchronization tool which only place Integer value which can only be accessed by using following functions only.

- ~~wait~~ wait() / P() / Degrade() / lower() / down()
- Signal() / V() / upgrade / up()

} this increment or decrement the values of integer after accessing it

Both functions are atomic

Atomic function ← Wait(s){ → for entry section in critical section
if any process executing it and in b/w it preempt, then it will again execute from start
while(s<=0);
s--;

Signal(s){ → signal that process come out of the C.S
s++;
for exit of critical section

Busy waiting:
Process is executing, consuming the CPU but is able to execute the further.

Types of semaphore

Binary Semaphore
can store only 2 values 0 or 1
→ use for provide the mutual Exclusion for implement the solution of critical section.

Counting Semaphore
unlimited domain of values (any integers)
→ use when we will have to provide the limited access or we have to control access of particular resources that has multiple instances

Critical section solution

S=1
while(Tsup){
wait(S);
CS;
signal(S);

while(1){
wait(S);
CS;
signal(S);}

Date 20
M T W T F S S

Characteristics of Semaphore:

→ Used to provide mutual Exclusion
→ used to control access to resources
→ Solution using semaphore can lead to have deadlock
→ Solution using semaphore can lead to have starvation
→ Solution = = = be busy waiting solutions
→ Semaphores may lead to a priority inversion → low priority process get CS, before high process.
→ Semaphores are machine independent.

Q:1 Consider a counting semaphore S, initialized with value 1. Consider 10 process P1, P2, ... P10. All process have same code as given below but, one process P10 has signal(S) in place of wait(S). If all process to be executed only once, then maximum number of process which can be in critical section together:

```
while (True){
 wait(S);
  C.S
  signal(S);
}
```

Ans 3
P1 P10 P3

Q:2 A shared variable x, initialized to 0, is operated on by four concurrent processes W, X, Y, Z as follows. Each of the process W and X reads x from memory, incr by 2, store it to memory and then terminates. Each of the process Y and Z reads x from memory, dec by 3, store it to memory and terminates. Each process before reading x invokes wait() operation on a counting semaphore S and invokes signal() function on the semaphore S after storing x to memory. Semaphore S initialized to two. What is the max possible value of x after all process complete execution. Ans = 4

Classical Problems

Date Mar, 3, 2023
M T W T F S S

① Bounded Buffer (Producer - Consumer Problem)

Producer block → if buffer is full
consumer block → if buffer is empty

Variables

→ Mutex: Binary semaphore to take the lock on buffer (Mutual Exclusion).
→ Full: Counting semaphore to denote number of occupied slots in buffer.
→ Empty: " " " " empty slots " ".

Mutex = 1 , Full = 0 , Empty = n

```
Producer(){

wait(Empty)
// Produce item
wait(mutex)
// add item on buffer
Signal(mutex)
Signal(Full)
}
```

```
Consumer(){

wait(Full)
wait(mutex)
// remove any item from buffer
Signal(mutex)
// consume the item
Signal(Empty)
}
```

Date 20
M T W T F S S

(2) Readers-Writers problem

Situation where we have a file shared b/w many people.

→ If one of the people tries to editing the file, no other person should be reading or writing at the same time, otherwise changes will not be visible

→ However, if some person is reading the file, then others may read at the same time.

Solution:

→ if writer is accessing file all other readers and writers will be blocked.

→ If any reader is reading, then other readers can read but writers will be blocked

Variables

mutex: Binary semaphore → provide mutual Exclusion

wrt: = = to restrict readers and writers if writing is going on.

readcount: Number of active readers, Int value.

mutex = 1 , wrt = 1 , readcount = 0

```
Writer( ){                  Reader( ){                  // Perform Reading
                            wait(mutex)                 wait(mutex)
wait(wrt)                   readcount++                 readcount--
  // perform writing        if(readcount==1)            if(readcount == 0)
Signal(wrt)                   wait(wrt)                   signal(wrt)
}                           Signal(mutex)               signal(mutex)
                                                      }
```

Date 20
M T W T F S S

(3) Dining – Philosophes Problem

→ k philosophess spated around a circle.
→ There is one chopstick b/w each philosophes.
→ A philosophes may eat if he can pick up the two chopsticks adjacent to him
→ One chopstick may be picked up by any one of its adjacent followess but not both.

Solution using Binary Semaphore

chopstick[k] = { 1, 1, 1, ... }

This solution can lead to deadlock

```
{
    wait(chopstick[i])
    wait(chopstick[(i+1)%k])
    //eat
    signal(chopstick(i))
    signal(chopstick[(i+1)%k])
}
```

Some ways to avoid deadlock

1.) There should be at most (k-1) philosophess on the table
2.) A philosopher should only be allowed to pick their chopsticks both are available at the same time
3.) One philosophes should pick the left chopstick first and then right chopstick next, while all others will pick the right one first then left one.

for last solution only two wait and two signal condition will

DEADLOCKS

Date 20
M T W T F S S

Operations on Resource

↳ Hardware, software

3 operations on resource

1.) Request: If process required for a resource to OS
2.) Use: When OS allocates resource to process then process use it.
3.) Release: When use is complete the process Release the resource

Deadlock:

If two or more process are waiting for such an event which is never going to occur

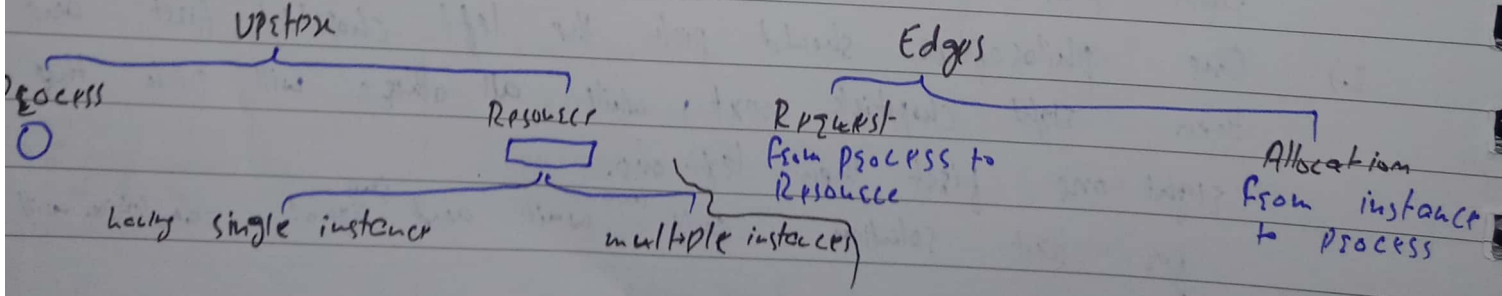
Necessary Conditions for deadlock

Deadlock can occur only when all following conditions are satisfied

1.) Mutual Exclusion: If one process using a resource then other process can't use it.
2.) Hold & wait: each deadlocked process should hold atleast one resource and should wait for atleast one resource.
3.) No-preemption: No any forceful preemption of resources
4.) Circular wait:

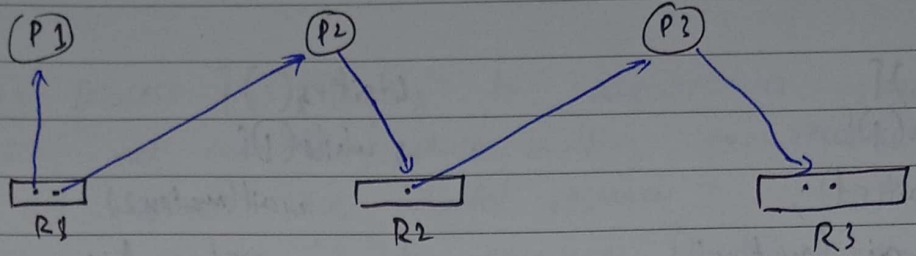
Two instances of ~~res~~ resource if two same resource are present (ex 2 printers)

Resource Allocation Graph (directed).



Date 20
M T W T F S S

ex:



Recovery from Deadlock:

1. Make sure that deadlock never occur • Prevent system from deadlock or avoid deadlock
2. Allow deadlock, detect and recover
3. Pretend that there is no any deadlock.

Pg
Deadlock prevention:

① Make sure that one the necessary conditions from four will never occur ~~by~~ implement system ~~computer~~ like this

Prevent any of four necessary conditions to occur

Preventing Mutual Exclusion
→ increase the resources (can't be implement)
→ purpose a process like this that not require ~~any~~ share variable ~~is required~~ (all process is independent) not possible again they require resources

Prevent Hold and wait
→ A process will either hold or wait but not together
→ If all resources are available then acquire all or just wait for all (starvation can occur)

Date 20
M T W T F S S

Second Solution: writers Precedence

```
Readers(){
  while(1){
    wait(rd);
    wait(mutex1);
      readcount++;
      if(readcount==1)
        wait(wrt)
    signal(mutex1);
    signal(rd)
    //Read
    wait(mutex1)
      readcount--
      if(readcount==0)
       signal(wrt)
    signal(mutex1)
  }
}
```

```
Writers(){
  while(1){
    wait(mutex2)
      wrtcount++;
      if(wrtcount==1)
        wait(rd);
    signal(mutex2)
    wait(wrt)
    //write
    signal(wrt)
    wait(mutex2)
      writecount--;
      if(writecount==0)
        signal(rd)
    signal(mutex2);
  }
}
```

Date 20
M T W T F S S

Continue...

② → If process is trying to acquire a resource which is not available, while holding some resources; then process will release the allocated resources

③ → If a process holds resources but not using them, than these will be poor utilization of resources

Preventing No Pre-emption

P1 request for R_1 & R1 is held by P_2, P2 is also in wait(necessary) for other process/resource OS may Pre-empt R1 from P_2 and will give it to P_1
Problem. not every type of Resources can be pre-em

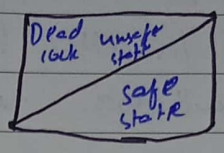
Preventing Circular wait:

① All resources have been given sequence numbers (unique R_1, R_2
② Any process while holding a resource R_i and Request for R_j when $j > i$
③ If a process is holding a Resource R_i and wants another resource R_j when $j < i$ then process will ha to release R_i and will have to acquire R_j first

Date 20
M T W T F S S

Deadlock Avoidance

↳ keep whole system (Processes) in safe state deadlock avoidance OS tries



In deadlock avoidance, the request for any resource will be granted if the resulting state of the system doesn't cause deadlock in the system

Deadlock avoidance will be implementing using the Bankers Algorithm

Bankers Algorithm:

Bankers Algorithm is a Resource allocation and deadlock avoidance algo that tests for safety.

Process	Allocation	Max	Available	Need
P_1	1	3	2,4 for P_2	2
P_2	5	8		3
P_3	3	4		1
P_4	2	7		5

< P_3, P_1, P_2, P_4 > safe sequence. There can be more than one safe sequence

system has 12 resources

① Execute P_3 → all processes can execute hence it is in safe state
= P_1
= P_2
= P_4

Q:2

Process	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	5	3	2	1	2	2
P_2	3	0	2	9	0	2	7	4	3	6	0	0
P_3	2	1	1	2	2	2	10	5	5	0	1	1
P_4	0	0	2	4	3	3	10	5	7	4	3	1

After Execution	Available		
	3	3	2
P_1	5	3	2
P_3	7	4	3
P_0	7	5	3
P_2	10	5	5
P_4	10	5	7

Safe sequence $< P_1, P_3, P_0, P_2, P_4 >$

we have n number of processes and m number of resources

① Allocation: matrix of size n x m
② Max: = = n x m
③ Need: = = n x m
④ Available: array of size m

Date 20
M T W T F S S

Bankers Algorithm:

1.) Let work and finish be vectors of length 'm' and 'n' respectively.

Initialize work = available

finish[i] = false for i = 1...n

2.) Find an i such that both

(a) finish[i] == false

(b) Need[i] <= work if no such i exists goto step 4

3.) work = work + allocation[i]

finish[i] = true

goto step 2

4.) if finish[i] = true for all i

then the system is in a safe state.

What will happen if process P_1 requests one additional instance of resource type A and two instances of Resource type C?

$Request_1 = \langle 1, 0, 2 \rangle$

1. Request is valid (→ step 2) or not (→ error) Request$_i$ ≤ Need$_i$
2. Enough resources available (→ step 3) Request$_i$ ≤ available, else wait
3. Allocate and check safely → if safe → request granted o/w denied.

i) Allocation$_i$ = Allocation$_i$ + Request$_i$

ii) Need$_i$ = Need$_i$ − Request$_i$

iii) Available = Available − Request$_i$

Date 20
M T W T F S S

Questions

1.) Consider a system with 3 processes A, B and C. All 3 processes require 4 resources each to execute. Minimum number of resources the system should have such that deadlock can never occur?

Ans → 10

2.) system with 4 processes A, B, C and D. All 4 processes require 6 resources each to execute. Max number of resource the system should have such that deadlock may occur.

Ans → 20

3.) System with 3 processes that share 4 instances of the same resource type. Each process can request maximum of k instances. Resource instances can be requested and released only one at a time. The largest value of k that will always avoid deadlock is $k = 2$ 6

4.) Consider a system with n processes with single resource R. Each process requires k instance to execute what is the maximum instances of R to cause a dead

$R = n(k-1)$

For no deadlock = $R = n(k-1) + 1 \Rightarrow nk - n + 1$

Date 20
M T W T F S S

5.) Consider two processes P_1 and P_2 accessing the shared variables X and Y protected by two binary semaphores SX and SY respectively both initialized to 1. P and V denote the usual semaphore operators, Take

SX = 1 SY = 1

P_1 :

while true do{
L1: ...
L2: ...
X = X + 1
Y = Y - 1
V(SX);
V(SY);}

P_2 :

while true do{
L3: ...
L4: ...
Y = Y + 1;
X = Y - 1;
V(SY);
V(SX);}

avoid deadlock the correct operations at L1, L2, L3 and L4 are respectively

(A) P(SY), P(SX); P(SX), P(SY) ×
(B) P(SX), P(SY); P(SY), P(SX) ×
(C) P(SX), P(SX); P(SY), P(SY) ×
(D) P(SX), P(SY); P(SX), P(SY) ✓

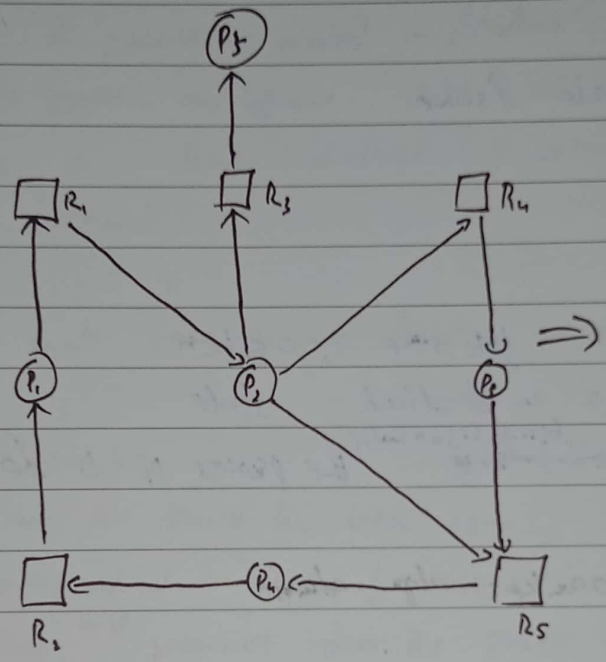
Date 20
M T W T F S S

Deadlock detection

1. When all resources have single instance → wait-for graph
2. When resources have multiple instances → detection algorithm

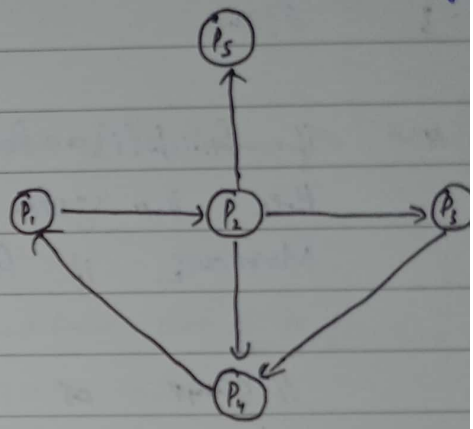
1.) Wait-for Graph

created from Resource allocation graph



vertices → only processes

$P_i \rightarrow P_j$ P_i wait for resource which is held by P_j



wait-for graph

check for cycle if exist then deadlock else no

if resource has more than one instance then is not any guarantee that deadlock occurs.

Date 20
M T W T F S S

2.)

If all processes can execute => No deadlock

Algorithm

1 // same work = available . for i=0 ... n-1 if Request$_i$=0 then finish$_i$ = true else finish[i]= false

2 //same

b) request[i] ≤ work //same

3 //same

4 if finish[i] == false for some i, 0≤i<n then the system is in deadlock state

Moreover, if finish[i] == false the process P_i is deadlocked.

// same as bankers algo almost

When should the deadlock detection be done? Frequently or infrequently? both are time consuming.

1.) Do deadlock detection after every resource allocation => Time consuming

2.) = = = only when there is an clue. (cpu speed decreased)

Date 20
M T W T F S S

Recovery from deadlock

Three basic approaches

1.) Inform the system operator and allow him/her to take manual intervention
2.) Terminate one or more processes involved in the the deadlock
3.) Preempt resources

Process Termination.

1.) Terminate all processes involved in deadlock → bad (very costly in term of time and progress)
2.) Terminate process one by one until the deadlock is broken (Time consuming to check deadlock) (Problem which process to select next for terminate)

Factors that can go into deciding which process to terminate next:

1.) Process priorities
2.) How long the process has been running, and how close it to finishing
3.) How many and what type of resources → process holding
4.) How many more resources does the process need to complete.
5.) How many process will need to terminated.
6.) Whether the process is interactive or batch.

Resource pre-emption

Imp issue to addressed when preempting resources

1.) Selecting a victim → select which process and resource
2.) Rollback → Rollback the progress that done by preempted resource if it take resource start from 0. Or the resource
3.) Starvation → Ask hi process so bas bas resource la sahy hon.

Memory Management

Date 20
M T W T F S S

↳ RAM

→ A Module of OS

Memory fragmentation: is when the sum of available space in memory is large enough to satisfy a memory allocation request but the size of any individual fragment or (continuous fragments) is too small to satisfy that memory allocation request.

Functions of Memory Management

1.) Memory allocation
2.) Memory deallocation
3.) Memory protection → A process can access only its own address space remaining memory is protected from process.

Goals of Memory Management

1.) Maximum utilization of space (Minimum Memory fragmentation).
2.) Ability to run larger programs with limited space
↳ using virtual Memory concept.

Memory Management Techniques

Contiguous	Non-Contiguous
Entire process should be stored in MM on consecutive locations	process can be stored on non-consecutive locations
Fixed Partition, Variable Partition	Paging, Segmentation

Only one can be implement in OS at a time

① Contiguous Memory Management:

i) Fixed Partition

The MM is divided into fixed no of partitions and each partition can different size, and each partition can be used to accomodate max of process.

→ Max degree of Multiprogramming is limited by number of partitions

Memory Management Requirements:
→ Speed is not the only issue
→ Relocation
→ Protection
→ sharing
→ Physical Memory Organization

Date 20
M T W T F S S

Partition Allocation Policy

1.) First fit
2.) Best fit
3.) Worst fit → ~~It remembers the previous partition~~
4.) Next fit → same as first fit but it remember the previous partition and search next from upsegment of it

whichever policy is used there is internal fragmentation (in both partition both there 10% waste).

If extra space is allocated to process more than required space. then wastage of that extra space is known as internal fragmentation

Variable Partition

MM is not divided into partitions initially. When a new process arrives, a new partition is created for same size as process size

If ~~area~~ enough space is available to store a process but not contigous hence the wastage of space is known as external fragmentation

Solution ⇒ compaction:
collect entire allocated process into one side of memory, so that other side of memory can have entire free space.

Date 20
M T W T F S S

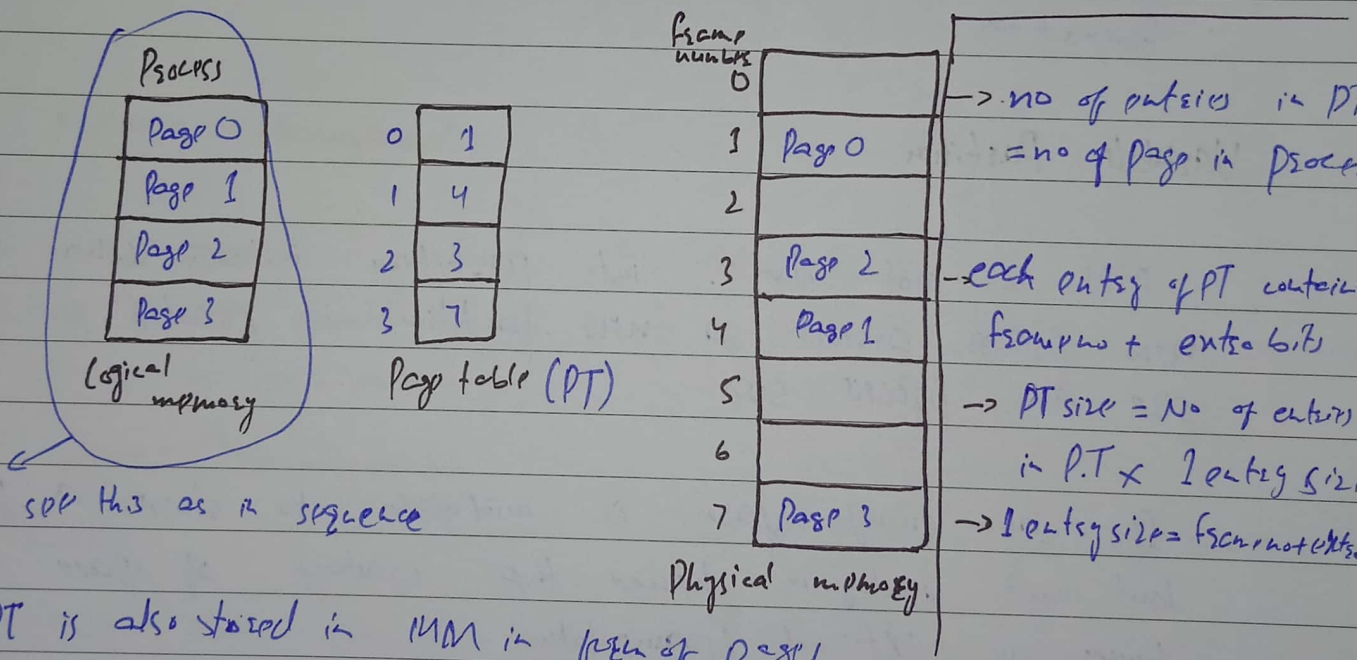
② Non-Contiguous Memory Management

Process is scattered in memory, not allocated at one area

Two Techniques
→ Paging: Scattered in same size of memory Areas
→ Segmentation: Scattered in variable size of memory Areas

Paging

① → Process is divided in equal size of pages.
② → Physical memory is divided in same equal size of frames.
③ → Pages are scattered in frames.



CPU see this as in sequence

→ PT is also stored in MM in form of pages
→ PT is maintained by OS.

④ → Processor will have a view of process and its pages
⑤ → Page Table is used to map a process page to - physical frame.

Each entry of page table store frame No

Date 20
M T W T F S S

a word & a byte requires cpu at a time

Whenever CPU requests a specific content

1.) Find out content present in which page
2.) search in P.T to get the frame no in which page is stored
3.) Goto that frame No and get the content from that frame.

2 times the physical memory is accessed → one for P.T
→ one for content.

Address Translation

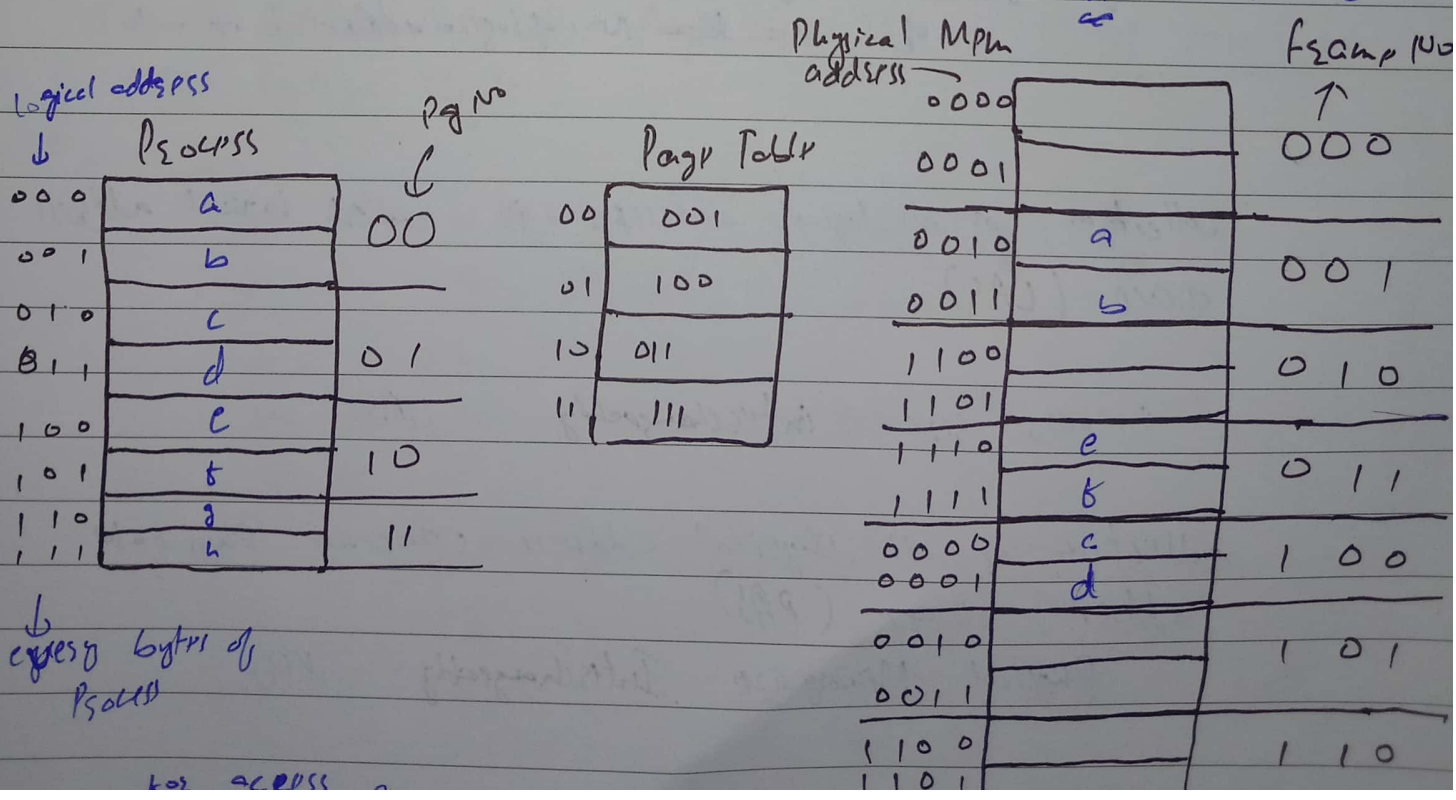
Let

Page size = 2 bytes

Process size = 4 pages = 4×2B = 8B

Physical mem size = 8 frames = 8×2 = 16B = 2^4B

Physical mem → 4-bits required

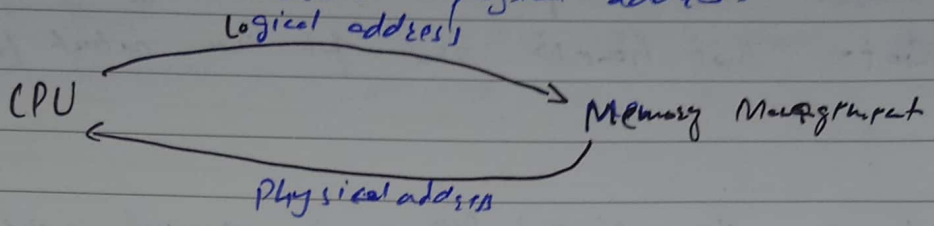


access bytes of Process

for access

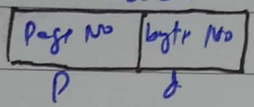
Date 20
M T W T F S S

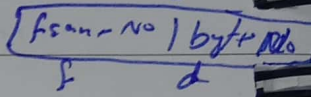
for access a CPU generates one address but it is not physical address it is logical view address Now how to access actual physical address here Memory Management come in Role to access physical address



CPU always generates logical address

Logical address give info → same for physical Memory Address





2^k pages → k No of bits required for page No

$k = \log_2$(No of Pages in Process)

L = bits for byte No = $\log_2$(Page size)

& L = No of logical address bit − k

collection of all logical addresses is called logical address space (LAS)

Process size interchangeably LAS

collection of all physical address space → Physical address space (PAS)

Physical Memory size Interchangeably PAS

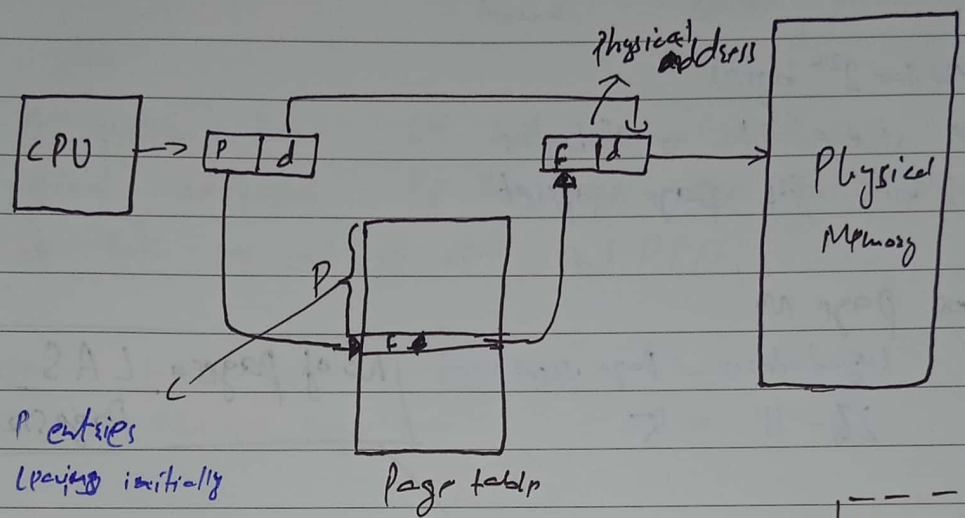
What if PT size is much larger? and PT can not store itself in one page?

Date 20
M T W T F S S

bits for frame No = $\log_2$ (No of frames in Physical Mem)

bits for byte No = $\log_2$ (Page size)

Line No
displacement
Page offset



There is a register PTBR (Page table Base Register) => which holds starting physical address of page table

Find page Table size

Logical Address = 24 bits Page size = 4kb
table Entry = 1Byte

Logical Address = 24 bits 2^{24} logical address space
Page size = 4kb = 2^{12} bytes
12 bits for page offset
P = No of bits in page No = Logical Address − Page offset = 24 − 12 = 12
P = 12
No of Pages = 2^{12} = 4kb

Practise Questions

Date 20
M T W T F S S

Q:1

Logical Address = 26 bits , Physical Address = 32 bits Page size = 2kb
One page table entry size = 4 bytes

① Bits in page offset

Logical Address space 2^{26} bytes
Page size = 2kb = 2^{11} bytes
11 bits for page offset

② Bits for page No

Logical address − Page offset
26 − 11 = 15

No of pages = LAS / Page size

③ No Bits for pages No of frames

$2^{\text{Bits for page no}} = 2^{15} = 32$kb

④ No of frames in physical Memory

No of bits for frame No = Physical address space − Page offset
32 − 11 = 21

No of frames = PAS / Page size

No of frames = $2^{21} = 2^{11}$ kb

⑤ Bits for frame No
21 bits

⑥ Page table size
No of pages × 1 entry size
2^{15} Bytes × 4 bytes = 2^{17} bytes
Page table size = 128 kB

Date 20
M T W T F S S

! Entry content → | Frame No | Extra bits |

1 Entry size = 4B = 32 Bits
= 21 bits + 11 bits
Frame No. Extra

Q:12 A system has 64 bit virtual addresses and 43 bit physical addresses. If the pages are 8kb in size, the number of bits required for VPN and PPN.

VPN → logical (virtual) Page No = 51
PPN → Physical Page No = 30

Q:13 Logical address space of 8 pages, with page size = 1024 bytes
Physical Memory contains 32 frames

1.) Bits in LA

= Page size = 2^{10} bytes
8 pages → bits for No of pages = 3 bits
Page offset = 10
Bits of LA = 10 + 3 = 13

2.) Bits for PA
= 32 frames = bit for No of frames = 5 bits
Bits for PA = 10 + 5 = 15

3.) Page table size
No of pages = 8 × 1 Entry size
= 8 × 5 bits
40 bits 5 byte.

1 Entry size = frames + ...
= 5 + ...
= 5

Date 20
M T W T F S S

Q.5

Size of Physical and logical address spaces & Page table containing 64 entries of 11 bits each (including a valid bit) and a page size of 512 bytes

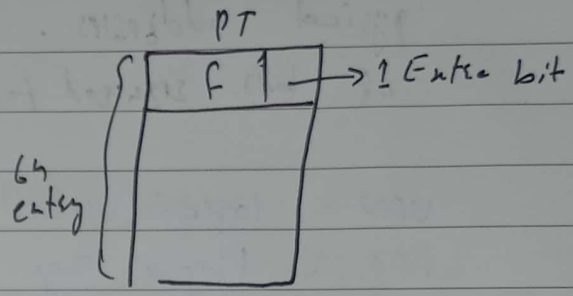
Page size = 2^9 bytes

No of pages = Entries of page table = 2^6

6 bits for page No and 5 bits for page content

LAS = 2^{15} bytes size

PAS = $2^{14+9-1} = 2^{22}$ bytes



Q.6

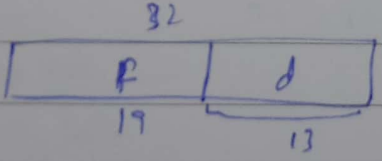
8 Kb pages, 32 bit physical address space. Each page table Entry contains (frame No bits) a valid bit, a dirty bit, three permissions bit and the translation. If the maximum size of the page table of process is 24 MB. the length of virtual address space supported by the system is ____ bits

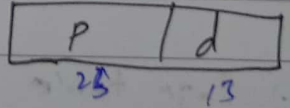
Page size = 8 KB ⇒ 2^{13} Bytes

Physical address = 32 bits

PT Entry = f + 1v + 1d + 3P ⇒ 19 + 1 + 1 + 3 = 24

PT size = 24 MB





PT size = no of pages × 1 entry size

$2^{24} = (2^{20}) \times 8$ = x × 24 bits

$x = 2^{23}$

= 23 + 13 = 36 bits

Page #

Date 20
M T W T F S S

(7) Logical Address space L bytes, Physical address space P bytes
Page size of P bytes and Page table Entry size E bytes
Find all things in 2 !

Time Required in Paging

Effective memory access time = PT access time + content access time from mm

By default, P.T is stored in m.m

$T_{eff} = 2t_{mm}$ t_{mm} = mm access time

Special case: If P.T is very small and is stored in Registers
$T_{eff} = t_{mm}$ Registers access time is negligible

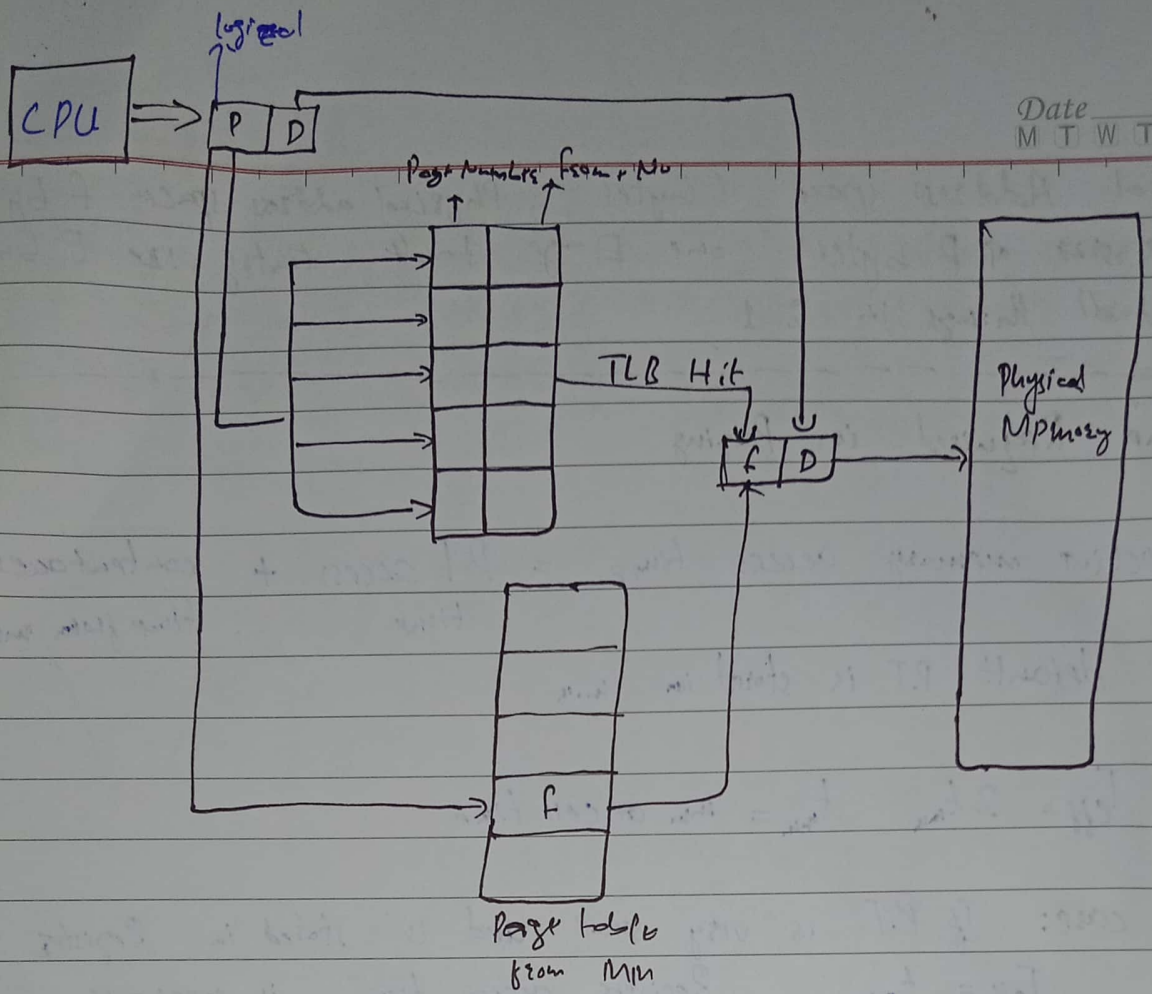
Performance improvement

↳ TLB is used to improve the performance of paging
↳ Translation lookaside buffer (To store Physical address)

TLB:

TLB is a memory hardware that is used to reduce the time taken to access a user memory location

It stores few frequently accessed page table entries so that CPU can get Physical address w/o accessing the m.m (in less time)



Effective Access time with TLB

H -> hit ratio

$$T_{effective} = H * (t_{TLB} + t_{mm}) + (1 - H)(t_{TLB} + 2 * t_{mm})$$

t_{TLB} → Translation, t_{mm} → content, $(1-H)(t_{TLB} + 2 * t_{mm})$ → to ensure it's a miss

Simplified:

$$T_{effective} = t_{TLB} + t_{mm} + (1 - H)\, t_{mm}$$

Question 1

MM access time = 300ns

TLB = # = 20ns

TLB Hit Ratio = 80%

Effective memory access time?

$$T_{eff} = \frac{4}{5}(20 + 300) + \frac{1}{5}(20 + 2 \times 300)$$

$$= \frac{324}{5} \times 4 + \frac{620}{5} = 256 + 124 = 380ns$$

consider a system using TLB for paging with TLB access time of 40ns what hit ratio is used for TLB to reduce the effective memory Access time from 400ns to 280ns

H = 0.8 Ans

$40 = 40 + t_{mm}(1 - h)$

Date 20
M T W T F S S

Semaphore synchronization

Header file → #include<semaphores.h>
Data type for variable → sem_t
Methods for handling semaphores

int sem_init(sem_t *sem, int pshared, unsigned int value);
int sem_wait(sem_t *sem);
int sem_post(sem_t *sem);

pshared → 0 for thread sync. And Non 0 value for process sync
value → Initial value for semaphore variable

Address binding:
process of mapping logical addresses to Physical addresses in memory.
→ compile Time
→ load Time
→ Execution Time
→ logical And physical address are same in compile time and load time address binding scheme
→ " " " " differs in execution time address binding scheme.

Dynamic loading → load the main module at first and others when needed
Dynamic linking → body of function in library call the function is called them to provide body respective library is linked
stub → piece of information

Date 20
M T W T F S S

Important points

→ Process creation is achieved through the fork() system call. The newly created process is called the child process.

→ Cause process to be created. System Initialization, Execution of process creation system call by a running process, A user request to create process

→ Instance: An instance is a single copy of software running on a single physical or virtual server.

→ Antivirus uses too much CPU is that it runs constantly in background

Boot loader, is a small program that has a single function. It loads OS into memory and allows it to begin operation.

File handling in C

FILE * fptr; → object pointer for communication b/w file and program

fptr = fopen ("filename", "mode");
path

Mode →

r → open for reading → if file doesn't exist returns NULL

w → open for write → = = = it will be created if exist overwrite.

fclose (fptr);

fprintf (fptr, "%d", num);

fscanf (fptr, "%d", &num);

char x = fgetc (fptr); → input from a file single character at a time returns the ASCII code of the character read by the function.

(x != EOF) read \n also

fputc (x, fptr);

fgets (char str[], size, fgets(a, 20, fptr);