
G
O
D
O
O
O
O
D


G
O
D
O
O
O
O
D

operating systems :→

Programs are of two types
- → System Programs
- → Application Programs

- ⊙ System programs manages the operation of the computer
- ⊙ Application programs does what user wants. It is a kind of user facing program with which user interacts.

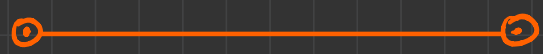
→ Application programs exsist because otherwise programmers will have to do all things. For ex :→ Reading a

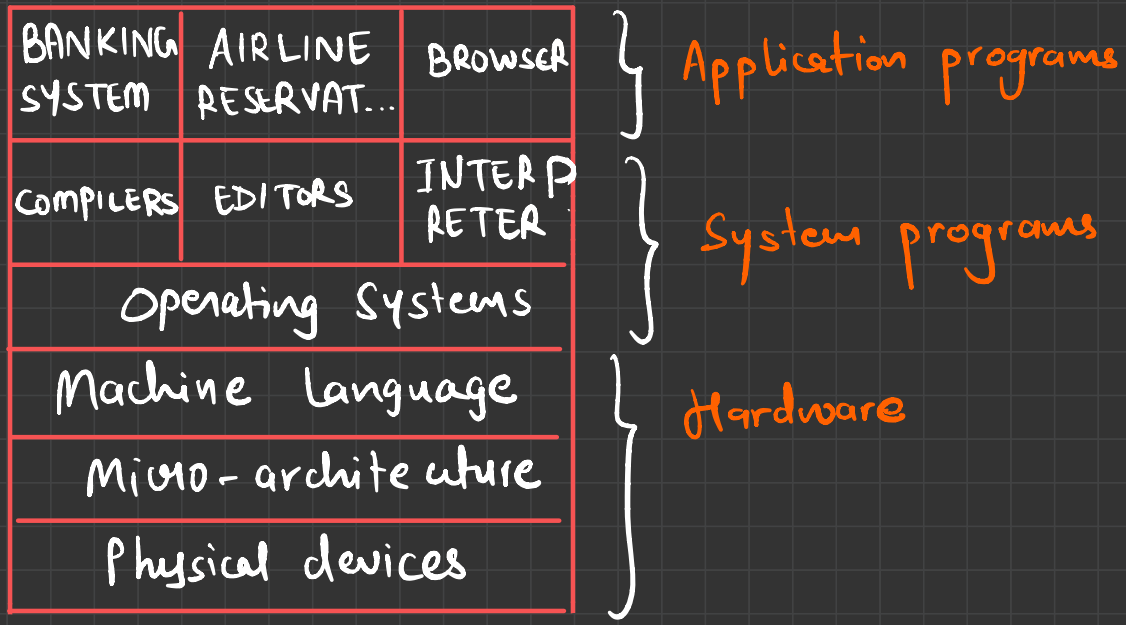
file from the disk, and also considering what can go wrong while reading a disk block.

That's why an interface was needed with which user can interact and do his work

So, the most fundamental system program introduced is OPERATING SYSTEM.

Application programs are built on top of the OS.





Kernel mode OS is that portion of sys program which runs in Kernel mode or supervised mode; which means user programs can't temper the hardware becz user prog. runs in user mode.

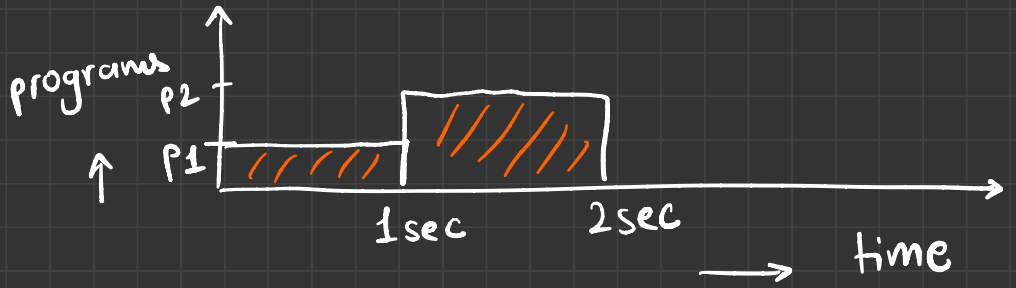
User mode generally runs appl^{n} programs in itself & interact with system programs running in Kernel mode, which means user mode's programs can't change or alter anything in hardware directly bcz that only Kernel's mode programs can do.

Operating System :→ is a system program which runs in Kernel mode and talks to appl^{n} programs and manages the resources. It makes the life of a programmer easy, for ex: instead of programmer adjusting disk heads, blocks to read data, we have file system as an interface using which user can access files & folders

Operating system handles so many complex operations.

And resource managment includes multiplexing

- Time multiplexing
- Space mult.



* scheduling processes on CPU

[Time multiplexing]

* scheduling / providing mem. to programs

[space multiplexing]

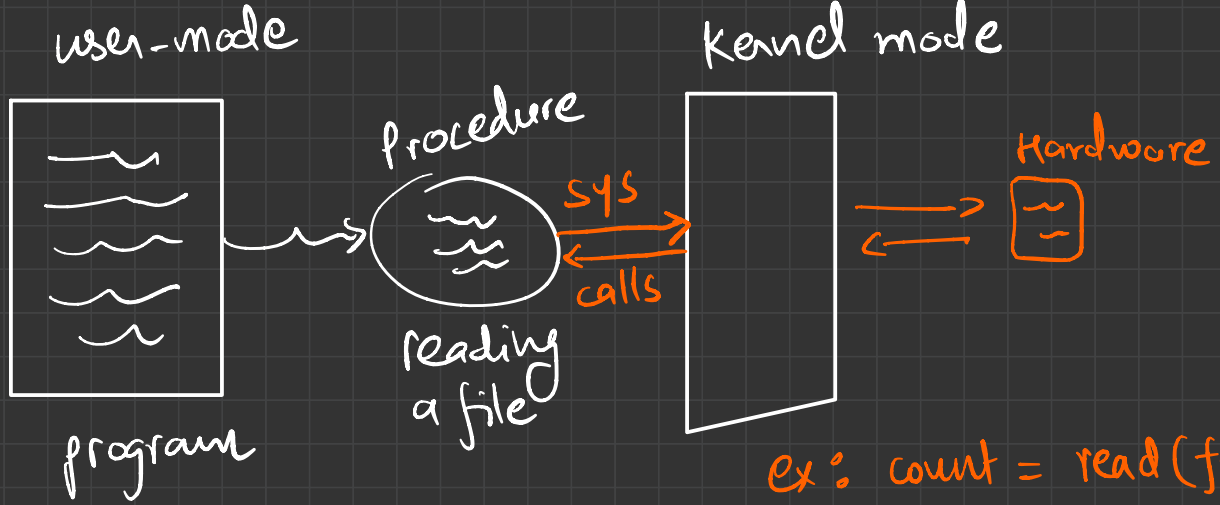
MINIX we will study MINIX OS and will code also if possible so that everything we'll understand in a great detail and can benefit from it lifelong

MINIX is inspired from UNIX OS because UNIX - did change license to not use it for any - University course OR study purpose.

It is written in C-language. MINIX inspired, LINUX originated from MINIX only after a person named "Linus Torvalds" made few changes in MINIX and Linux 1.0 born on March 13, 1994.

① SYSTEM CALLS :-

whenever user/appl" programs wants to run a program that needs to interact with hardware, for ex :- if our program wants to read a file from our Hard disk, it has to go to kernel mode to accen the resources.



ex :- count = read (fd, buffer, nbytes).

SYSTEM CALLS FOR PROCESS MANAGEMENT :→

pid = fork() → creates a child process identical to parent

pid = waitpid (pid , &staloc , opts) → wait for child to terminate

s = execve (name , argv, envp) → replace a process core image

exit (status) → terminate process execution and return status

size = brk(addr) → set the size of the data segment

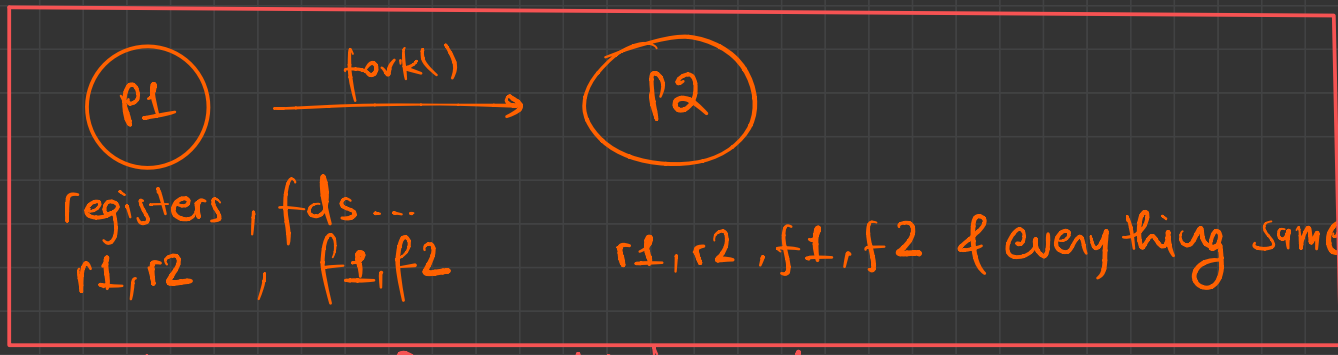
pid = getpid() → return the caller's process id

pid = getgrp() → returns caller's process group id

pid = setsid() → creates a new session & returns its proc. group id

l = ptrace (req, pid, addr, data) → used to get trace & debugging

⊙ fork()

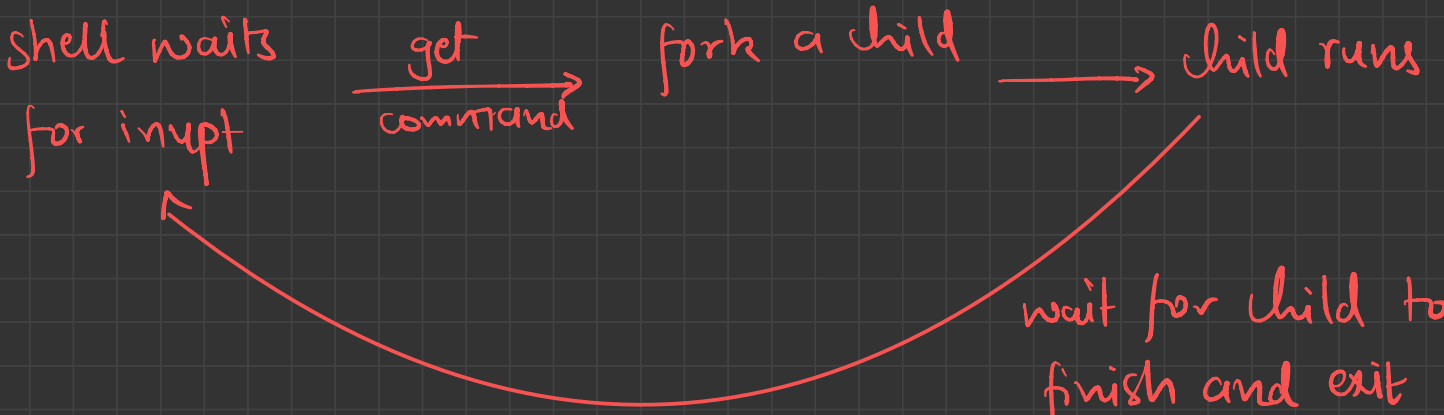


→ at the time of forking everything is same including registers, fds & variable's values.

→ when both runs, they run separately and state change of one doesn't effect another.

pid = fork() returns 0 for child & child's PID for the parent.

running a command in shell :→



{ to wait, parent executes a waitpid system call }

waitpid() waits for any, child to terminate
specific

→ waitpid (pid, &Statloc, opts)

→ after child process's completion the address pointed out by "statloc" will have child's termination - status. { normal, abnormal or exit value }

Now, when child executes the user command → It does so by calling execve system call

High Level working :→

[C-program]

parent waiting ←

child executing the command ←

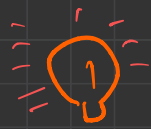
```
#define TRUE 1
while (TRUE) {
    typeprompt();
    readCommand();
    if (fork() != 0) {
        waitpid (-1, &status, 0); }
    else { execve (commd, params, 0); }
}
```

execve(name , argv , envp)

name of the file to be executed

pointer to the arg array

pointer to the env array



Note :-

we will see this later on too.

→ exit (status)

↳ It has value 0-255

→ It is used by the processes to exit/terminate and status returned will go/store to the adds pointed out by &statloc. & will be returned to the parent-

→ lower value like 0, means normal exit/termination and higher values represents some errors

SYSTEM CALLS FOR SIGNALING :→

A process can use sigaction() syscall to announce that it is ready to listen signals

Ctrl+C

process state stored

signal state pushed to a signal handler that process gives the address of while calling sigaction()

- Signals can be blocked in MINIX3
- Blocked signal are held pending until unblocked
- sigpending() call returns a set of bitmap {0, 1, 1, 0} to represent the signals which are pending and blocked
- sigprocmask() call allows a process to define a set of bitmap for kernel to tell the blocked signals
- Insted using a function by a procus, the processes can define a constant SIG_IGN to tell that signals of specific type are blocked/ignored.
- SIG_DFL is to restore the default action of the signal when it occurs.

⊙ the default action of the signal is either - kill the process or ignore the signal which depends on the signal.

* while running a command in the background

"command &" ⟶ shell forks a child process (using fork)

① ↓ runs the command (using exec)

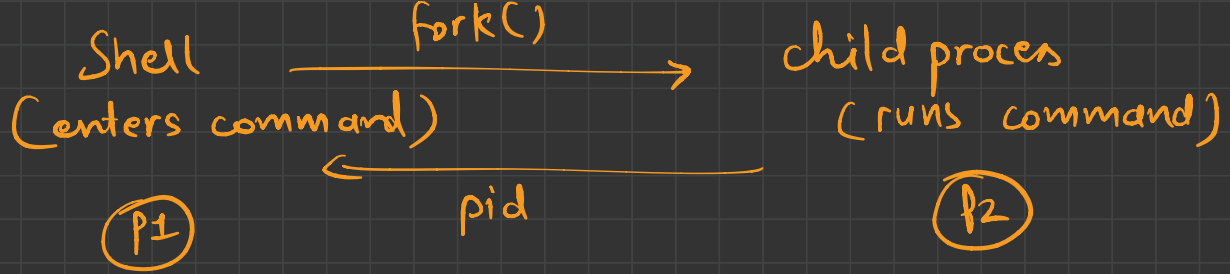
① After the fork() but before running the command - using exec(), the shell does

sigaction(SIGINT, SIG_IGN, NULL) | sigaction(SIGQUIT, SIG_IGN, NULL)

→ It blocks SIGINT and SIGQUIT signals
[because background processes shouldn't be stopped using SIGINT (ctrl+C) and SIGQUIT (ctrl+\)]

* So, for the foreground these signals will be ignored.

Q Then how a background process can be stopped ?



→ Kill system call can be used to kill a process but, to kill P2 from P1 they should be related.

→ unrelated processes can not signal each other.

→ By sending signal 9 (SIGKILL) [kill -9 <pid>]

→ In real life, many a time a process is required to do something after an interval of time. for this SIGALARM is used

SYSTEM CALLS FOR FILE MANAGEMENT :→

```
fd = creat("abc", 0751)
```

↳ creates a file with "abc" as name and with -

rights represented by 0751.

- 0 → 0 in C represents that constant is in octal
- 751 → represents rwx bits

$7_8 = (111)_2$
$5_8 = (101)_2$
$1_8 = (001)_2$

$7_8 = 7 \times 8^0 = 7 = 111$

$74_{10} = 7 \times 10^1 + 4 \times 10^0$

$112_8 = 1 \times 8^2 + 1 \times 8^1 + 2 \times 8^0$

- rwx → read, write, executable permission
- owner (7) → rwx permission
- his group (5) → rx permission
- others (1) → x permission

↓

- fd = creat(---) not only creates file but also opens it for writing through the same fd (file descrip - tor)
- apart from this, to write a file it should be open, write OR read, then close.

open() → read() / write() → close()

- whenever reading OR writing a file a pointer needs to be maintained. lseek() call can be used to move the pointer anywhere
- lskeek() → has three params where first one is the file descriptor (fd), second one tells the position

and third tells the relativity
↳ relative to start, end or current

⊙ MINIX3 stores the info like file mode [regular, special or directory, size and time of last modification. and other information.

* programs can use stat() or fstat() syscalls to get this info.

* OPTIONAL :→ In UNIX like systems like LINUX there are special files too which exists in /dev

These files can be created using a special syscall called mknod().

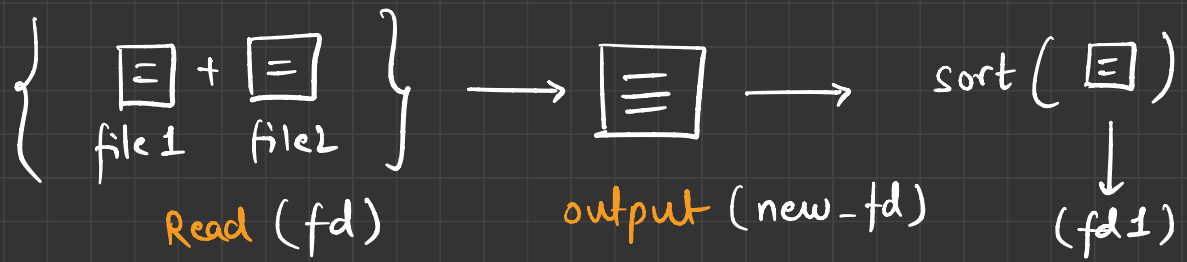
* mknod ("/dev/abc", 020744, 0x0402);

major device (sata disk)

minor device (disk 3)

Interprocess - comm" :-

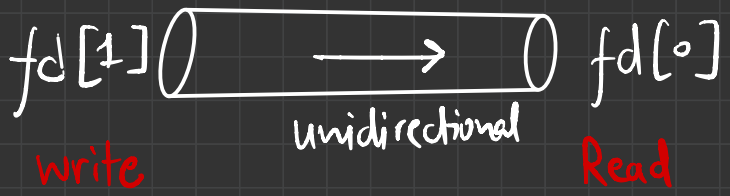
```
cat file1 file2 | sort
```

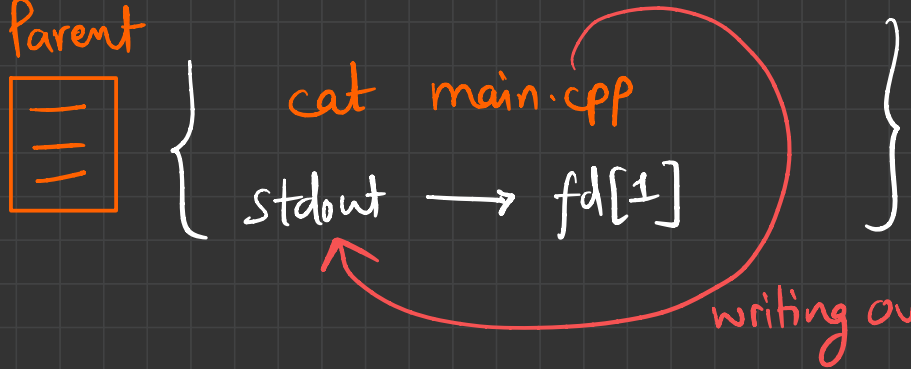


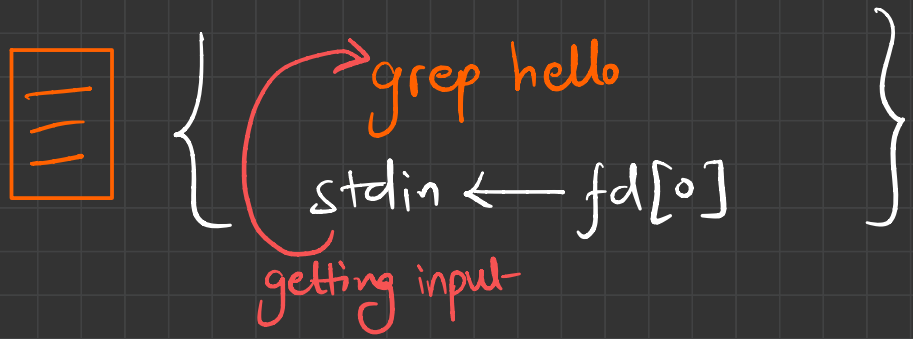
How pipes work :- [dup() sys call]

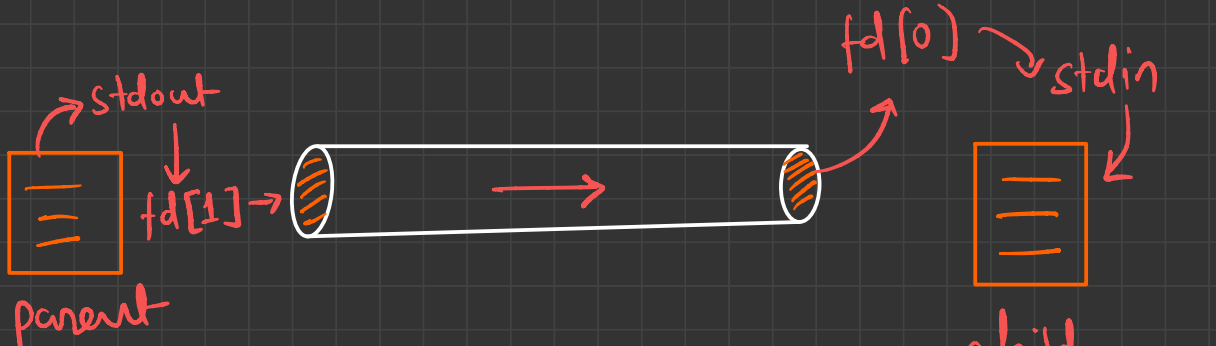
→ " cat main.cpp | grep hello "

```
int fd[2];
pipe(fd);
```









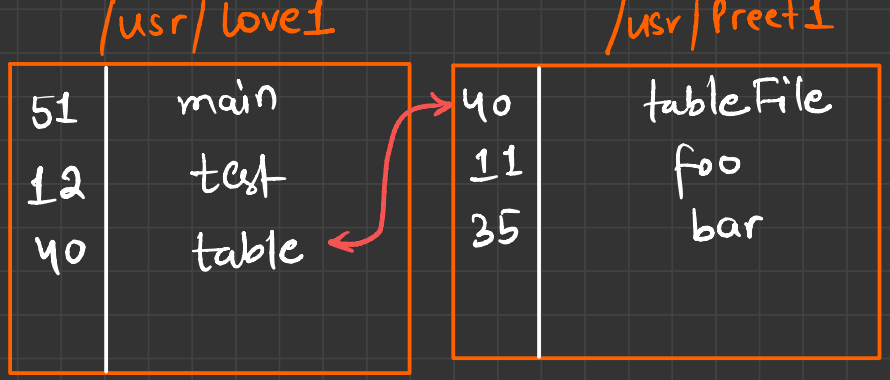
{ cat main.cpp }

{ grep hello }

{ IPC USING PIPE }

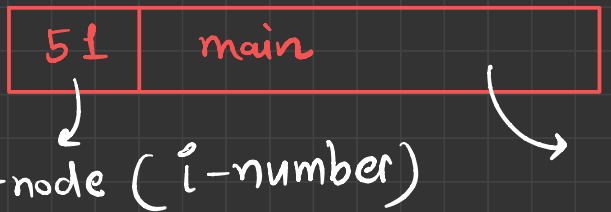
* [SYSTEM CALLS FOR DIRECTORY MANAGEMENT] *

- mkdir() → make an empty directory
- rmdir() → remove an empty directory
- link() → It creates link btw files or we can say It allows same file to appear in two different directories. (shared file)



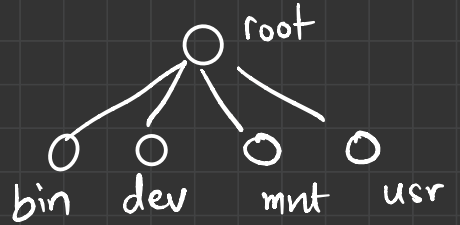
→ link("/usr/love1", "/usr/love2")

→

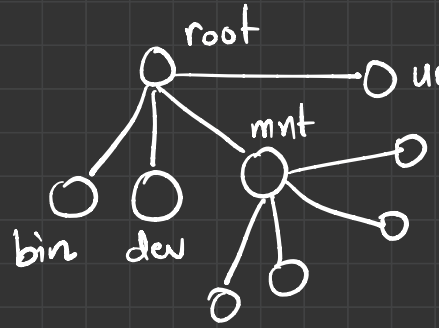


⊙ mount() → mount system call allows two file system to be merged into one.

→ mount ("/dev/cdrom0", "/mnt", 0);



→ Before mount() syscall



→ After mounting CDROM

→ chdir() → changes the working directory

→ chroot() → changes the root, all absolute paths starting with "/" will be replaced with new root directory.

* use case :→ if we want to show user1 only few files then superuser can change the root and remote users can only see the portion of file system

under this new file root.

SYSTEM CALLS FOR PROTECTION :→

→ chmod ("file", 0644)

- 100 (others)
- 100 (group members)
- 110 (owner)

→ SETUID & SETGID bits can also be set using fourth bit as :→

chmod ("file", 02755) — SETGID

chmod ("file", 04755) — SETUID

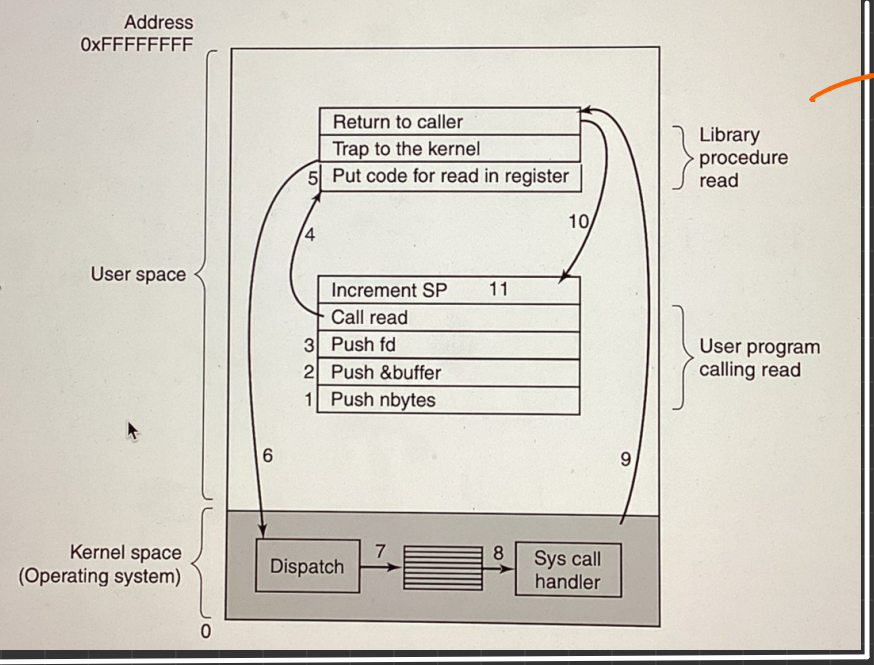
- if user runs any program with "UID" bit on then user's effective UID is changed to that of file owner. Similarly for GID.
- to get user's real UID / GID system calls present are → getuid, getgid.
- to get effective uid / gid → geteuid, getegid.
- chown() → change the owner of the file.
- umask (022); mask the bits
  like creat ("file", 0777) → creat("file", 0755)

Operating Systems :→

* **Monolythic-OS** :→ when everything runs in a single Address space. While it can have a little structure but still not very decoupled.

Sys' calls are called by using a special call i.e. Kernel call.

This call switches user mode → Kernel mode.



→ Required 11 steps to read

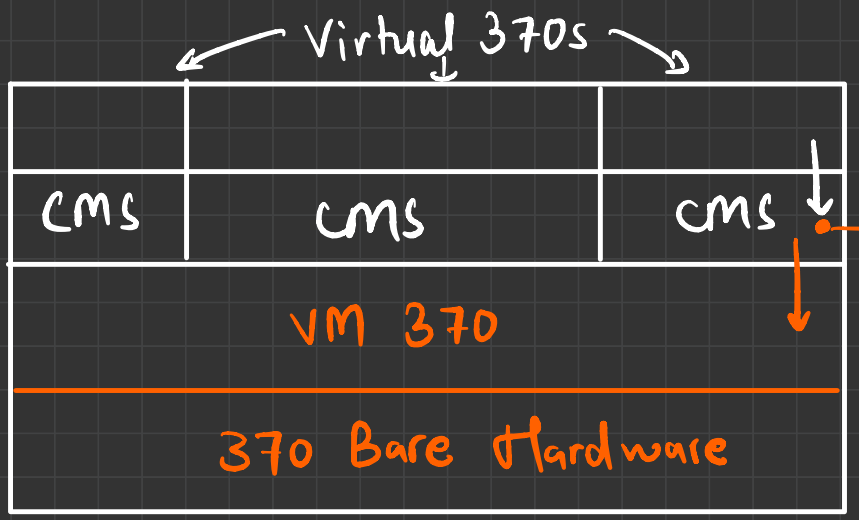
- read (fd, buffer, nbytes)

* LAYERED - OS :→

Layer	Function
5	The operator
4	User programs
3	Input/output management
2	Operator-process communication
1	Memory and drum management
0	Processor allocation and multiprogramming

- abstraction
- Modular & clean
- well defined structure & clear communication

* Virtual machines



CMS → Conversational Monitor system. (Single user interactive OS)

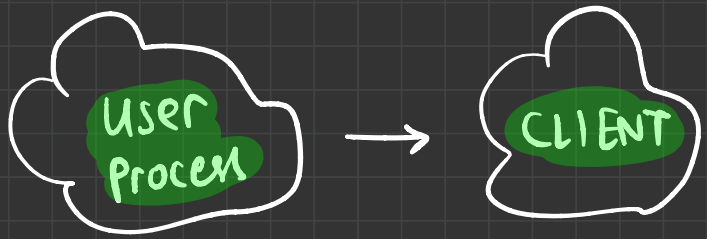
- different machines running layerwise
- This architecture helped running OLD MS DOS programs on Pentium.

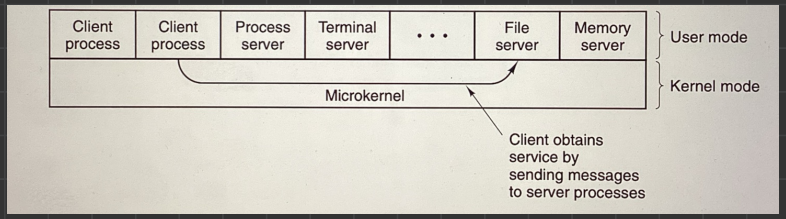
* EXOKERNELS :-> It is similar to VM architecture but at the bottom instead of kernel, exokernel runs. which allocates the resources to different VM but also isolating.

For ex. instead of providing whole disk to the VM, it might give 0 → 1023 disk-blocks to one VM and 1024 to 2047 to another

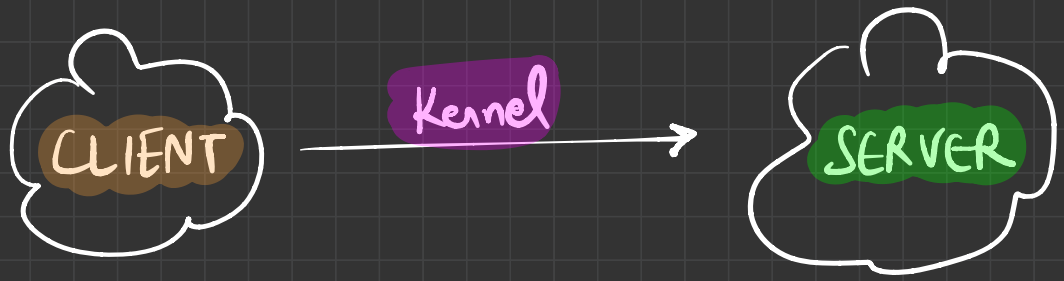
* CLIENT SERVER OS :->

- This approach was to focus on moving most of the functionality out of Kernel and make it simple.

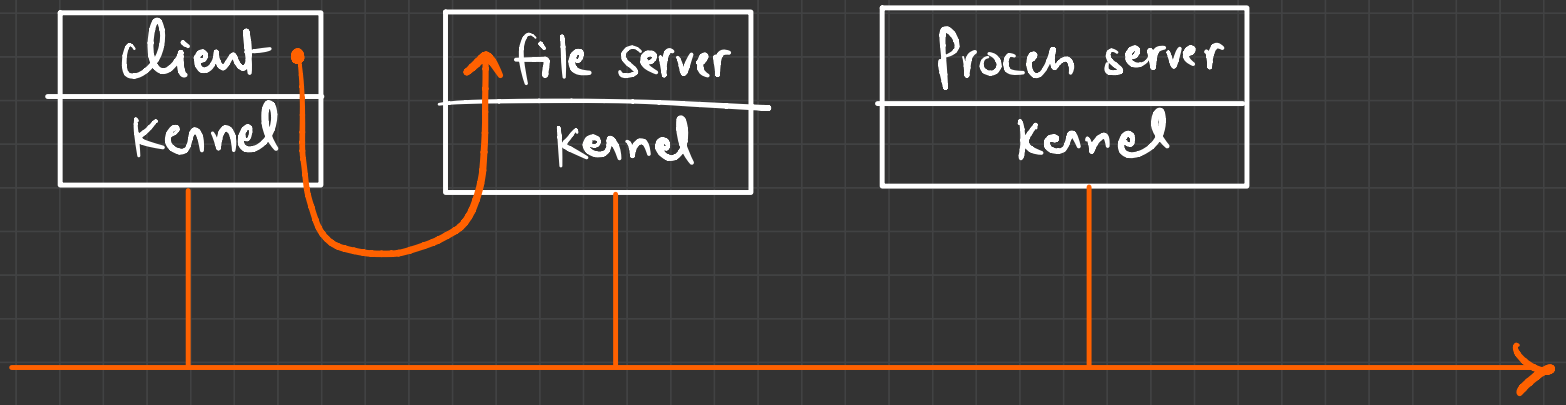




→ Kernel here now handles communication



CLIENT-SERVER IN DISTRIBUTED SYSTEMS !->



* clients are not aware if requests are being served from local or any remote program.

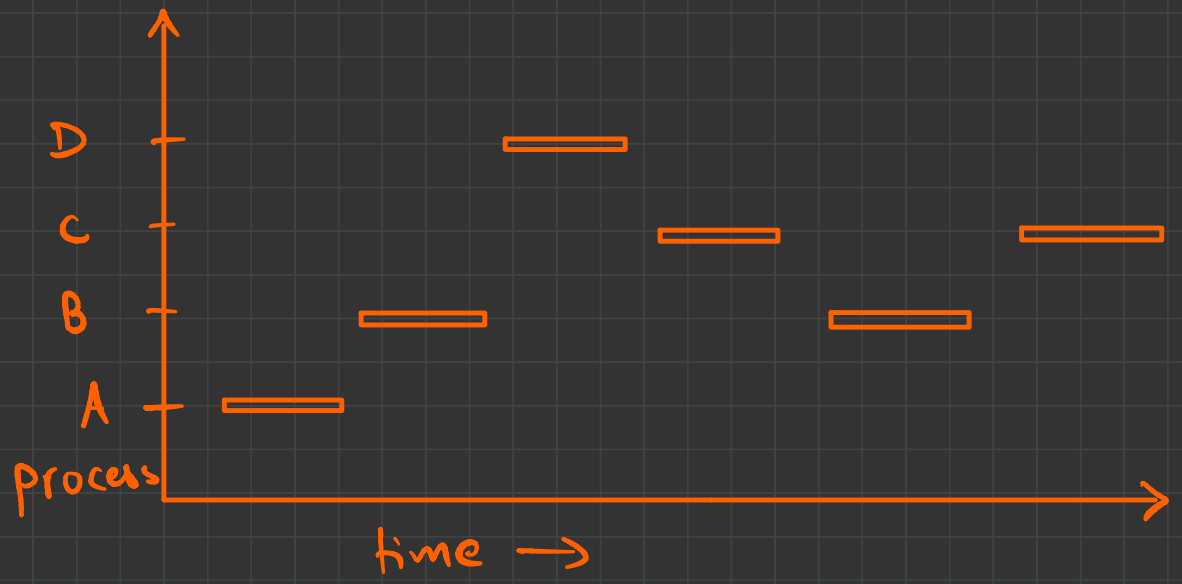
* PROCESSES o->

* Points to remember [imp] :→

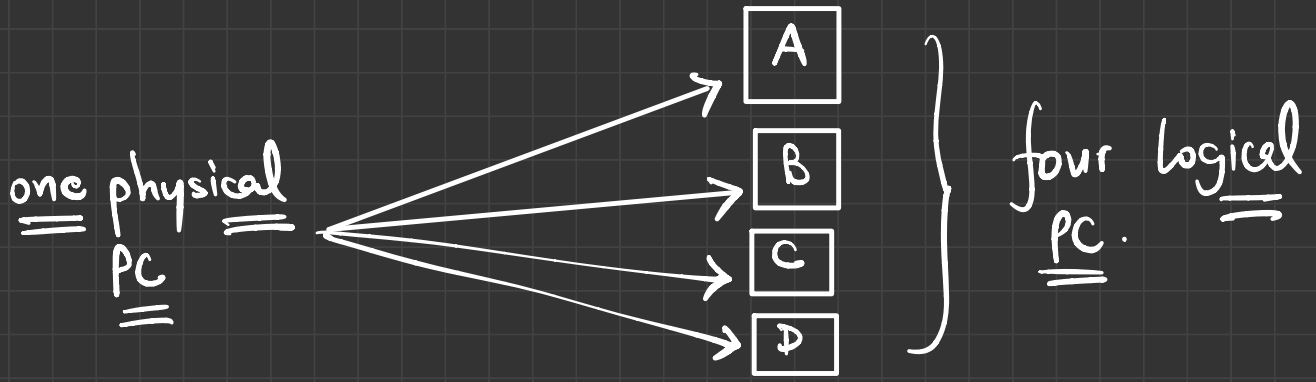
- Running Program { set of instructions } is called a - process.
- One CPU can run only one process at a time
- Using time multiplexing, different scheduling - diff. processes can run on a single CPU but it doesn't mean that single CPU can run multiple processes at a given instant.

  It is called pseduparallelism.

⊙ CPU switches back and forth btw the processes



*



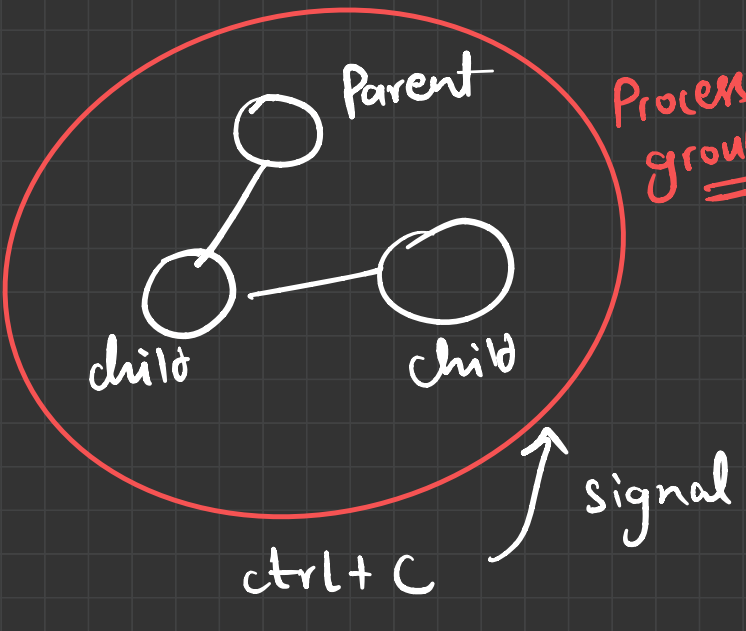
* **Process creation** :-> Processes are created at various times. for ex. when System starts few background processes might run & these are called demon processes.

for ex: fork() can create a child processes & after this call both parent & child will have same memory image. Everything will be copied but both will operate in the different address space.

Note :-> However, It is possible that child can share the open files or fds.

Process Termination :->

a) Normal exit
b) Error exit
c) fatal error
d) Killed by another process



This signal will sent to every process in the process group and it can be blocked, skipped etc.

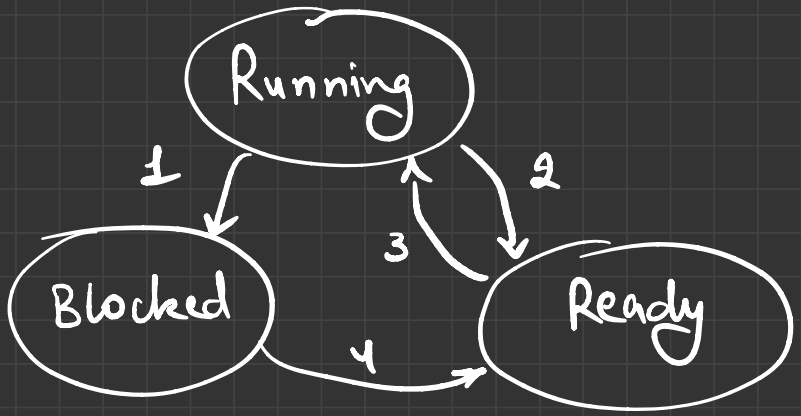
Process States :->

Each process is an independent entity having its own PC, Registers, stack, open files etc.

→ cat file1 file2 | grep tree

* It might happen that "grep" is ready but input is not available so It can not continue logically

States :-



(Running → Blocked) when input is not available process goes into waiting and wait for input

→ In some OS, process might call a sys call to go into blocked state

→ while some listen through pipe/file and goes into blocked if nothing is available

---

(Running ⟷ Ready) this happens due to - process schedular.

* if input is ready process goes (Blocked → Ready) and gets scheduled.

*** Process Implementation** :→ To implement processes, OS maintains a process table OR process control blocks. These tables OR data stru. Contains one process per row having stored process's info like :→

a) process state
b) program counter
c) stack pointer
d) memory allocation
e) status of open files
f) scheduling information

for ex :→ MINIX 3 - OS process table :→

Kernel	Process management	File management
Registers	Pointer to text segment	UMASK mask
Program counter	Pointer to data segment	Root directory
Program status word	Pointer to bss segment	Working directory
Stack pointer	Exit status	File descriptors
Process state	Signal status	Real id
Current scheduling priority	Process ID	Effective UID
Maximum scheduling priority	Parent process	Real GID
Scheduling ticks left	Process group	Effective GID
Quantum size	Children's CPU time	Controlling tty
CPU time used	Real UID	Save area for read/write
Message queue pointers	Effective UID	System call parameters
Pending signal bits	Real GID	Various flag bits
Various flag bits	Effective GID	
Process name	File info for sharing text	
	Bitmaps for signals	
	Various flag bits	
	Process name	

Figure 2-4. Some of the fields of the MINIX 3 process table. The fields are distributed over the kernel, the process manager, and the file system.

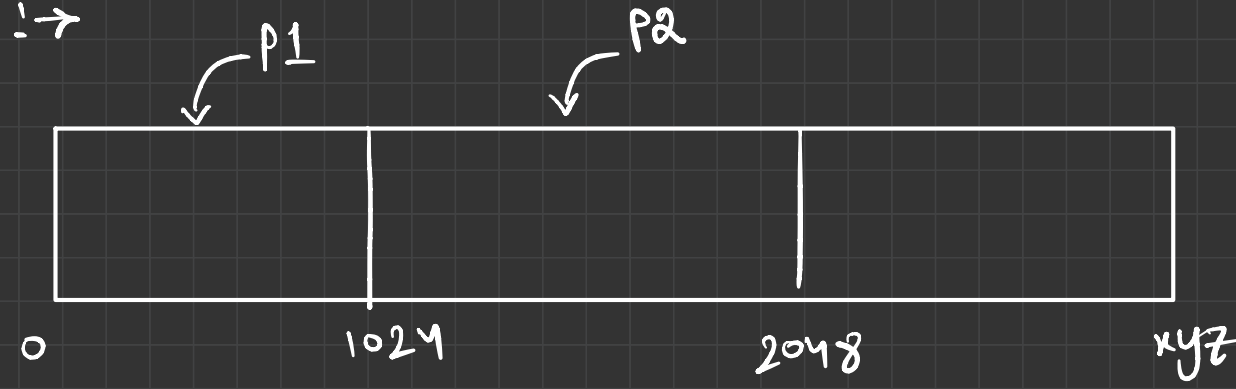
* One entry in the process table have these many field associated

* whenever an interrupt - comes and current process goes to block / ready state then all the info gets stored, so that to retrieve next time when this process will be scheduled again.

* Hardware interrupts are not handled by our High level programming languages.

* instead assembler will run a routine to handle interrupt and run other process.

THREADS :→



⊙ P1 and P2 have their own address space and we can say that they are running a single thread

of control.

- Now, a Process can spawn multiple threads to do things in parallel.

for ex.

P1

```
fd = read()
// operation on fd

int x = 1+2;
print(x)
```

} Thread1

} Thread2

* Now, these threads can do things in parallel
* as Thread 1 reads and do something on file
* Thread 2 doesn't depend on Thread 1.

Optimisations :->

a) Run T1, shift it to ready/block state till file is being read

b) meanwhile T2 can take CPU and do its calculations.

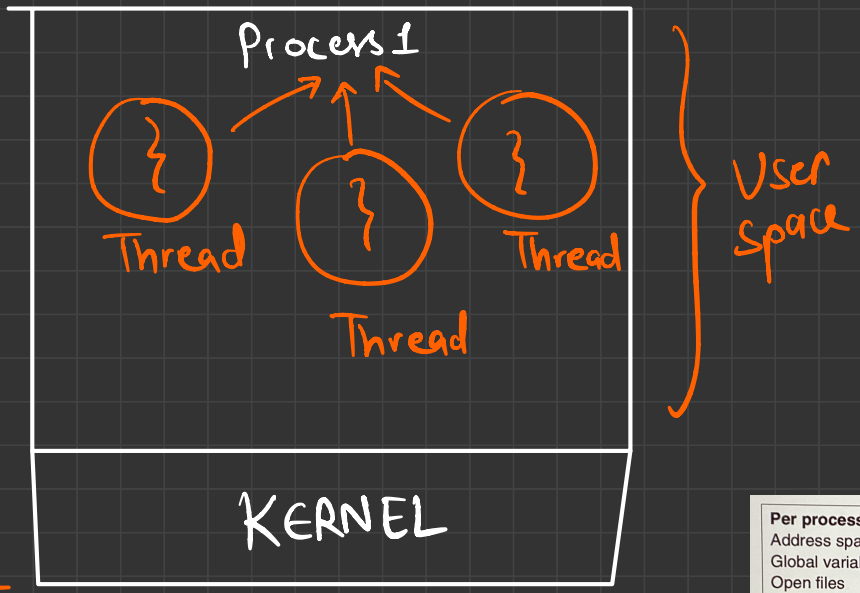
c) when file is ready, T1 can run again.

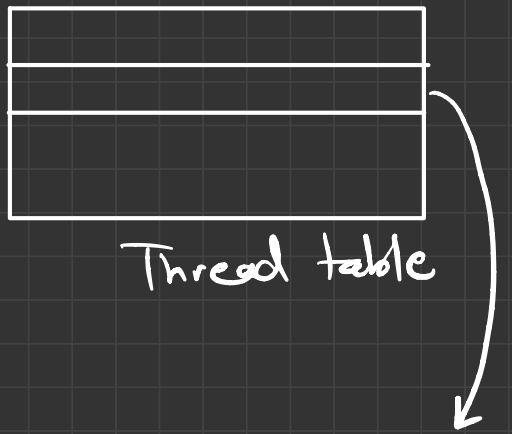
* Threads are lightweight small processes that shares the address space of the process.

* THREAD :→ It has its
  * own PC → to keep track of which instruct^{n} to run next.
  * own registers → which hold current working - variables.
  * own stack → To keep track of its execution History.

Note :→ There are two types of threads → a) Hardware
b) Software

* Virtual threads, GoRoutines uses software threads





Per process items	Per thread items
Address space	Program counter
Global variables	Registers
Open files	Stack
Child processes	State
Pending alarms	
Signals and signal handlers	
Accounting information	

Figure 2-7. The first column lists some items shared by all threads in a process. The second one lists some items private to each thread.

* SOME PROBLEMS TO CONSIDER

* doing thread management in User space is much faster than doing it in Kernel space because sys calls are expensive.
* But doing everything in User space has downside
  - → if one thread blocks for some I/O
  - → Kernel would block the entire process
  - → because Kernel is not aware if process has other threads waiting and they could be scheduled meanwhile
* So, Hybrid approach of both above is used mostly

* if P1 has 3 threads, and after fork() call
  - → should child have all of them too
  - → if yes, what will happen if any of them gets blocked on I/O [input from keyboard]
  - → if user types, will both threads of child and parent will get the input copy?
  - → if one thread is writing a file and other is reading. How to handle this?
  - → if threads are running at user space level,

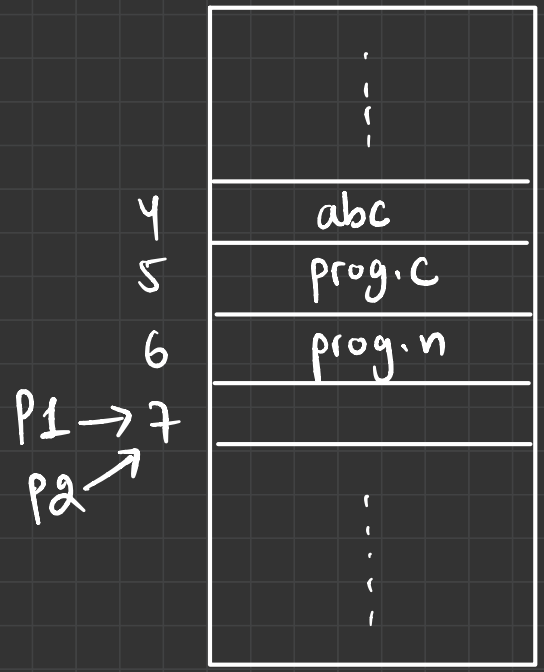
how signals will travel from sys call to the specific thread ?

→ because Kernel is not aware about signal to send to which thread if they are running in the user space.

→ All of these things gets complicated after the introduction of threading model.

→ Some kind of backward campatibity needs to be present.

RACE CONDITIONS :o→



Spooler directory

out = 4 (file to print)

in = 7 (next free slot)

P1

next_free_slot = 7

store file in=7 (override)

in+=1 (8)

P2

stores file in = 7

in+=1 (8)

↳ P2 will not get any output from the file it stored at in=7 because it is overriden by the P1.

→ Hence, P2 will keep waiting infintely at this position on the spooler directory.

CRITICAL SECTIONS :→ when a block of a program access or relies on reading or writing from a sharable resource / file / memory that chunk of code / block is our critical section.

Now, to avoid RACE condition we need mutual exclusion on that shared resource / file / memory.

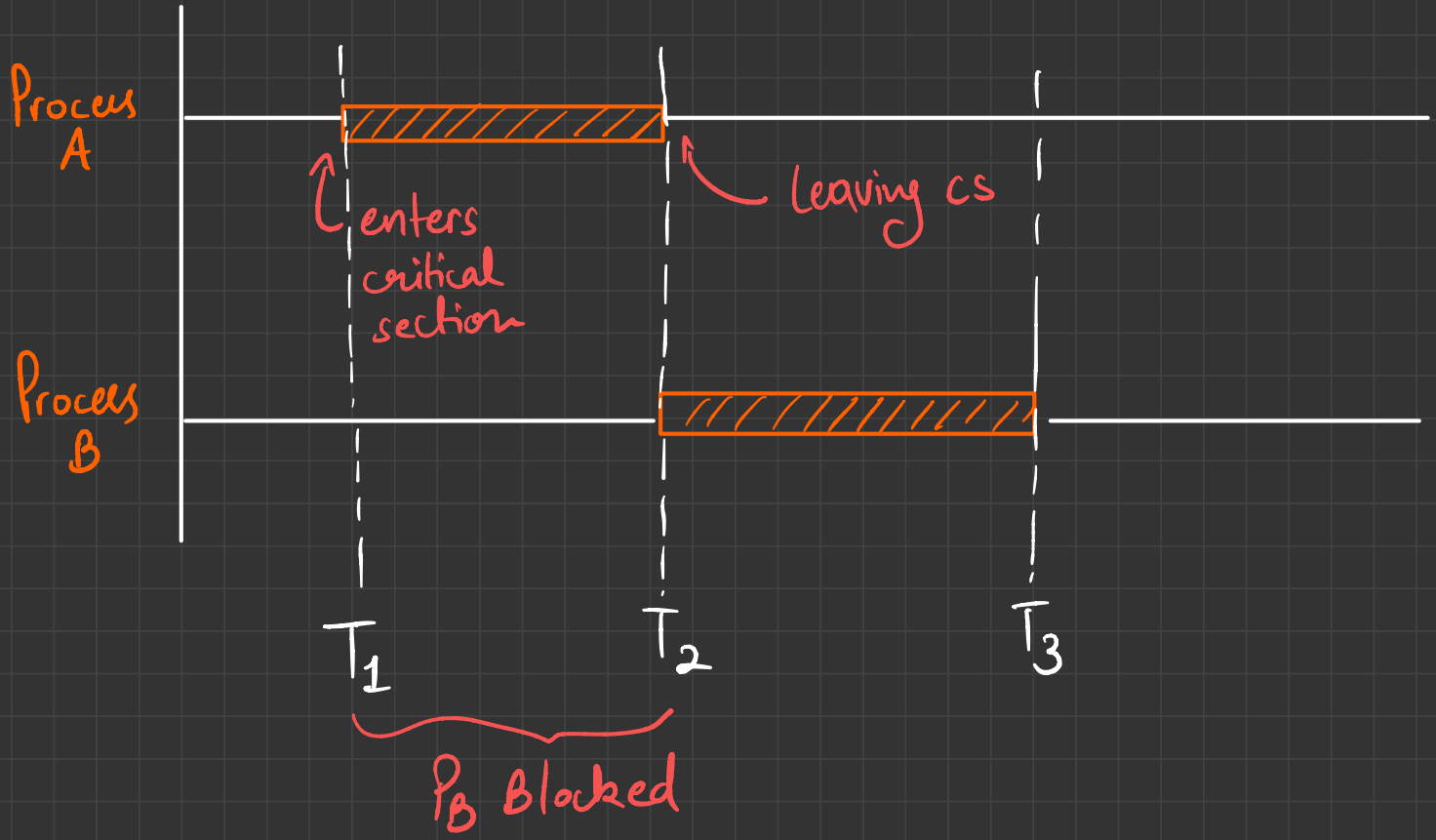
imp* A Good solution must have these 4 conditions to hold

1) No two processes must be inside critical section

2) No assumptions to be made about speed & no. of CPUs.

3) No process running outside the critical - section should block other.

4) No Process have to wait forever to enter its critical section.



* Let's achieve mutual exclusion :->

1) Disabling interrupts :->

- Most obvious solution that when a program runs, we can disable interrupts and make program runs till completion.
- Problem is if this process goes into ∞ loop or so one can't stop it hence wait forever
- 2nd issue is if Process runs on CPU1 and CPU2 can still access the shared memory because interrupts will be disabled on the CPU on which

we ran the instruction to disable.

2) Lock variable :->

```
var lock = 0 ;
```

* Shared variable

```
while (lock != 0) ;
lock = 1 ;
do_something ();
lock = 0
```

Process A or B

P_A ⟶ reads lock = 0

< context switch >

P_B ⟶ { reads lock = 0
set lock = 1

< context switch >

P_A ⟶ { set lock = 1

* Now, Both P_A and P_B are in their critical sections.

reason :→ lock access / read → set lock → run

this is not atomic

3) Wait turn :→ (busy wait)

```
while (TRUE) {
    while (turn != 0)
     critical_region();
    turn = 1;
    non_critical_region();
}
```

```
while (TRUE) {
    while (turn != 1)
     critical_region();
    turn = 0;
    non_critical_region()
}
```

- first continuosly checking turn in while loop wastes CPU cycles unnecessarily.
- Spin lock :→ A lock that uses busy-wait like this.
- P_A ⟶ { turn = 0; runs critical section; set turn = 1 }

<context switch>

P_B ⟶ { turn = 1; runs critical section; set turn = 0 }

* Now, Both processes ran their critical sections and both are in their non - critical sections.

* now, let's assume P_A run its whole while loop again because turn = 0;

$P_A \rightarrow$
- turn = 0
- runs critical section
- set turn = 1
- runs non critical section

< P_B still running its non critical section >

⊙ now, turn = 1 and P_A will run ∞ till P_B completes its execution in non critical section

- This violates our condition 3rd that no process should be blocked on a process which is running its non critical section.

* ——————— *

4) Peterson's Solution [combining the idea of locks & turns]

refer to the code in the video.

{ Basically using two variables decouples the dependency on a single variable due to which context switch in btw can't do anything }

→ It is only valid for two processes

→ Busy wait solution consumes unnecessary CPU cycles.

5) The TSL instruction * optional

<atomic Hardware instruction>

test & set Lock

```
TSL RX, LOCK
```

↳ It reads the content of LOCK from memory & loads into Register RX & then stores non zero value at memory adds LOCK.

```
enter_region:
        TSL REGISTER,LOCK       | copy LOCK to register and set LOCK to 1
        CMP REGISTER,#0         | was LOCK zero?
        JNE ENTER_REGION        | if it was non zero, LOCK was set, so loop
        RET                     | return to caller; critical region entered

leave_region:
        MOVE LOCK,#0            | store a 0 in LOCK
        RET                     | return to caller
```

Figure 2-12. Entering and leaving a critical region using the TSL instruction.

→ will be called by a Process while entering the critical section

↓ will be called while Leaving CS.

6) Sleep and Wakeup :→

* Peterson's Solution can achieve mutual exclusion but limited

* TSL can surely achieve globally not only

at CPU level.

gmp:

* But both solutions are of busy wait behaviour which is consuming our CPU cycles unnecessarily

* They also couples the solution with process - scheduling. Let say P_H and P_L are two processes P_H has high priority than P_L and P_H can run whenever it comes in ready state.

So, when let say P_L is in its CS, then P_H completed its I/O and is ready to run.

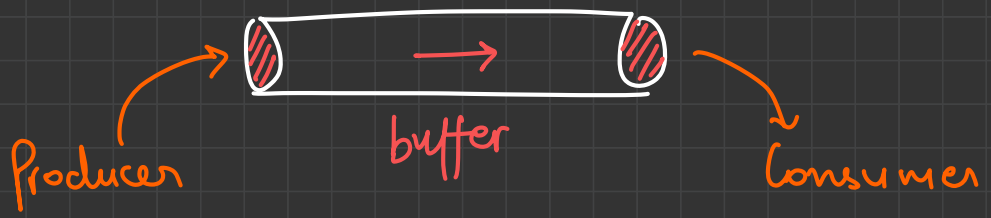
* now, P_H will run and P_L has to wait.
as P_L was running its cs and P_H is scheduled but P_H will wait ∞ as P_L has not exited its cs. [priority inversion problem] *

* ——— *

Solution is to use sleep wakeup sys calls.

Let's understand it with the producer consumer problem.

The Producer Consumer Problem :->




```
#define N 100                                   /* number of slots in the buffer */
int count = 0;                                  /* number of items in the buffer */

void producer(void)
{
     int item;

     while (TRUE) {                             /* repeat forever */
          item = produce_item();                /* generate next item */
          if (count == N) sleep();              /* if buffer is full, go to sleep */
          insert_item(item);                    /* put item in buffer */
          count = count + 1;                    /* increment count of items in buffer */
          if (count == 1) wakeup(consumer);     /* was buffer empty? */
     }
}

void consumer(void)
{
     int item;

     while (TRUE) {                             /* repeat forever */
          if (count == 0) sleep();              /* if buffer is empty, got to sleep */
          item = remove_item();                 /* take item out of buffer */
          count = count - 1;                    /* decrement count of items in buffer */
          if (count == N - 1) wakeup(producer); /* was buffer full? */
          consume_item(item);                   /* print item */
     }
}
```

* Race condition here is also possible despite using Sleep wakeup.

* let say count = 0, Consumer reads count as zero
* Just before going to sleep() consumer gets context switch
* Now, producer will produce & increase count by 1.

* when count = 1, it means buffer was empty and hence producer will try to wakeup the consumer
* consumer was not slept as we know, so now it will go to sleep as at the very first time count consumer read was 0.
* Now, child will sleep and for producer count is 1, hence producer will keep on producing items till "N" and will goto sleep
* Now, Both producer and consumer will sleep - forever.

——— # c=1

* To Solve this (storing no. of wakeups and sleeps) a var called semaphore was introduced.

Semaphores :-> It is a variable having two operations which are down & up.

Pop^n : wait / sleep / down

down :-> decrease counter by 1, sleep if counter < 0

V op^n : signal / wakeup / up

up :-> increase counter by 1, wake up if counter <= 0, means some process / thread is waiting.

IMP.

* Checking the semaphore, increasing / decreasing it then going to sleep or waking other up :-> This

whole operation needs to be atomic.

Solving the producer - consumer problem using Semaphores

>> Refer to the video & code there*

———————————

[optional] :→ Semaphores internals :→

* operating system disables all the interrupts when - checking, updating, putting process to sleep or wakeup
* if multiple CPUs are there, TSL instruction is - used to make above process atomic

* As, we know TSL does use lock variable and locks a process at global level.

* As checking semaphore, updating & putting process to sleep / wakeup is a small procedure :-
  - ↳ Hence, it can block interrupts
  - ↳ or can use TSL incase of multiple CPUs

* Semaphores is a user level procedure for the mutual exclusion.

IMP. *

———

Synchronization premitives :- Semaphores, conditional variables and mutexes.

Semaphores	Mutexes	conditional variable
i) sync premitive that maintains a count and maintains a count to accers shared resource by multiple processes	i) mutual ex. premitive used to protect shared resource by multiple threads	i) cv is a sync. premitive used to make threads wait until a condition occurs
ii) can have a value of zero or any +ve integer iii) Supports two atomic ops: wait(P) & signal(v)	ii) Binary state : locked or unlocked iii) owned by the threads who owns this.	ii) always used with mutexes iii) allows threads to be blocked until notified to resume

iv) can be used both for mutual exclusion and condition synch.	iv) lock(), unlock() and try_lock() ops are there	iv) wait(), notify_one() and notify_all()
Use-Cases a) Protecting C.S b) managing a fixed no. of resources c) signaling btw - threads	Use-Cases a) protecting shared data / resources b) ensuring exclusive access to C.S	Use-Cases a) producer-consumer scenerios b) Any situation - where thread sleeps / waits - until a condition meet.

———

Message Passing :-> We can use message passing - for mutual exclusion and also for effective IPC.

- IPC is a similar problem to producer consumer prbm
- But while communicating btw processes mutual - exclusion, process sync. is hard
- IPC using common data structures might give headache to us but this can be solved - using (semaphore, conditional vars) OR (message passing).

Problems in message passing :→

→ while message passing (P1 → P2) message might get lost in the network.

→ To solve the above problem, we can ask for an "Ack" from P2 (P1 ← P2) to make sure to send the next message from P1 → P2.

→ Problem with "Ack" is that, it might get lost in the network and P1 might send up duplicate message again (P1 → P2)

Soln① → To Solve all the problems, we can use "SEQ" no.s which will prevent drop and duplicate msgs

Soln ②

Or else we can send (P2 ← P1) empty messages from the consumer (P2) to producer (P1) and P1 can fill messages and send to (P2).

(i) if Producer produces faster than consumption then producer will wait until it gets an empty message to fill and send to the consumer.

(ii) If consumer consumes faster than producer - then all empty messages will be sent & will be waiting for the producer to fill them and consumer will wait.

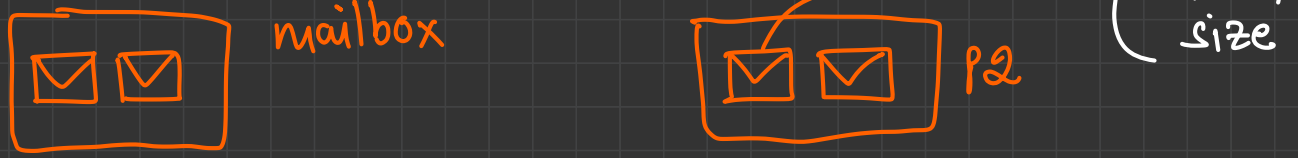
Note:- sys calls [send() / recieve()] are atomic in nature -

So, no need to lock while sending or recieving anythi
because sys-calls are atomic & blocking in nature

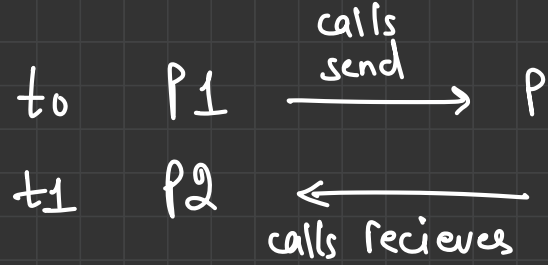
* Solving Producer consumer problem using message passing

```
#define N 100

void producer(){
    int item;
    message m;
    while(TRUE){
        item = make_item();
        recieve(consumer, &m);
        build_message(&m, item);
        send(consumer, &m);
    }
}

void consumer(){
    int item, i;
    message m;
    for(i -> 0 to N){
        send(producer, &m);
    }
    while(TRUE){
        recieve(producer, &m);
        item = take_item(&m);
        send(Producer, &m);
        consume_item(item);
    }
}
```

* IPC can also be solved using some kind of data-structure and we can address processes and hence can store/map the process adds with the message.
* This data structure is called mailbox. mailbox-DS can define capacity and messages can directly goto the process using adds stored.
* Mailbox DS will buffer messages, can be produced and consumed accordingly.



* we can avoid buffering too using rendezvous - system (gt is used in pipes , MINIX and UNIX)



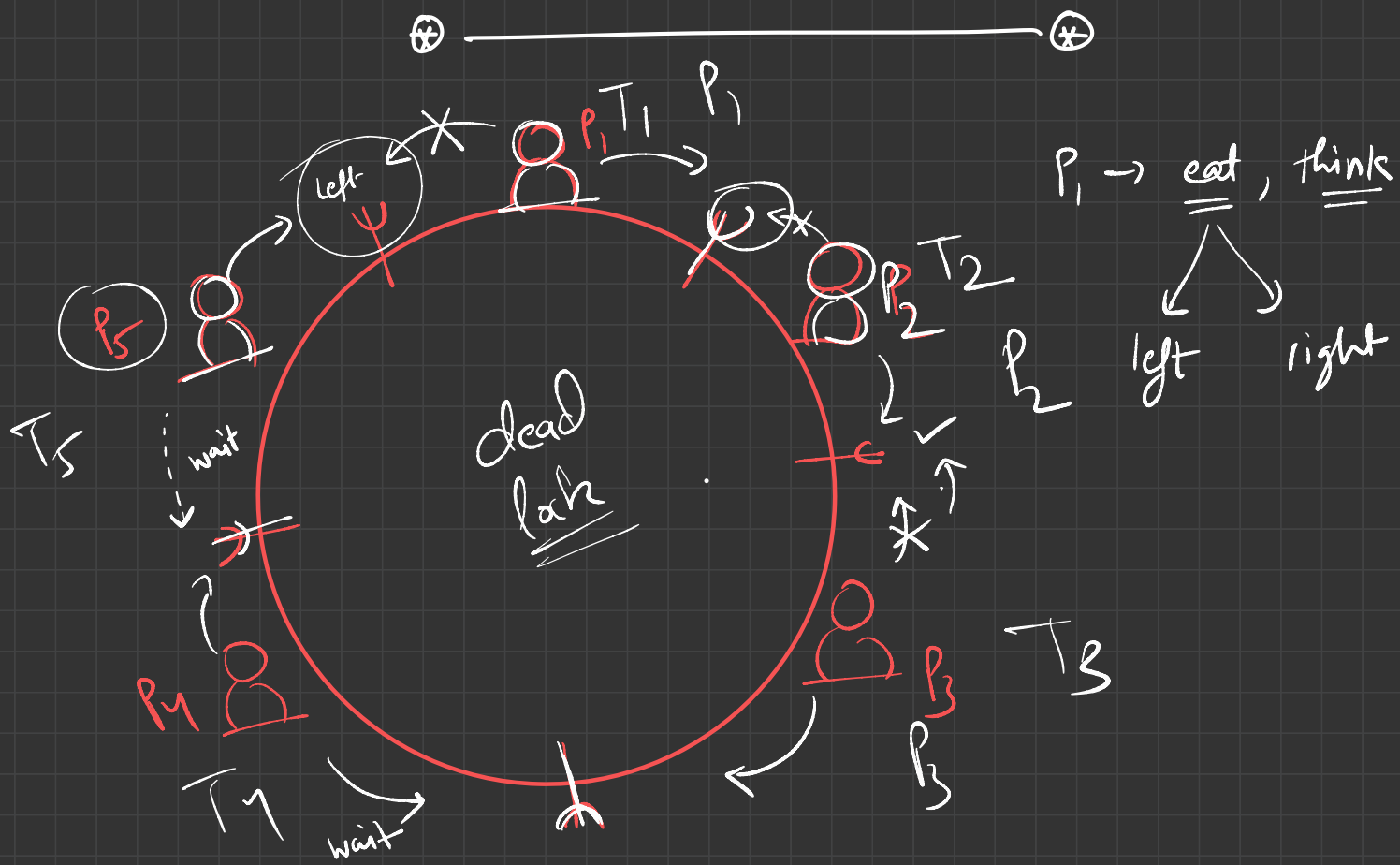
send (destination_pid , &m) ;

recieve (source_pid , &m) ;

NOTE

(i) we will solve "THE DINING PHILOSOPHER PROBLEM" problem in video.

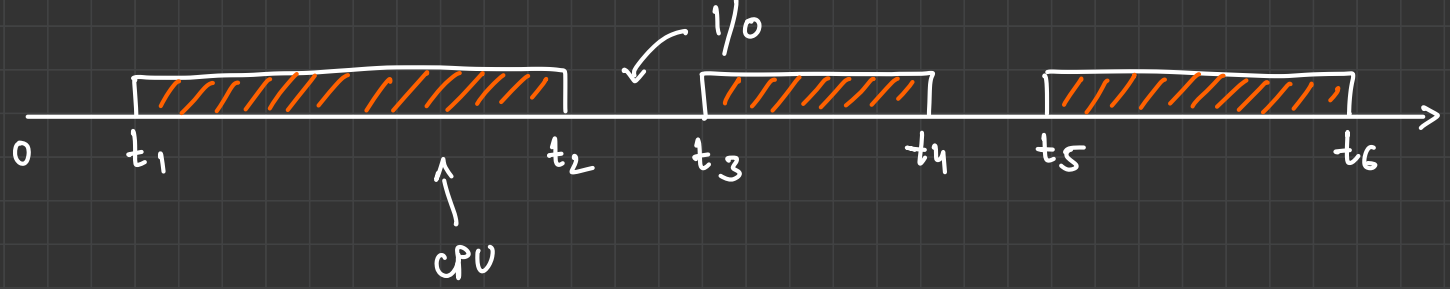
(ii) Homework :- Read and solve "THE READERS & WRITERS PROBLEM"

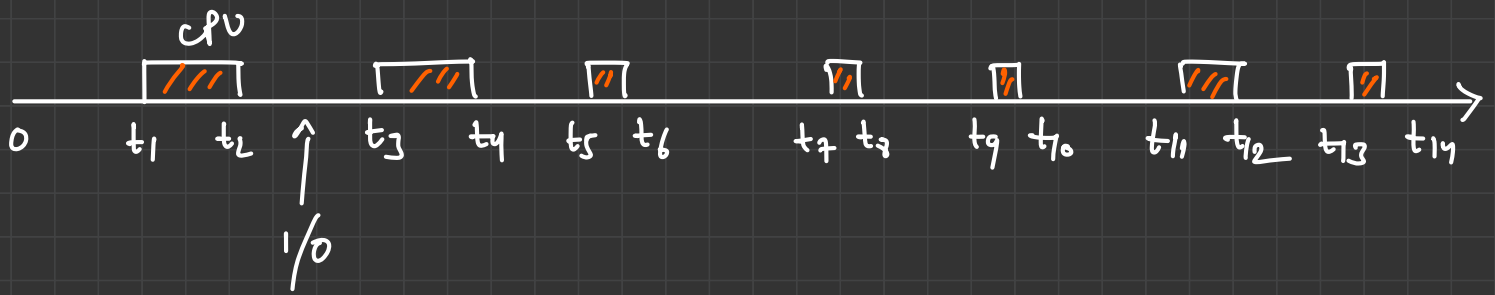

left
P_1 T_1 P_1
P_5
T_5
wait
P_2 T_2
P_2
dead lock
P_4
T_4
wait
P_3
P_3
T_3
P_1 → eat, think
left
right

PROCESS - SCHEDULING :->

There are two types of tasks
- CPU bound
- I/O bound

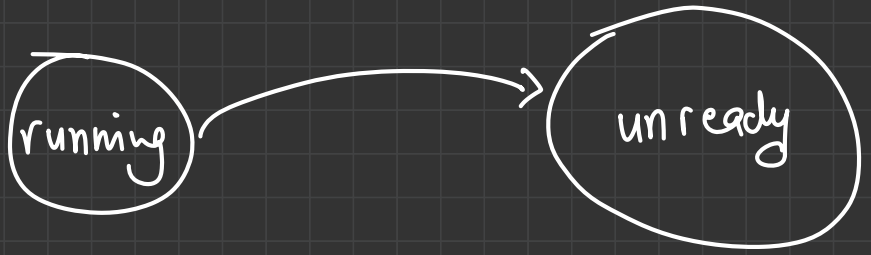
→ CPUs are getting faster than disks more frequently





WHEN PROCESS SCHEDULING IS ABSOLUTELY REQUIRED

- When Process exits
- blocked on I/O or Semapho re



* Process scheduling is required sometimes :→
  - when a new process created.
  - I/O interrupt or clock interrupt occurs.

→ Generally, In the above cases re-evaluating the processes priority is done or another process might

get scheduled.

- I/O interrupt generally occurs when an I/O operation gets completed and process waiting for an I/O gets signal that it can run now.

- clock interrupt means process has run for enough time and now it needs to give resources to some other process

Scheduling algorithms works in two ways while dealing with clock interrupts.

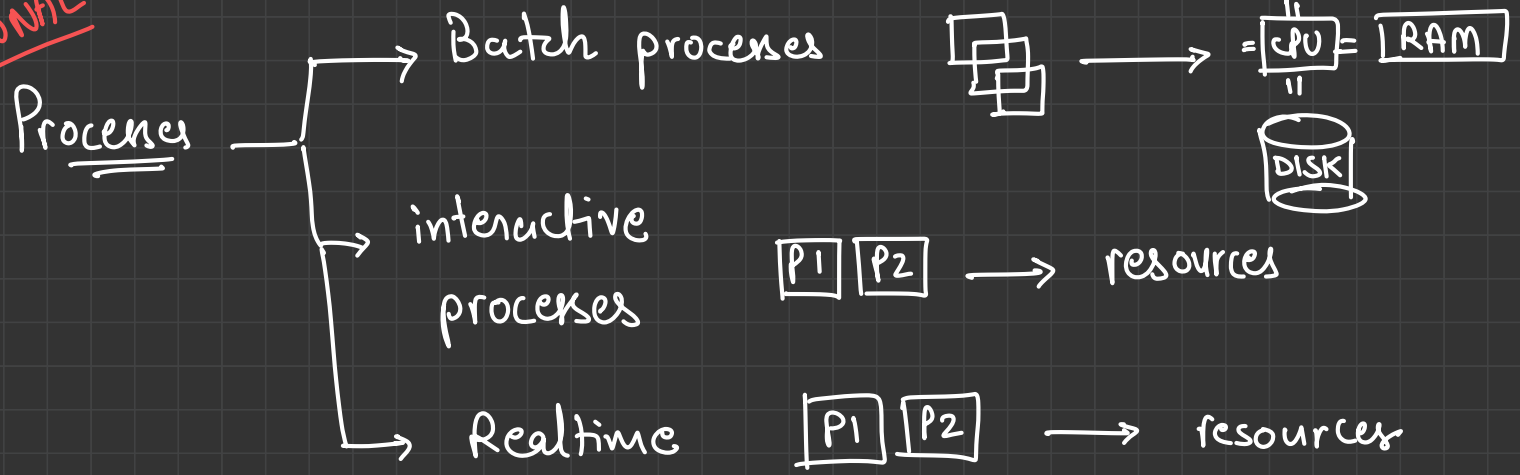
* imp.

Scheduling → Pre-emptive scheduling
Scheduling → Non-Premptive scheduling.

- Non-Premptive scheduling algo :→ It picks up a process and let it run until process becomes blocked on I/O or voluntarily releases the CPU.
- Premptive scheduling algo :→ It picks up a process and gives a time (t) to let it run and at the end clock interrupt occurs to tell the process to give out the resources and let another one to schedu

- Scheduling algorithms differs according to the nature of the process OR environment

* OPTIONAL



- Batch processes are generally not being waited by the end users to get finish ASAP. Hence, non-premptive OR premptive algorithms with longer time periods are acceptable.

- Interactive processes are generally being interacted and it is necessary that they deny service to others and they run but can't run so due to some bug or so. Hence, premption is needed.
- Realtime processes are generally very quick responding processes and they are assumed to run and exit quickly. Hence, premption not strictly needed always.

———

SCHEDULING ALGORITHMS GOALS :→

All systems should carry :→

- Fairness :→ giving each process a fair amount resources for a fair time.
- Policy enforcement :→ seeing policies are being carried out.
- Balance :→ keeping systems/CPU busy all the time

———

Note :→ Having all CPU bound tasks or having all I/O bound tasks is no good. Because keeping all the

components busy means able to complete/run more tasks per second.

→ if we schedule all I/O tasks CPU will be idle (OR) if we schedule all CPU tasks disk will be idle.

→ Having a good mix of both type of tasks is good as compared to one type of tasks.

Metrics which indicates our scheduling algorithm's - efficiency :→

- Throughput
- Turnaround time
- CPU utilisation

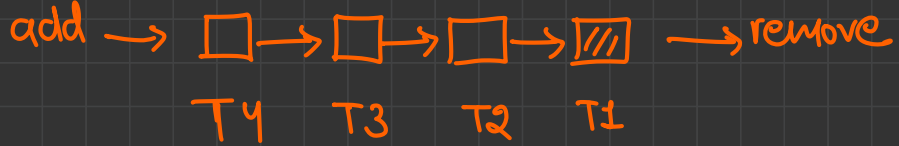
* Turnaround time is the average time from a job submission to its completion

* Throughput is the number of tasks/jobs a system completes per second.

NOTE: (a) A scheduling algorithm OR a system having good throughput isn't mean that the system is more effective because system might be scheduling shortest jobs always and delaying long jobs and hence might give high throughput but at the expense of worse turn around time for long jobs.

⊛ SCHEDULING IN BATCH SYSTEMS :→

a) First come first served :→

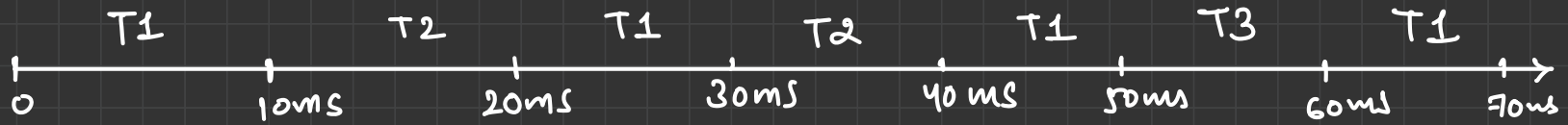


* task got first will be scheduled first.

FIFO → premptive
FIFO → non-premptive

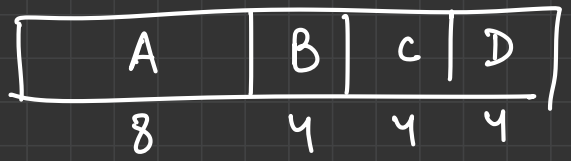
Non-premptive :→ disadvantage is if T1 is a compute heavy task and it takes 1sec to run and T2, T3 and T4 are I/O bound tasks. Then while T1 thinks for long T2, T3, T4 will keep on waiting.

premptive :→ if we give 10ms as premption time for CPU bound task then resources will be used effectively.

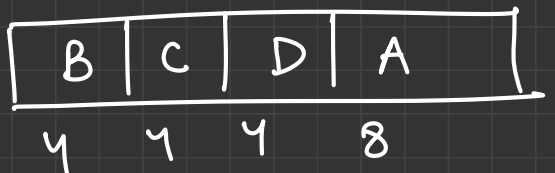


↳ scheduling algorithm is prempted for 10ms and runs - CPU bound task every 10ms.

b) Shortest Job first :→ In this algorithm, it assumes the runtime in advanced which is only possible by - detecting some kind of pattern. [Non Premptive]



a) original order



b) Shortest Job first

Let say they finish job in time :→

JOB	TOTAL TIME
A	a
B	a+b
C	a+b+c
D	a+b+c+d

$$\text{avg time} = \frac{a + (a+b) + (a+b+c) + (a+b+c+d)}{4}$$

$$= \frac{4a + 3b + 2c + d}{4}$$

→ as we can see "a" contributes more as compared to "b", "c" and "d".

→ Hence, running shortest Job first makes a lot of sense.

c) Shortest remaining time next :->

Promptive version of shortest Job first is called - Shortest remaining time next.

→ when a proces is scheduled and a new proces is ready

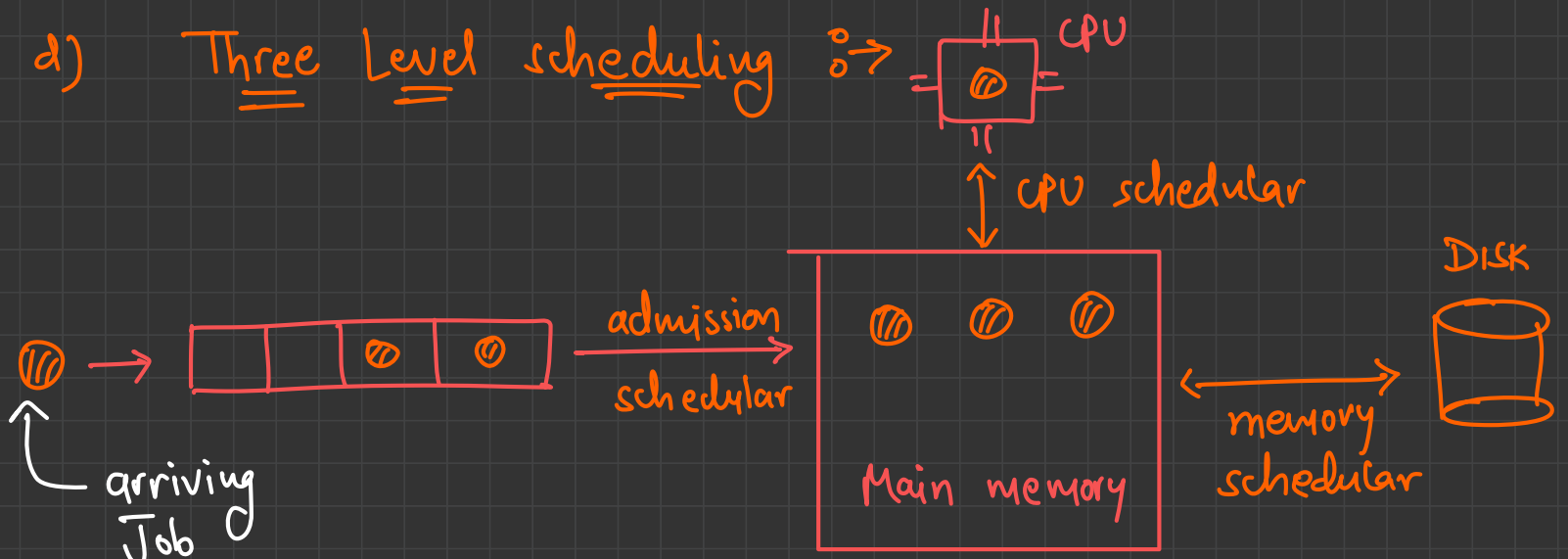
→ It compares the time of current running proces with the time remainiy for new -

process to complete its execution

→ if tnew < tcurrent then it schedules new process

→ Disadvantage :→ Think*

d) Three Level scheduling :→



- Job arrives and stored in the Job queue. This queue is stored in the disk.
- Admission schedular may use FIFO, prioritising mixed I/O and CPU bound tasks or so and selecting the Job accordingly.
- once a job is accepted, it can become a proces and go in the main memory (RAM)
- Incase already there are too many processes in the main memory and hence they might get swapped back to the disk (expensive op^n).

- Due to this swap and all, memory sometimes gets fragmented (will discuss in memory mang. section)

- doing disk swaps and so is expensive and hence memory schedular should be - intelligent enough alongwith other systems and hence not do it many times. (<1 per sec).

To optimize the system performance, memory schedu have to carefully decide that how many processes it can have in the Main memory called "degree of programming".

It can use some criterias to do its work effectively :->

a) How long process has been swapped in/out

b) How much time process consumed CPU for recently.

c) How big the process is

d) How imp. the process is

Third Level of scheduling is CPU scheduler. Any algorithm can be used here (discussed - above)

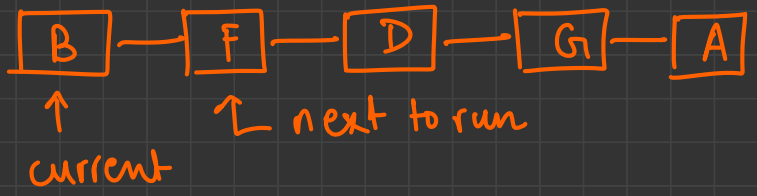
———————

SCHEDULING IN INTERACTIVE SYSTEMS :->

Interactive processes :-> with which user generally interacts and they wait for the user input and try to respond asap! Priority of interactive process is more than Batch & less than real time proces

a) Round Robin Scheduling :->

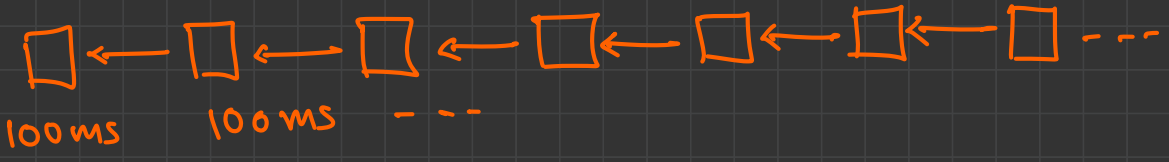
* Interactive process are generally scheduled by the algorithms which give less response time as possible.



- every process is given a a time to run called - quantum.
- Process keeps on running till that quantum
- CPU prempt the running process and run - another one if current procen blocks or runs till its quantum
- A list of runnables is maintained and when a process runs till its quantum then it got pre-empted and added at the end of the list

- But the problem in round robin is deciding the quantum and switching context when running - processes got switched takes time because of saving loading registers, memory maps, flushing & reloading memory cache etc
- to reduce the switching time, quantum can be increased (say 1ms to 100 ms)
- But problem with large quantum is, for ex:- assume user hits 10 carriage return keys at same (almost) time in 10 processes

- Now, all of them will be runnables and few processes (early ones) will quit immediately while others will not respond asap!



Conclusion :→

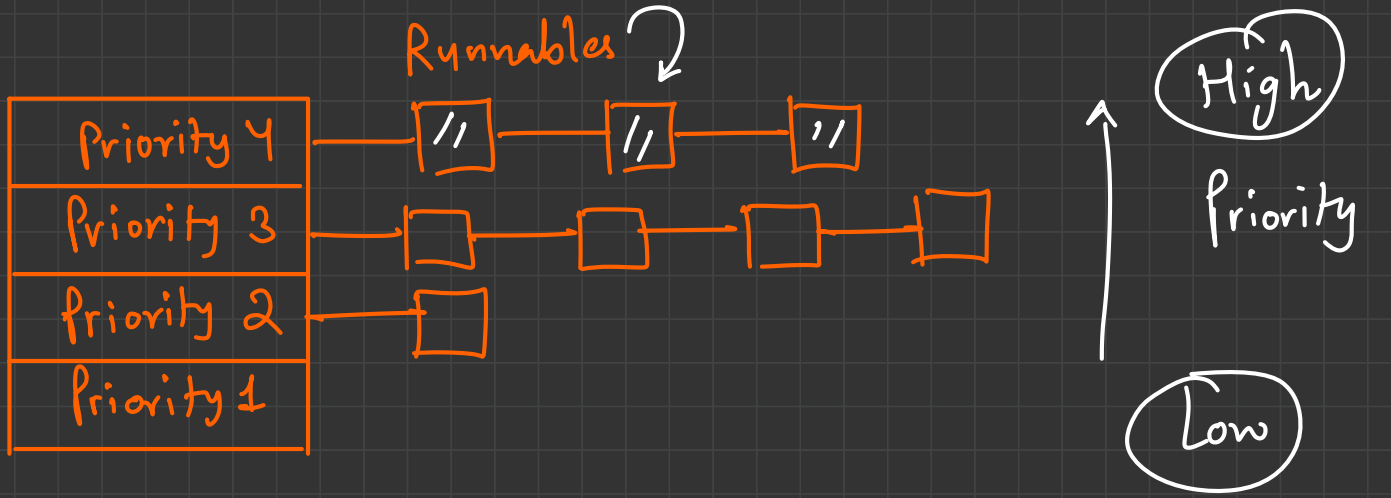
a) short quantum → quick response but low CPU efficiency due to frequent switches

a) large quantum → poor response time but CPU efficiency increases with less switching.

b) Priority Scheduling :→

- Process in the same system may have different priorities
- Running high priority tasks first makes sense but high priority tasks may run infinitely.
- Hence, decreasing priority of the tasks makes sense and grouping similar priority tasks and running them first and reduce their priority when they run for a quantum.

↓



- run P4 each for a quantum
- decrease their priority
- recalculate their priority and shift them based on their new priority.

↓

*optional

* in old times, an advanced scheduling was discovered which was :->

a) if a Process needs let say 100 quanta to complete its execution

b) multiple queues approach taken. where process first allowed to run for 1 quantum then moved lower

c) next time it wound run for 2 quanta and context switched. next time it will run for 4 quanta, then 8, then 16, 32 etc. till its 100 quanta completion

d) and if user type carriage return in any process (terminal) its priority will become highest and will be moved to top priority queue.

optional *

c) Shortest process next :-

Q Similar to the batch processes scheduling we can use it right ? As it provides really - good response time.

Ans Yes, But interactive systems are more like "run, wait for user command, execute, repeat."

predicting shortest Job next here is difficult

→ But estimation (aging) can be used here

→ Let say estimated time initially is T_0 and next run estimated as T_1. Now we can estimate time by taking "$aT_0 + (1-a)T_1$" and can update our estimate using this formula.

$a = 1/2$

T_0, $T_0/2 + T_1/2$, $T_0/4 + T_1/4 + T_2/2$, $T_0/8 + T_1/8 + T_2/4 + T_3/2$

* we can set "a" accordingly to remember old - estimates for longer time or lesser time (aging)

d) optional * Guaranteed Scheduling :→

→ It is related to promising a user to run "n" processes with some performance.

→ for ex :→ to run "n" tasks on one CPU with 1/n share of CPU entitled to each task we can calculate how much CPU task was entitled to and how much it used.

→ This ratio can be used as an estimate to schedule next task.

→ Ration = 0.5 means process has consumed half

what it was supposed to

→ Ratio = 2.0 means process consumed twice the CPU it was entitled to, hence now — scheduler will schedule task/process with lowest ratio and make its ratio moved ↑ to its — closest competitor. (above)

Optional*

e) Lottery - scheduling :→

→ Here, we distribute lottery tickets to processes

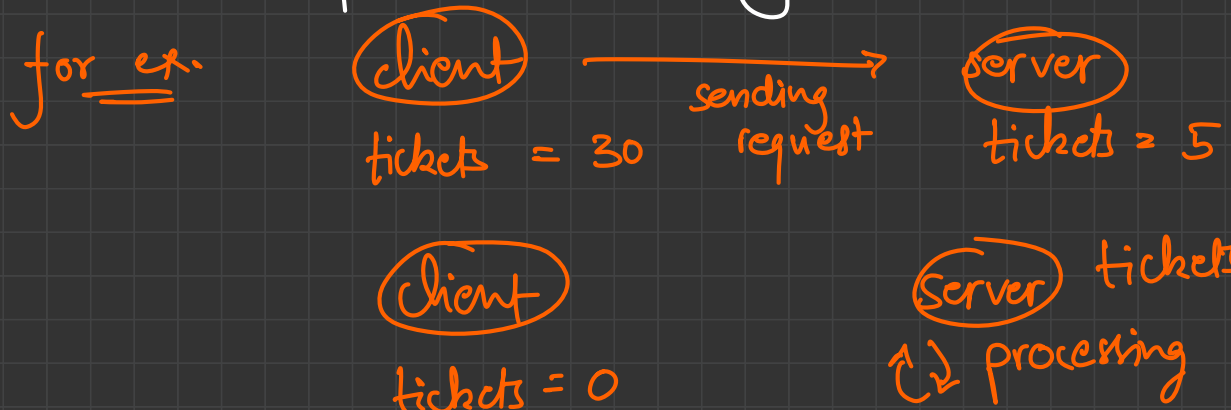
→ Having 20 lottery tickets to one process out of 100 means this process will consume around

20% of CPU in future when scheduled.

→ It is similar to priority scheduling but here it makes sense that having "f" lottery tickets means that process will get f% of the resources in the long run.

→ processes may also exchange tickets.

for ex.



→ Lottery scheduling can be used incase of video streaming also

for ex. Video server sending video at diff. frames to the Client

10, 20, 25 fps

tickets distribution in ratio 10:20:25 will give resources according to the proportion/load and processes scheduling will be proportional to the no. of tickets now.

↓

f) optional*

Fair share scheduling :→

- in earlier scheduling we assumed only one user is the owner of all the processer
- But what if multiple users having shared - resources (CPU, memory etc) to schedule the processer
- user 1 :→ A, B, C, D user 2 :→ E

  CPU :→ (50 : 50)

Ex: scheduling (Round robin) :→ possible scheduling :→

A E B E C E D E A E B E C E D E ---- (50% CPU)

* multiple scheduling algos. may also be used based on the CPU share b/w the users.

optional

SCHEDULING IN REAL TIME PROCESSES :→

Real time processes generally have hard and soft deadlines in which processes need to schedule and complete their execution.

hard - deadline → It needs to be met

soft - deadline → It is tolerable if not met.

- When an external event comes it is the job of the schedular to schedule the task
- Events can be categorized as periodic and aperiodic

A real time event stream occuring at periods P with i as the event if it needs Ci sec of CPU to complete, then these periodic events are - schedulable if :→

$$\sum_{i=1}^{m} \frac{C_i}{P_i} \leq 1$$

* ——————————— *

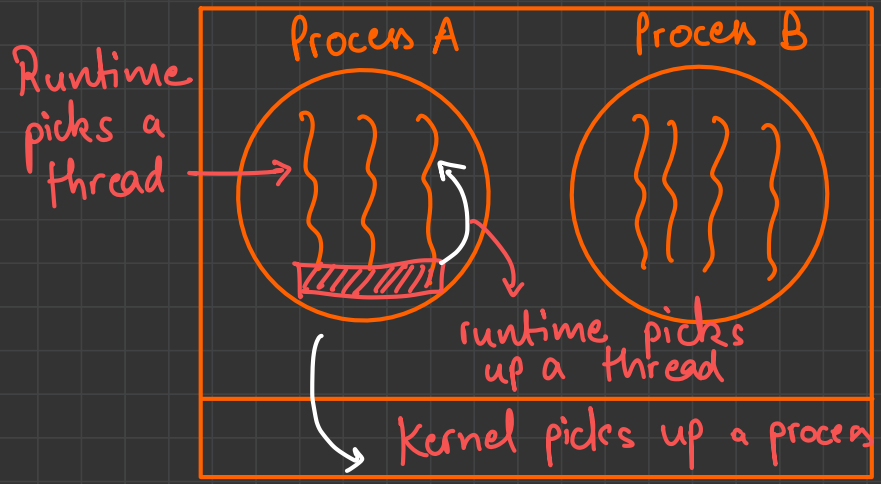
THREADS SCHEDULING :→

When you have multiple processes with multiple - threads then we have two level of parallelism
→ Threads and Processes

Scheduling also differs incase of user level or kernel level threads

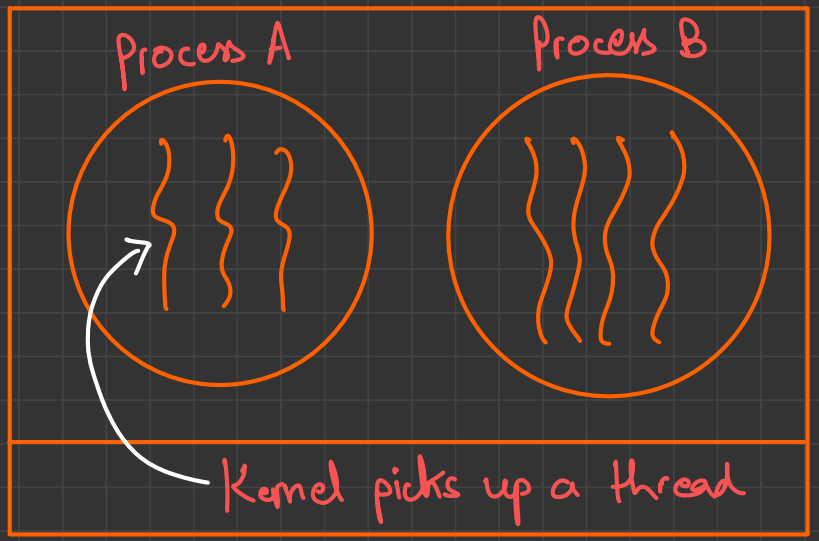
↓

* User Level threads scheduling :-> Here, Kernel doesn't know about different threads of a process and Kernel picks up a process instead of threads and runtime of the process decides the order or which thread to run (ex. Virtual threads of Java)



* Kernel picks up a process as it is unaware of user threads and hence - user's scheduler decides which thread to schedule.

* Kernel level threads



* Here, Kernel can directly schedule the threads

Here, as becz of Kernel threads Kernel can - prempt the thread also as it can interrupt threads with an interrupt here.

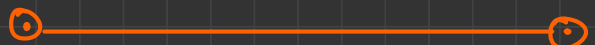
* Some differences :->

- Context switch in user level threads takes few instructions

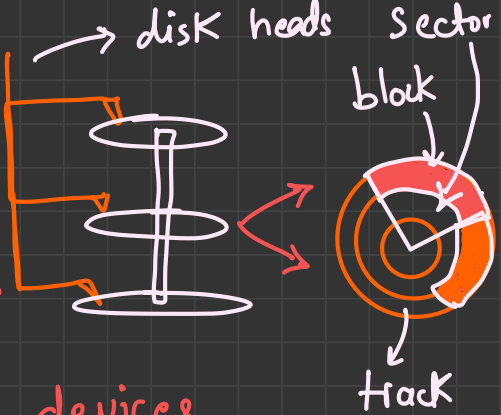
- Kernel level thread switch have to switch memory map invalidate cache which is slow

- Thread waiting for I/O will hang the process itself.
- User specific threads employ user level schedular. It can use this info intelligently. for ex: in a webserver if there is one dispatcher thread and two worker threads than schedular in runtime can schedule dispatcher thread and then a worker.
- Kernel level thread will not block the whole process when I/O wait comes.
- Kernel knows that switching from one thread of process A to the thread of process B, It is faster to switch a thread in same process A. It can use this info while scheduling / context switching.

* In general, application level thread schedulars can tune an application much better than Kernel bec? Kernel has no idea what a thread did as - compared to the appl^n schedular context that knows what diff. threads in an appl^n does.



* Input/Output :→ I/O



IO devices
- Block devices
- Character devices

optional

a.) Block devices are the ones which stores info in blocks. block size ranges from 512 bytes to - 32768 bytes.

The main property of block devices is that it is possible to read / write blocks independently of others.

b.) A character device delivers or accepts a stream of characters and doesn't have any seek - operation. Ex. printers

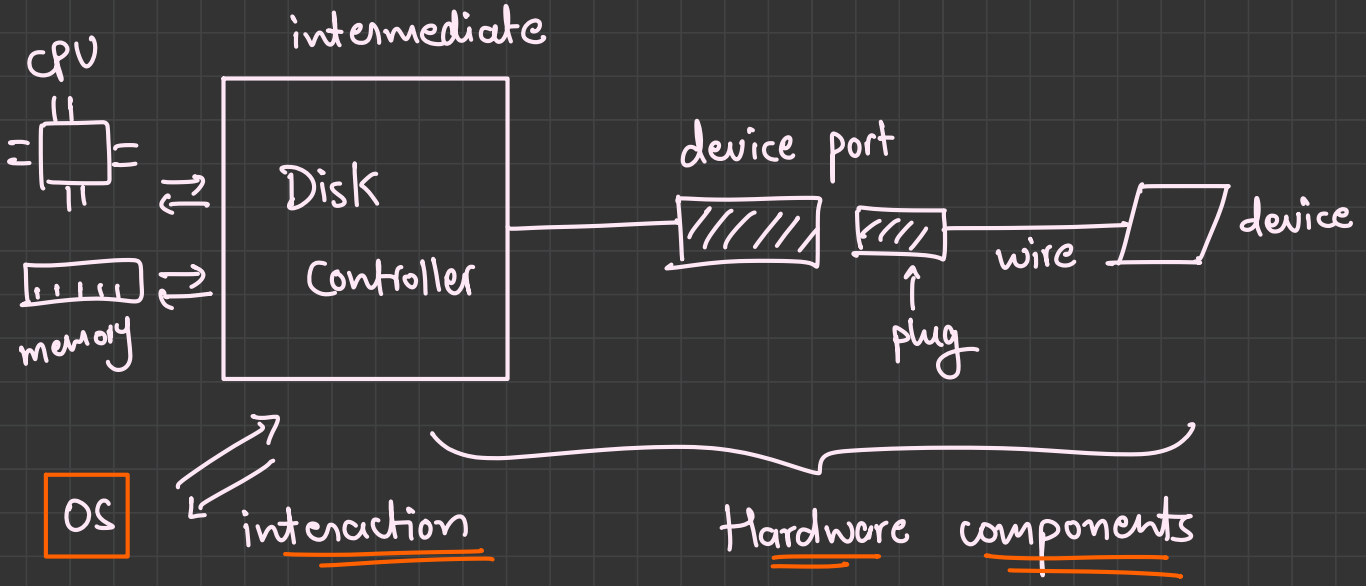
Most other devices that can't be seen as block devices can be assumed as character devices

* Some devices are not classified in either of them for ex. clocks. They are not block addressable & neither they produce character streams. All they do is interrupt someone at an interval

Some cool examples of data rate :->

Device	Data rate
Keyboard	10 bytes/sec
Mouse	100 bytes/sec
56K modem	7 KB/sec
Scanner	400 KB/sec
Digital camcorder	4 MB/sec
52x CD-ROM	8 MB/sec
FireWire (IEEE 1394)	50 MB/sec
USB 2.0	60 MB/sec
XGA Monitor	60 MB/sec
SONET OC-12 network	78 MB/sec
Gigabit Ethernet	125 MB/sec
Serial ATA disk	200 MB/sec
SCSI Ultrawide 4 disk	320 MB/sec
PCI bus	528 MB/sec

Device controllers :->



The interface btw controller and device is low-level. device controller might work like :⇒ device is storing information in block manner but what comes from the drive is a serial bit stream then controller will batch those bits in a block using its buffer and

then storing. It might cache data as soon as disk - head reaches the correct cylinder for future data read calls.

Memory Mapped I/O %→

- Each controller has some registers for communicating with the CPU. [control registers]
- Operating system can interact with controller by writing into these registers like read data, accept data, switch on/off device etc.
- OS can read device state from these registers too like if it is prepared to accepts another

command (or) not.

⊙ Another type is data register that OS can put data or read. for ex. Showing Pixels (monitor screen) putting data (state) of pixels into data registers of VRAM.

? But how does CPU interacts with these registers like reading/writing ?

ist option :→ assign a port no. to the registers - generally a 8, 16 bit integer and use those ports in a specific I/O instruction as :→

IN REG PORT

CPU reads from port (reg) and stores in its reg REG

OUT PORT REG

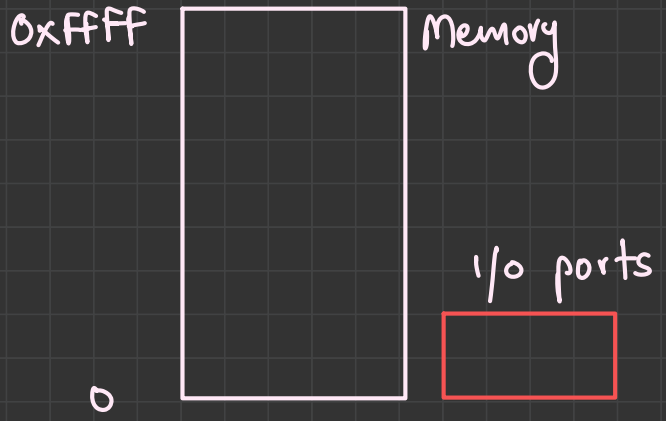
CPU writes contents of Reg to control register mapped by PORT.

ex. Computers like IBM 360 used this.

2nd-option :→ On other computers, I/O registers are a part of regular memory address space and this scheme is called memory mapped I/O.

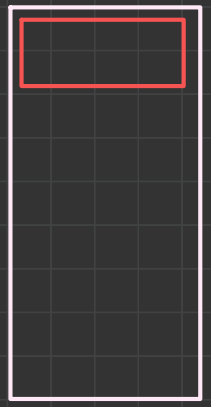
- Here, I/O registers are part of regular memory space

⊙ Each controller register is assigned a unique memory adds. to which no memory is assigned



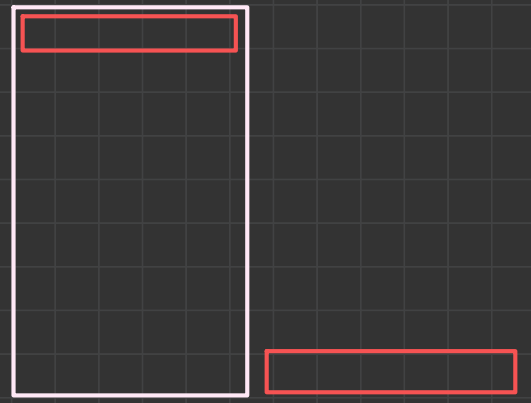
Two addresses

a) Separate I/O and memory space



One-address space

b) Memory-mapped I/O



Two address space

c) Hybrid (memory mapped → data buffers

Intel pentium uses this hybrid approach

(separate I/O ports for → control registers)

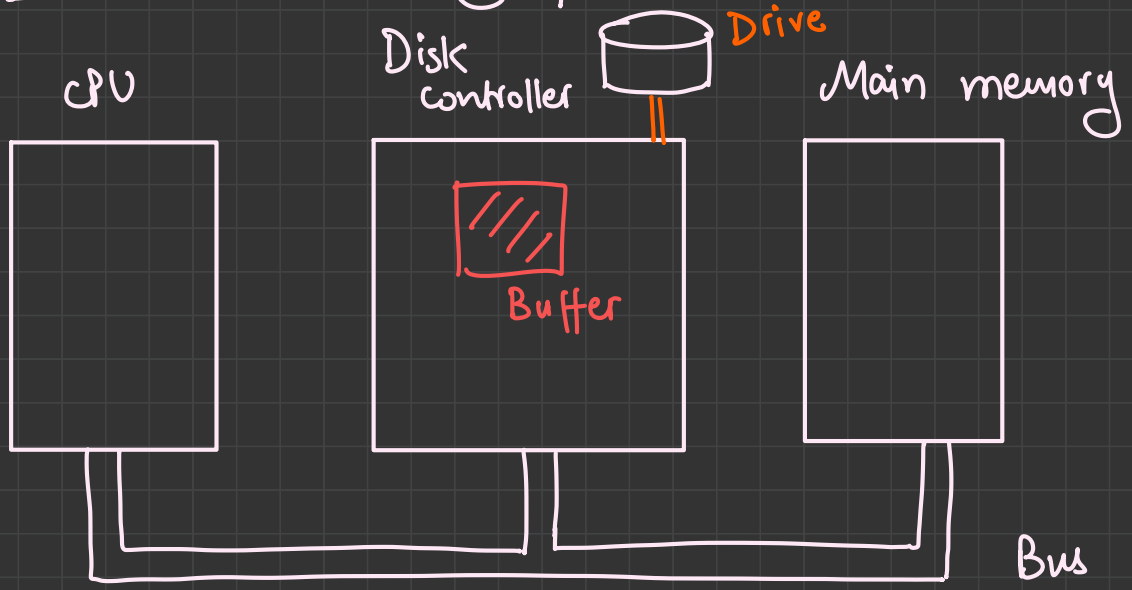
* I/O devices either uses status bit which CPU can read and understand that I/O device is ready to take or produce data.
* CPU will keep on checking until device is ready which is called polling or busy - wait
* Above technique is not effecient as CPU will - waiting unnecessarily if the device is busy.
* hence the another technique used is "Interrupts" which is triggered by the device and communicated to the CPU.

* When device generates interrupt, It does so by using "IRQ" [interrupt request] which is a piece of code that is being run to notify the CPU
* CPU then runs "IHR" (interrupt handler, a small piece of code) & informs the OS that device is ready.

* ——————————— *

Direct Memory Acces (DMA)

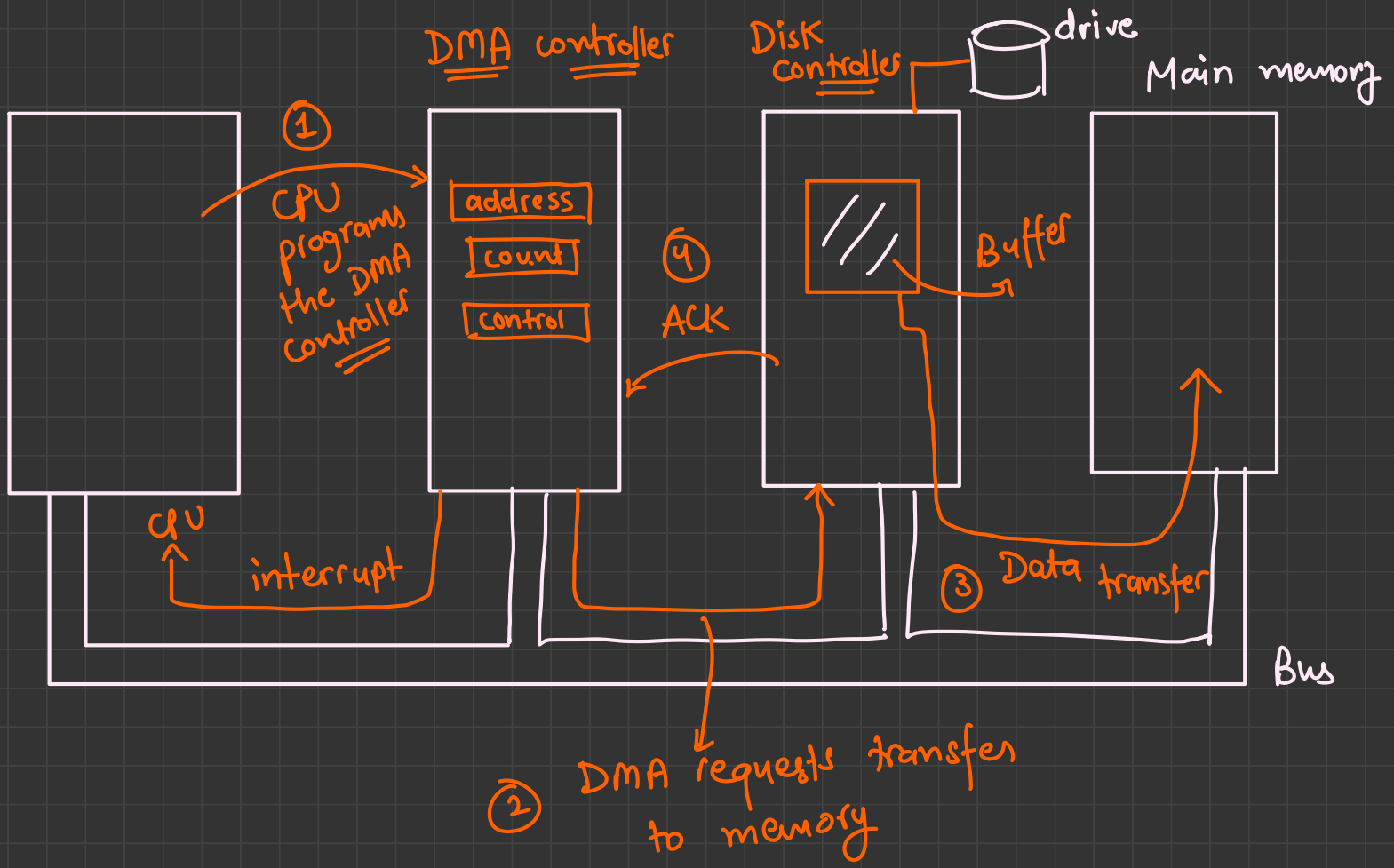
* without DMA (reading from Controller)



① first controller reads a sector or so from the drive serially (bit by bit) until the entire block storage in controller's buffer.

- Next it checks the checksum to check if read is complete without errors
- Now, control causes an interrupt.
- OS now reads from controller's buffer (a byte or word at a time) stores in main memory, — incrementing memory adds and decreasing the count of the bytes to read until it reaches zero.

* with DMA (reading from the device) :->


DMA controller
Disk controller
drive
Main memory
1
CPU programs the DMA controller
address
count
control
4
ACK
Buffer
CPU
interrupt
3
Data transfer
Bus
2
DMA requests transfer to memory

- first CPU programs the DMA controller by setting up its registers so that it knows what to transfer.
- It also issues command to Disk controller to read data from disk and store in its buffer and verify the checksum.
- DMA then starts read by making read call over the bus to the disk controller.
- Now, Disk controller writes the buffer content in the main memory. and issues a call to DMA to tell that operation is complete. (sends an ACK over the bus)
- DMA then increments the memory adds. to use and

decrement the count to read, until count becomes zero.

- Now, DMA causes interrupt and OS donot have to do anything as data is already there in the memory.

* ——————————————— *

PRINCIPLES OF IO SOFTWARE :->

GOALS :->

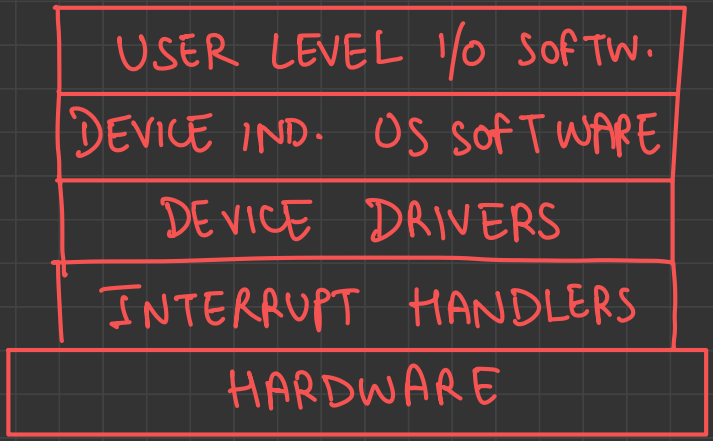
a) device independence [writting to HD or floppy]

b) uniform naming [/usr/ast/backup]

c) error handling

ex. a floppy disk can be mounted on this dir.

Layers of I/O software system



Layers

* DRIVERS needs to know position of tape reader, sectors, tracks etc. to perform any read/write (let say) operation

Hence, any device attached to a computer needs a device specific code to control it which is called device-driver. Each driver handles specific categories of devices.

For ex. A disk driver should support multiple-types of disks, with diff. speed or probably CD ROM too.

In general, Job of device driver is to accept abstract requests from the above device independ. software and execute that request

- first step of driver is to check if input params are correct or throw error if not
- then coverting this abstract request to concrete terms.
- if there is already a request executing then "it should" buffer / put in queue the request.
- Once driver is ready, and it submits / issue the commands to controller through controller's registers

- So, now after issuing command driver waits until device gets work done and it wakes up after an interrupt.
- if unable to complete, driver raises error. Otherwise transfers the data back to the device indep. software.

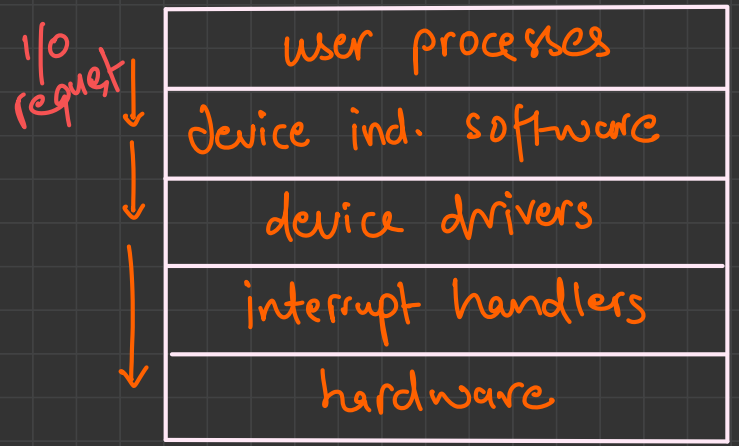
* DEVICE INDEP. SOFTWARE has following functions :->
  - → Uniform interfacing for device drivers
  - → Buffering
  - → Error reporting
  - → Allocating and releasing dedicated devices

→ providing a device independent block size (ex.)

* HOMEWORK :→ Read about spooling directory & deamon processes & how they work to print data let say to printer.

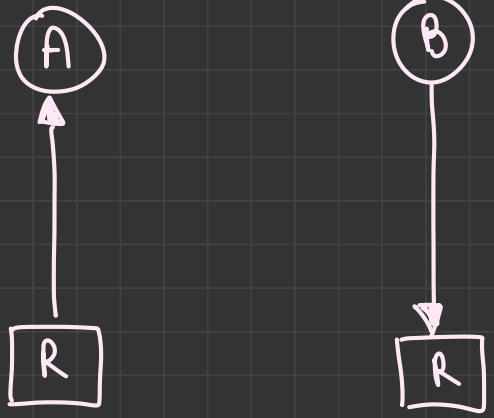
spoiler :- A new project coming soon

fns of each layer of I/O system



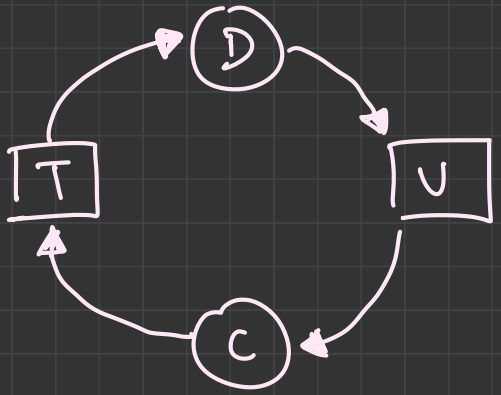
- make I/O req., format, spooling
- Naming, protection, blocking, buffering and allocation
- check status, setup device registers
- wake up driver when I/O comple
- perform I/O operation.

* Deadlocks * Imp.



⊙ A has resource R.

⊙ B is requesting resource R.



⊙ Deadlock

think !! does it mean having a cycle in a directed resources graph mean deadlock ?

Conditions to hold for a deadlock to occur :->

a) Mutual exclusion :-> Each resource is either currently assigned to one process OR is available

b) Hold & wait condition :-> Processes currently - holding resources that they requested earlier can request new resources.

c) No premption :-> resource allocated to a user - shouldn't be taken forcefully, they should be explicity released by the process.

d) Circular wait condition :-> There must be -

circular chain of two or more processes waiting for a resource held by the next chain's member.

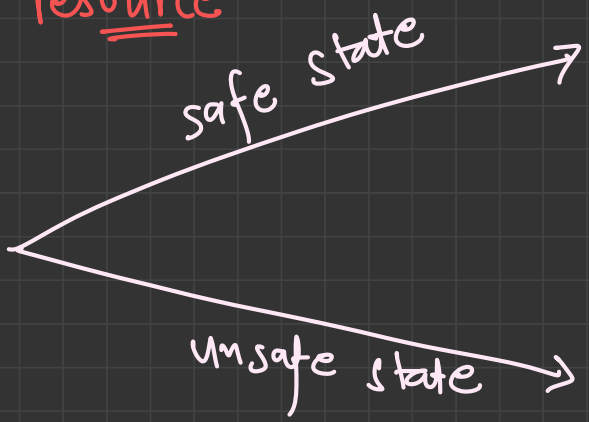
=

BANKER'S ALGORITHM :→

for single resource

A	0	6
B	0	5
C	0	4
D	0	7

Free : 10



Free : 2

A	1	6
B	1	5
C	2	4
D	4	7

A	1	6
B	2	5
C	2	4
D	4	7

Free : 1

Banker's Algorithm for multiple resources :->

Processes Tape cdrom Printer .....

A	3	0	1	1
B	0	1	0	0
C	1	1	1	0
D	1	1	0	1
E	0	0	0	0

Resources assigned

A	1	1	0	0
B	0	1	1	2
C	3	1	0	0
D	0	0	1	0
E	2	1	1	0

Resources still needed

* To check if state is safe or not :->

↓

- Look for a Row, whose unmet resources are equal or smaller than the process A. if no such Row exists then it is a deadlock
- Assume the choosen Row (process) gets all the resources it needed then after its completion, all resources must freed and can added upto process A resources.
- repeat above step 1 and 2 until all processes termination.

#

* Memory Management :->

Imp. most computers have memory heirarchy like :->

- small, fast, volatile mem → cache (L1, L2) Read more
- hundred of megabytes, significantly fast, volatile → RAM
- hundred of gigabytes, slow, cheap, non-volatile → HDD (or disk storage)

* It is the job of OS to manage these diff. mem. for ex. swapping data from RAM → disk & vice versa when there is no much space left.

BASIC MEMORY MANAG. :->

Memory management systems

- (a) ones who moves processes back and forth btw RAM and disk (swapping and paging)
- (b) ones who donot swap stuff

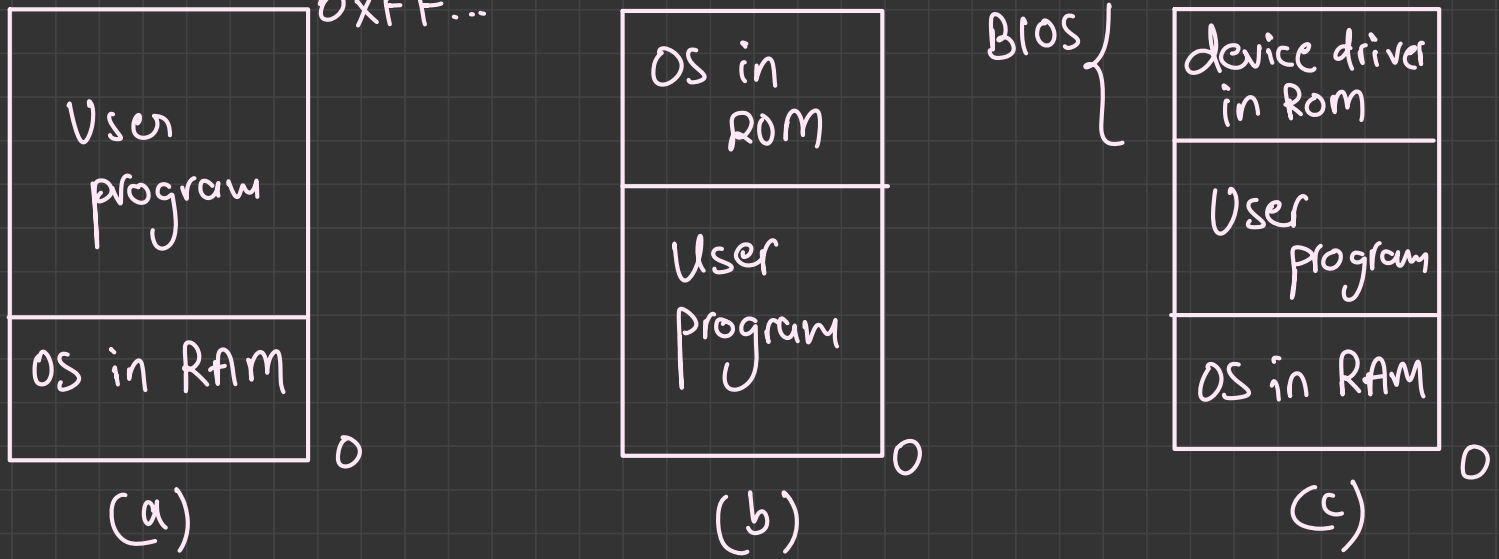
NOTE :->

* interview imp. topics :-> Paging, Swapping, fragmentation, virtual memory, caching algorithms

(b) Simple / Basic mem. management [ones who don't swap stuff.]

optional

(i) Monoprogramming without swapping or paging :->



(a) used in minicomputers (not anymore)

(b) used in embeded systems

(c) was used in PCs (MS-DOS) where one portion of program is placed in ROM known as BIOS.

BIOS: Basic input output system

(ii) Multiprogramming with fixed partitions :→

- Monoprogramming is only used in embedded systems nowadays.

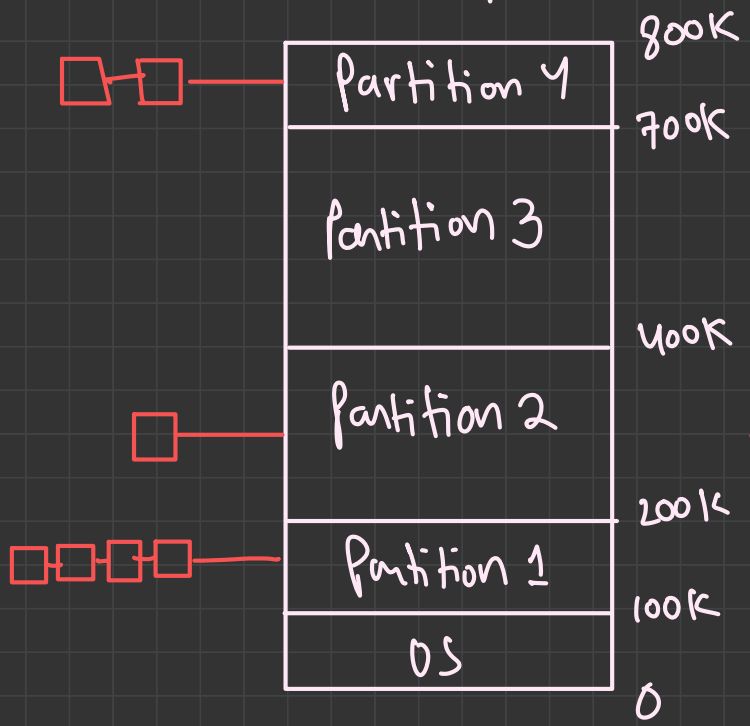
- Nowadays, every machine allows multiple programs to run
- One program might be waiting for I/O and hence another one can be scheduled to increase CPU utilization.

→ The easiest way to achieve multiprogramming is to have fixed n-partitions and might be unequal.

There are two schemes we can have :→

↓

a) Have different sized partitions with different queues to queue programs if partition already has a process running.



when a job comes, it would be given partition who has just enough mem. to occupy the program.

Cons :->

(i) if there are very less large programs then small - program's queue will fill

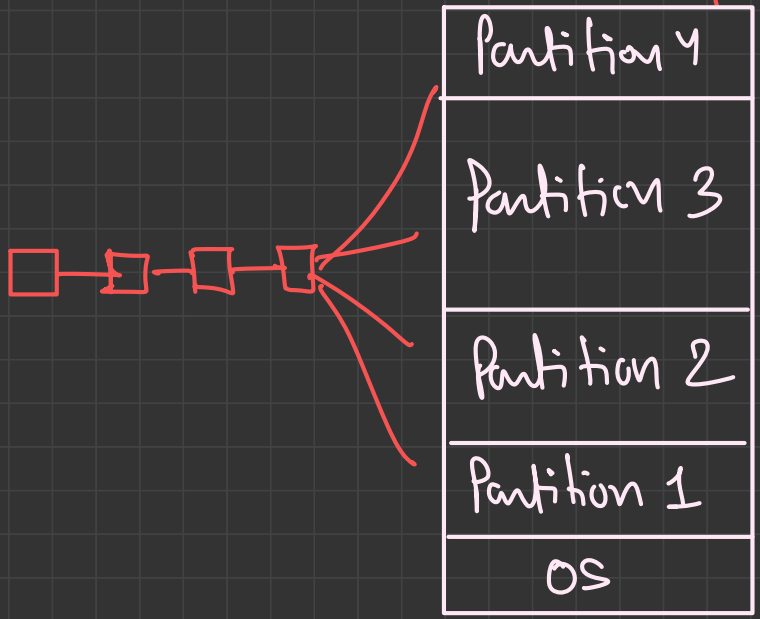
up and there would be a delay even a - large mem. (large partitions) is present.

→ biggest con for interactive systems

b) Second scheme could be have a single queue and search for largest task from queue to schedule as scheduling a small task randomly would waste resources

cons :- a) Here, also whatever scheme we use we will give priority to one kind of task more

b) for ex. picking largest and scheduling that to utilize resources would delay small programs and it is not acceptable for interactive sys. at all.



(iii) Relocation and protection :->

- using base and limit registers
- Job A is ready to run
  a) Memory partition starts at adds 100K
  b) Job A's size = 50K
  c) Base register set to 100K
  d) limit register set to 50K
- Now, let's say Job A has an instruction to LOAD data at adds 200

i) Hardware translation :→

a) actual mem. add. ⇒ 100K + 200

b) checks if this is in 500K range

c) prevents access beyond boundaries

- Base reg. enables dynamic relocation
- limit reg. prevents memory access beyond allocated partition

(iv) SWAPPING :→ fixed partitions and keeping processes in memory until they run is best approach to keep CPU busy.

gmp.

But when there is no memory, swapping processes to the disk and bringing it back when they have enough memory to run, can also be done.

Two approaches of mem. management

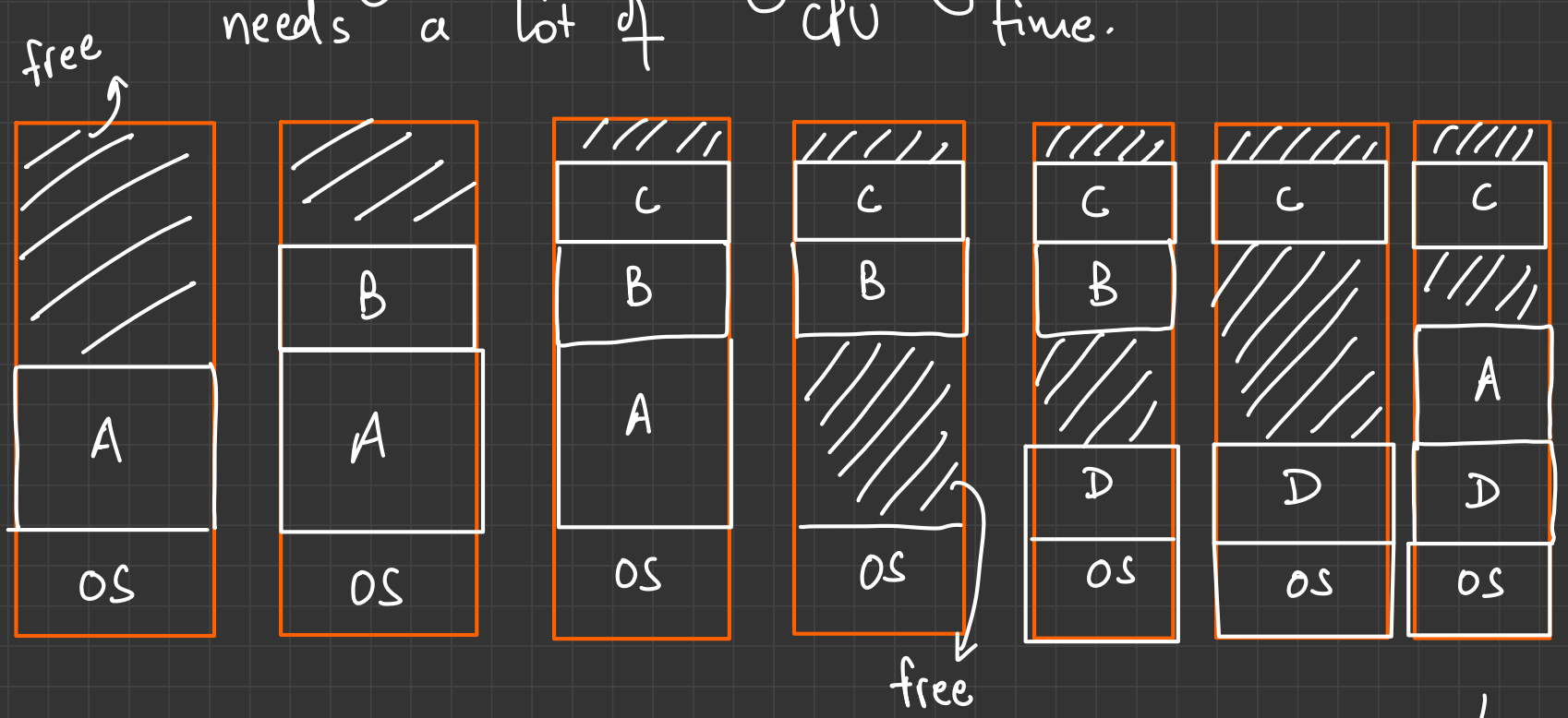
- Swapping
- virtual memory

* Swapping a process in when memory is available or swapping out once process completes itself or its time quanta, It increases the memory utilization

* Due to no fixed partitions and dynamic memory allocation it is very good acc. to memory utilization but allocation, deallocation is complex

* swapping processes might produce a lot of holes in memory and these holes might be combined to create a single unused memory chunk - called memory compaction.

* Memory compaction generally is not done as it needs a lot of CPU time.



* instructions in Process A needs address translation here.

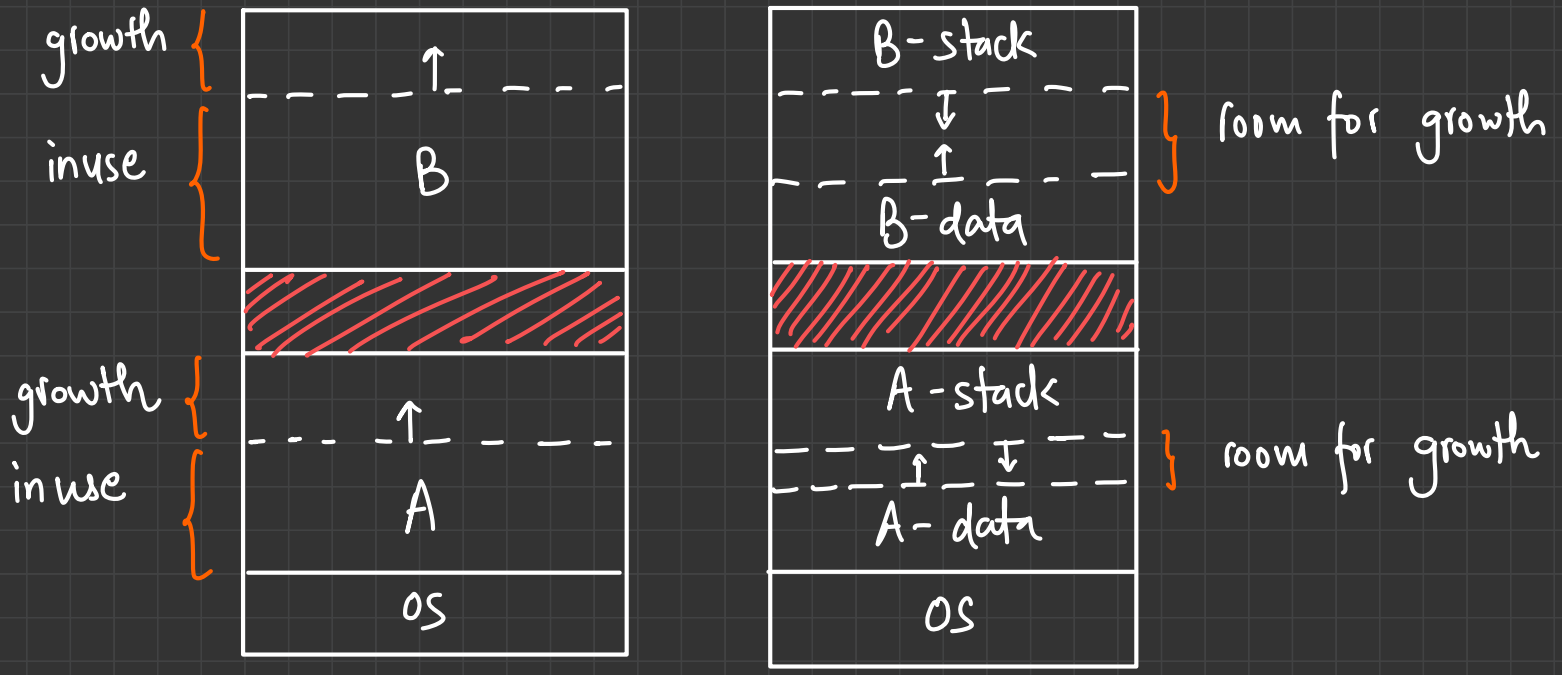
* if Process takes a fixed memory, then for the OS it is simple to allocate memory.
* But if Process grows in memory like most of the programs use dynamic memory where memory can grow dynamically because program uses memory from heap
* if program has hole adjacent to itself then it can use that memory to grow its heap into that hole.
* But if it is adjacent to any other program

than it can't grow into that process's memory.

* either OS has to allocate a hole for the program so that the program can grow in heap (OR) OS has to swap out some process back to disk to create a hole.

* So, it is always a case for OS that allocate some extra memory to the program to provide it a buffer to grow.

⊙ Diagram to show how two processes are allocated



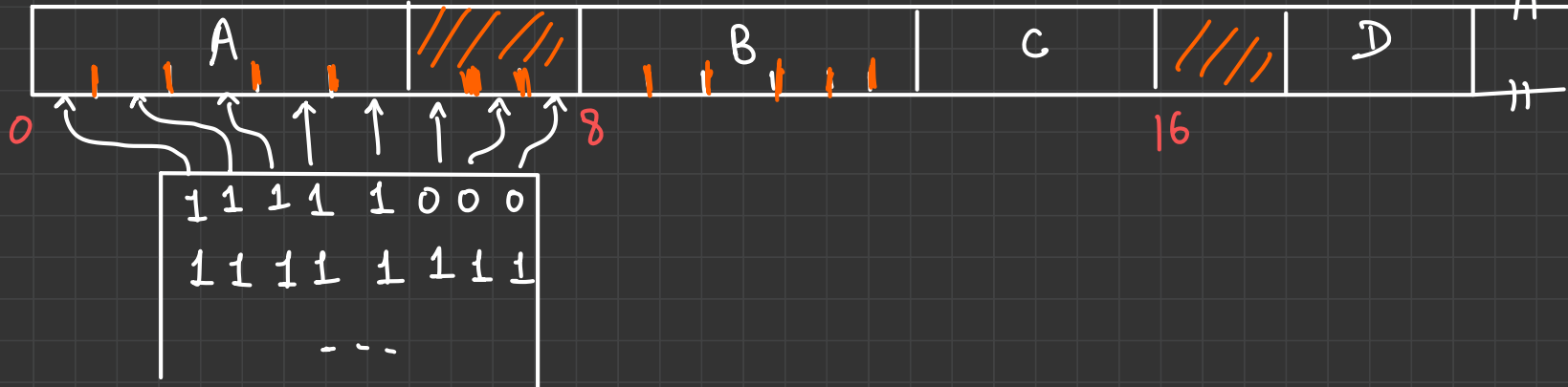
> two processes using buffer memory to grow / use dynamic memory.

(vi) (a) Memory mangement with bitmaps :->

optional

* OS has to manage memory and keep a track of free memory and it does this by using - bitmaps and free lists.

* with bitmap memory is divided up in allocation units which contains bits, 1 -> occupied 0 -> free



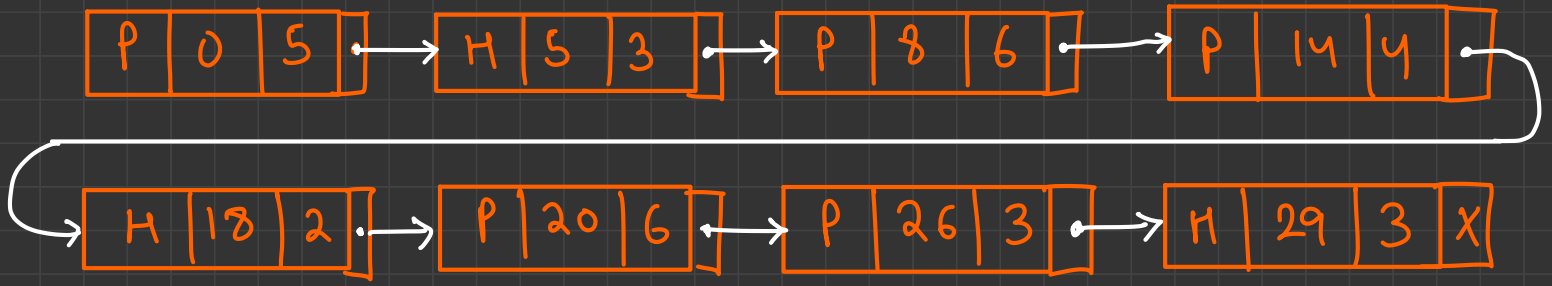
ex: if allocation unit is 4bytes let say and OS wants to bring 200KB process needs

$$= \frac{200 \times 10^3}{4} \Rightarrow 51200 \text{ allocation units}$$

→ So, 200 KB program needs 51200 of allocation units having 51200 zeros adjacent in a bitmap.

(b) Memory mangement with linked lists :→

↓



P → process H → hole ↳ sorted linked list according to the adds

* when process is swapped out, removing a process or creating a hole is straight forward
* replacing a process will require replacing a P with P or H.

* and then OS can use any allocation algorithm like best fit, first fit, next best fit etc.

#

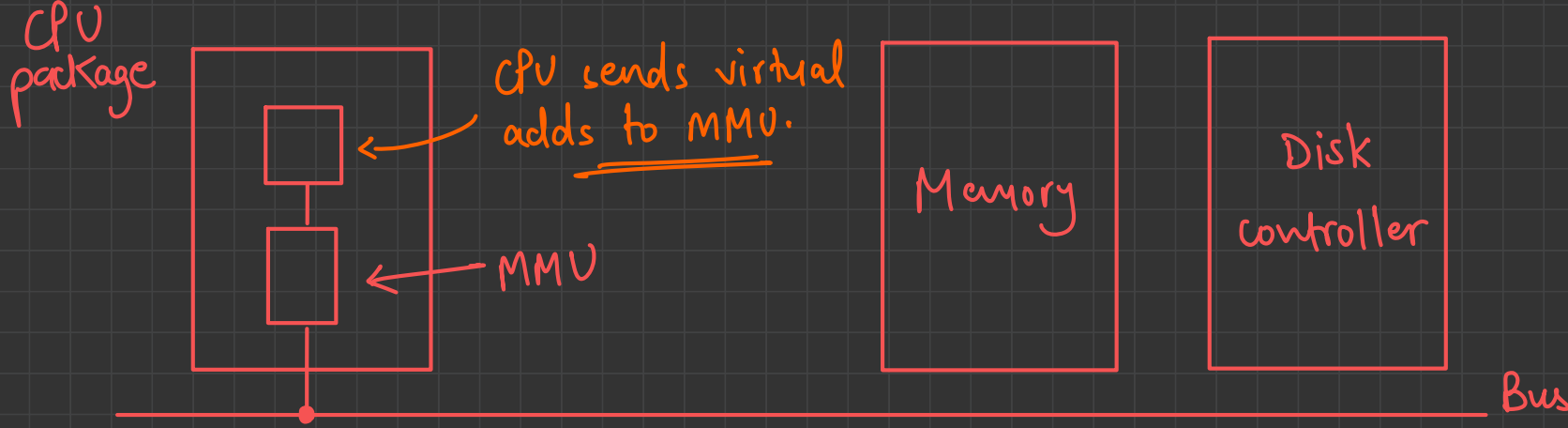
VIRTUAL MEMORY imp. :-> It is also a technique to swap processes from disk when memory required > physical memory available.

(a) Paging :-> Most virtual memory uses a technique called paging. To describe this let's take an example :->

↓

⊙ MOV REG 1000 :→ It does this to copy contents of memory address 1000 to REG or vice-versa. adds can be generated using indexing, base and segment registers.

→ These program generated adds are called virtual adds and form a virtual adds space.



* CPU generates virtual adds that goes to MMU and then the translated adds goes to the bus
* Lets take an example of our system having - 32KB of actual physical memory but let say that our system can generate upto 16 bits virtual adds i.e. $2^{16} \rightarrow$ 64KB. hence mapping would be :-

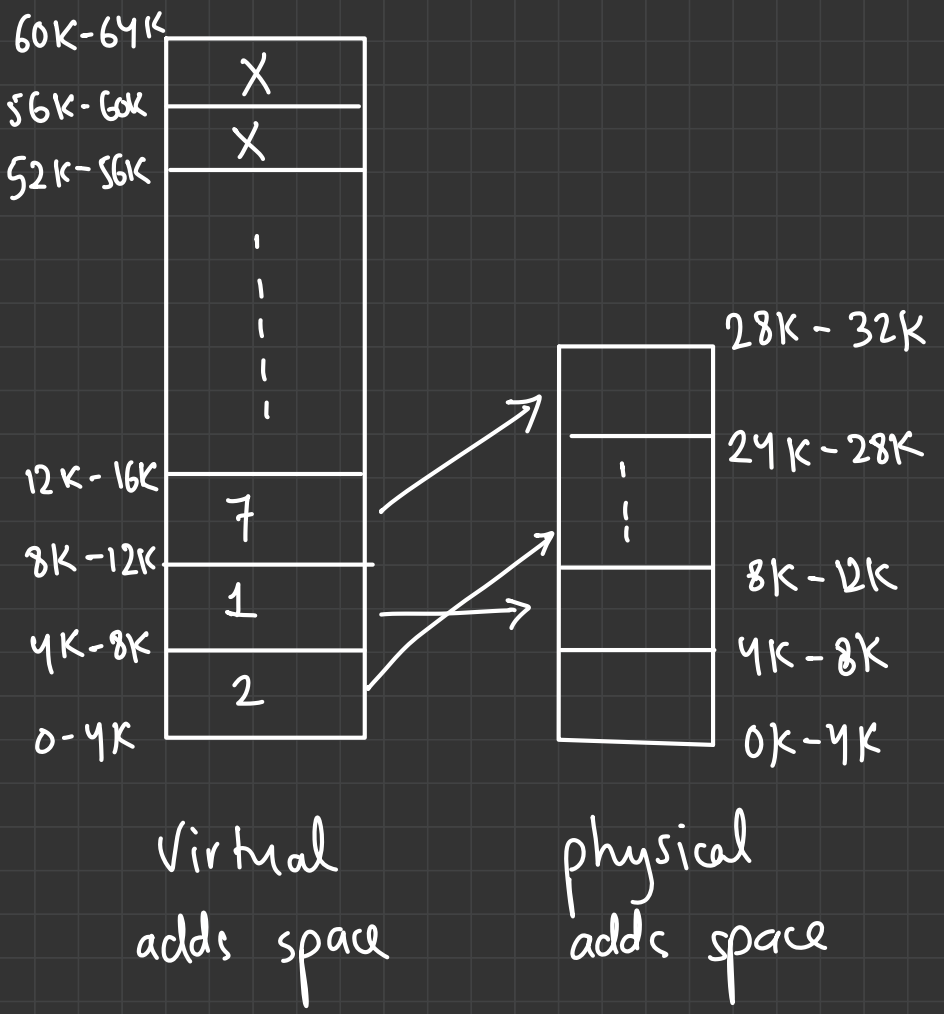
Virtual adds Space	$\longleftrightarrow$	Physical adds Space
64KB		32KB

* The virtual adds space is divided into somethiy called pages and corresponding pages in the - physical memory are called page frames

* The page and page frame are always equals in size. Virtual memory will have 16 pages and physical space will have 8 page frames in this case.

16 X 4KB (each page) = 64 KB

8X 4KB = 32KB



* Virtual adds 0 is sent to frame 2 - (8192 to 12287) as it is mapped to pf 2.

* MMU will map - (0-4095) virtual to (8192 - 12287) physical space.

* X means page not mapped
* 2 means page mapped to 2nd page frame (indexing from 0)
* when MMU gives a adds and sec page is unmapped that is called page fault.
* OS then picks up the least-used page referenced and writes its content back to the disk

4K
0 1024
0 1024 X 4
= 4096
→ 0 to 4095
0 based indexing

* Then It fetches the page just referenced and then it changes the mapping, references.
* While page fault a trap instruction is triggered and after taking the referenced page in - page frame Os again runs trap instruction.

↳ runs when invalid memory exception comes

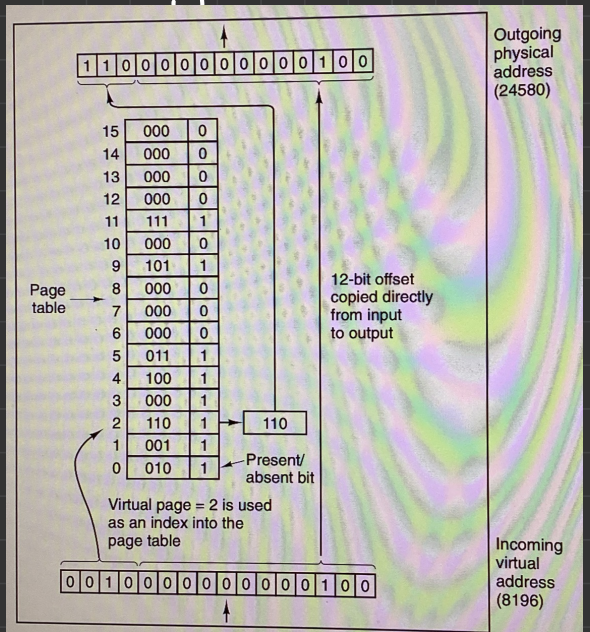
adds translation

(inside MMU)

16 bits (4 bits + 12 bits)

↓

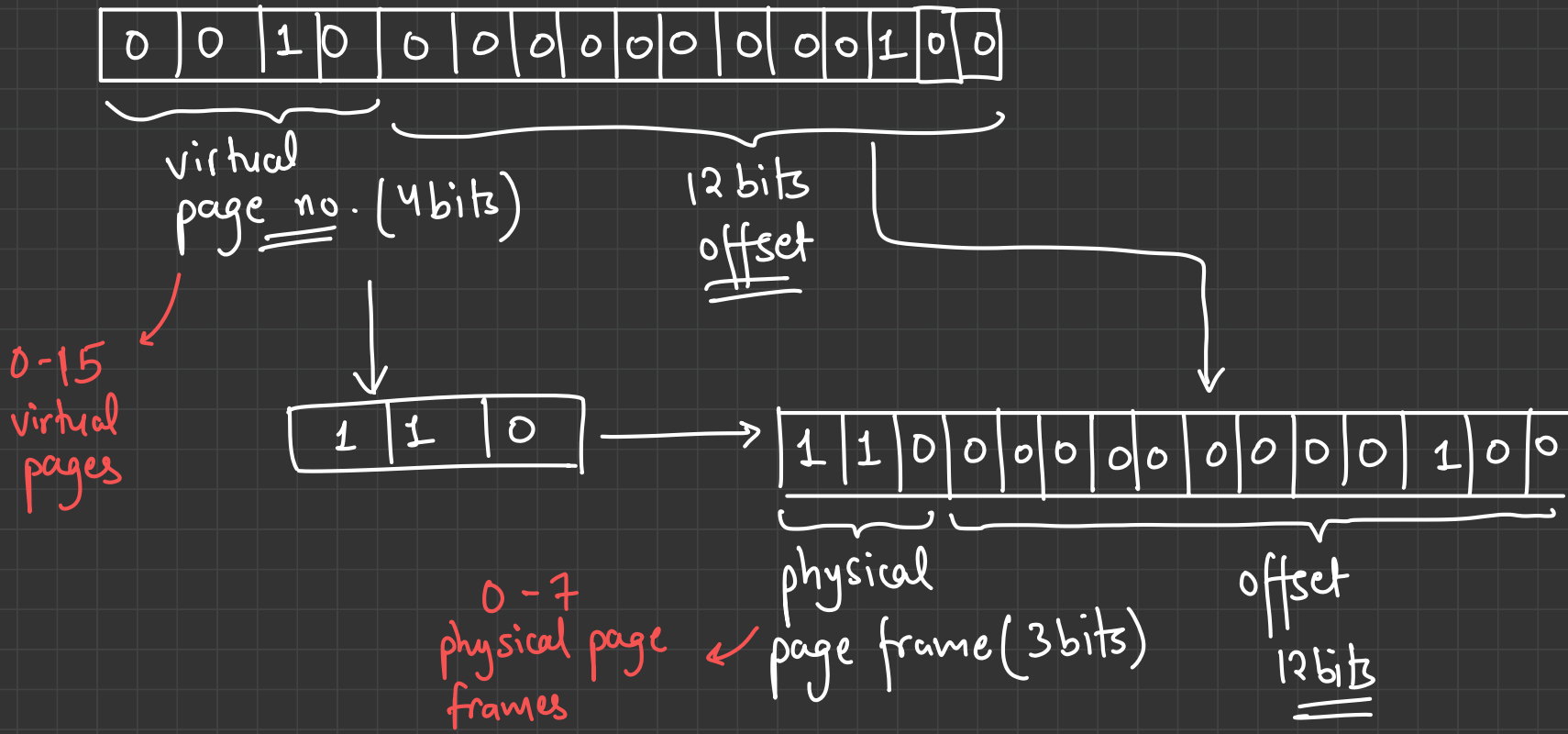
15 bits (3 bits + 12 bits)



↳ Page Tables :-> imp.

* The virtual adds is split into virtual page no. and offset. ex. 16 bits → 4 bits (page no.) + 12 bits (offset)
* 12 bits → 2^{12} → 4096 (4KB, considering lower unit in memory is byte here) [4KB each page]
* from lower 4 bits, page no. is found (0 to 15) and if virtual page is mapped with page frame let say page frame 3 (110) then these bits

are copied to higher order bits and physical adds is formed.



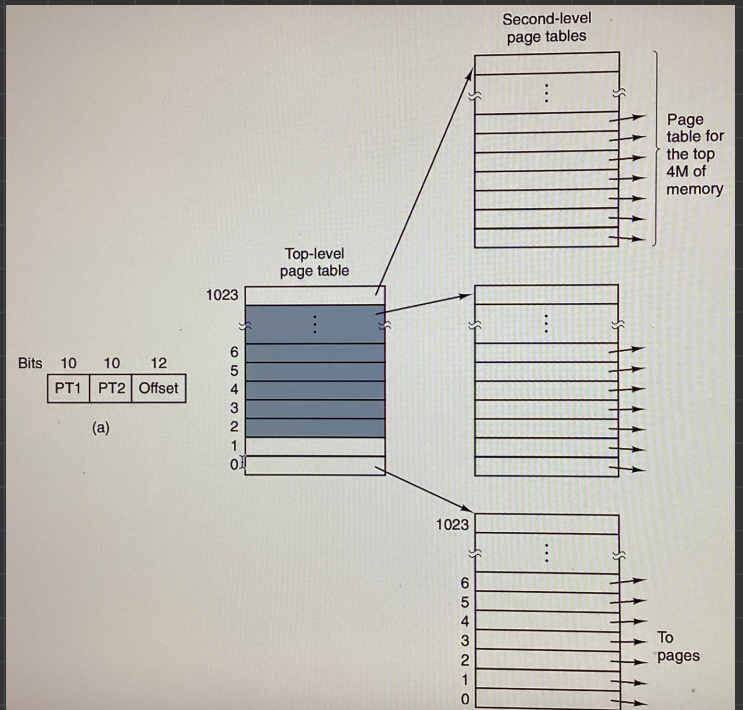
Multilevel Page Tables (optional) :->

* What if 32 bit virtual adds with 4KB page size (20bits + 12bits) -> It will have 1Million virtual pages
* storing 1M virtual pages page table in the memory is hard or not recommended.
* Here comes the multilevel page tables :->

(10bits) + (10bits) + (12bits)
PT1 PT2 offset

* The secret of Multilevel page tables is not to keep all page tables in the memory

* Lets see how MMU will access PTs here :⇒



* when a virtual addrs is pointed by MMU, It first extracts the index from PT1 (10bits)

* Each PT1 entry can have another 10bits indices hence here 3 PTs are pointed from 4M of each page entry in PT1

* Here, the other enteries are not used considering an example of program having 4MB as data segment, 4M as stack / text usage and 4M as a hole.

* ex. 32 bits virtual adds 0x00403004 which is 4,206,596 in decimal corrosponds to

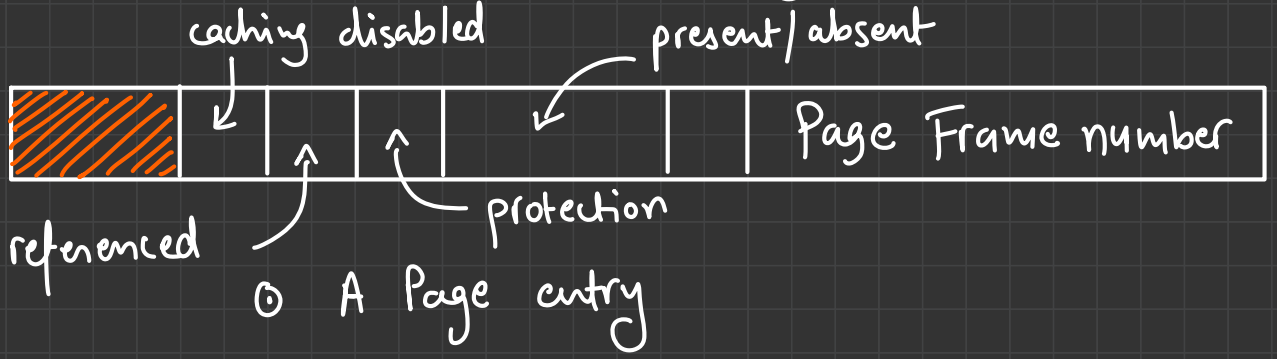
PT1 = 1 PT2 = 2 and offset = 4 and if in PT1 absent / present bit is 0 which will trigger page fault.

* Now, if page is mapped (absent / present bit = 1)

then as you can see only 4 PTs are needed although adds space have millions of pages

* Here, Top level PT, 0-4M page table, 4M-8M page table and 8M-12M page tables are enough

Structure of Page Table entry :->



0 A Page entry

* The present / absent bit corresponds to if page frame is available or not. if it is 0 then page - fault will occur.

* Protection bit tells what operations are permitted 0 means read / write 1 means only read

* Modified bit refers to if page was used or not. When the page is written, Hardware sets this bit. This bit is used when OS reclaims the page. if the page has been written or - modified i.e. dirty then it means it should be written back to the disk and if it is not

written/ modified which means it is clean then it is abandoned considering its copy would already be present in disk.

* Modified bit OR dirty bit can be used while page eviction becz pages which are clean can be evicted

* Caching bit allows / disables caching for a page. Pages which are mapped to device registers directly would turn off the caching when OS is waiting for an I/O operation to complete It will continue fetching word from the device to

make sure device responded, In this case - page will get refreshed and old data (caching) should not be used.

TLBs (translation lookaside buffers) :->

* In the absence of virtual memory, CPU will directly run/point to the instruction in the - memory.
* But with Page tables and all, additional and multiple references make CPU performance slow by 2/3rd.

↓

* The observation here before the solution is that in real world MMU/CPU refers a small no. of pages in most of the time, and rest are barely read or used. This is an example of LOR (Locality of reference). imp

* Solution here is to use a small hardware device that can translate virtual adds to Physical adds without going through the page table. This device is called TLB and it is present in MMU only, it consists a small no. of entries which are rarely more than 64.

Valid	Virtual Page	modified	Protection	
1	140	1	RW	31
1	130	0	R X	38
1	20	1	RW	62
1	19	0	R X	14
1	21	0	RW	75

Page Frame

* if entry is not - present when MMU is looking then it goes for PT lookup, evicts one entry from TLB and add that Page.

Software - TLB :->

→ * In most of the modern architectures like RISC, MISP, instead of keeping TLB in MMU it is kept in Software inhands of OS.

* when TLB lookup occurs and entry for a page is not present then TLB fault occurs and OS is responsible of fetching the required page and evicting one page from TLB and restart the instruction which made TLB fault

optional #

Homework :→ Read about inverted Page Tables & when they are needed OR used.

PAGE REPLACEMENT ALGORITHMS :→

- When page fault occurs, OS should try to evict - that page which is not heavily used
- There are some theoratical and practical - algorithms for page eviction. These algorithms are also applicable to other computer sciene thigs like cache.

* The optimal page replacement algorithm :->

→ It is similar to shortest job algorithm that we did in scheduling. It is possible in theory but can't implemented.

→ Each page can be labelled with a number of

txns that will run before the page will be referenced. So, Page with large number can be evicted as it will be referenced far ahead in the future.

→ This algorithm is not present in any real system. Lets study some practical examples / algos. that are being used.

---

* The Not Recently Used page replacement algorithm

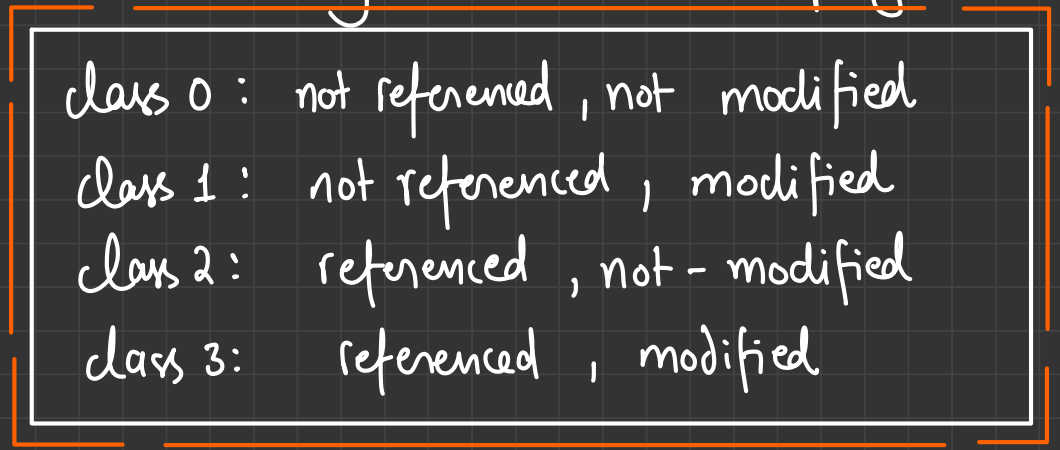
- As we have seen earlier that each page has bits like a R bit [whenever a page is referenced for Read/write] and a M bit [set to 1 when page is modified].
- These bits are present in every page entry
- These bits must be changed on every memory reference. Once a bit is set to 1, it stays 1 until the OS does change it so.
- If hardware doesn't have these bits then OS has to simulate these. So, whenever a page table is formed OS can mark all pages as 0 which

means they are not in memory. When page fault occurs then OS will set the R bit in its internal page table for that page and changes the page - table entry to point to the correct page with READ ONLY mode and restarts the instruction.

- If the page is subsquently written then another page fault will occur and OS will set the M - bit and restarts the instruction.

- On every clock interrupt R-bit of the pages that are not referenced recently can be cleared as compared to the pages which are referenced

- At any time when Page fault will occur we can have four categories in which pages can fall into :->

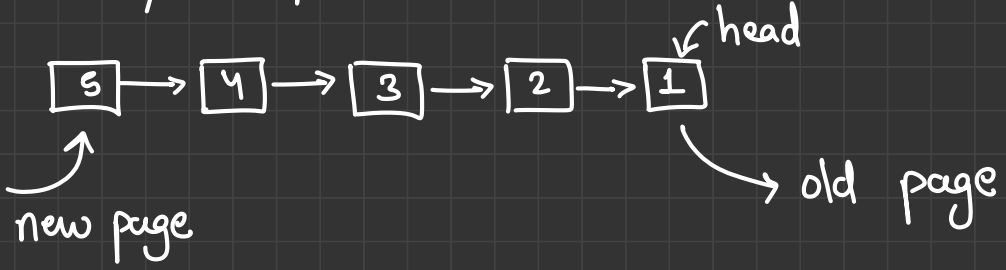


- class 1 can happen only when class 3 got its R-bit cleared by clock interrupt.
- clock interrupts donot clears the M-bit as it is needed to know if we need to write page back to

disk in future or not.

- The **NRU** (not recently used) can be used here now to remove the page which is modified but not referenced in atleast one clock cycle (20ms let say)

=

* FIFO (first in first out) Page replacement algorithm
  - Fairly simple and less overhead algorithm
  - 

- A new page can be pushed to tail and on page fault we can remove page from front which will be the oldest from others.
- Problem here is if 1 and 2 pages are old but in heavy use compared to 3, 4, 5 etc then it will still evict 1, 2 and will produce more page faults in future.

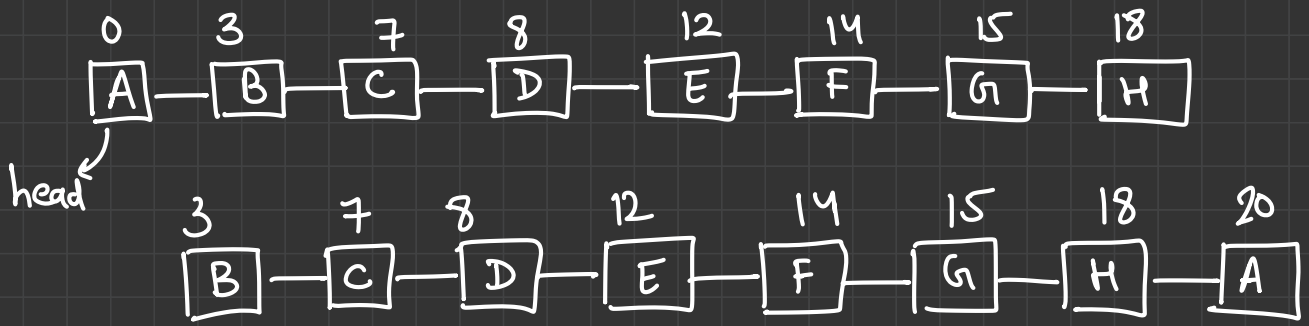
* The Second Chance Page Replacement Algorithm

- A simple modification to FIFO can be done which is throwing the head page without looking at it

can be altered.

- On eviction, It can inspect R-bit which if 0 then It means page is old and not referenced and can be evicted immediately.
- if R-bit is 1 then It is cleared and put at the end of the queue (second chance).

ex:



a) if R-bit of A is 0, then it can be evicted and written back to the disk (if M bit is set)

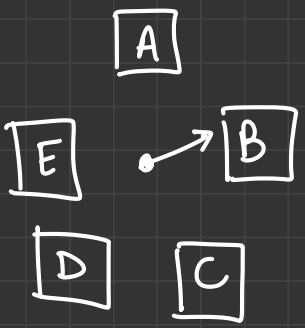
b) if R-bit of A is 1, then it will be pushed at the end of the list with time as the new time of arrival. and R-bit of A will be cleared before push.

* One by one this algo. applies this thing on every page and once it will reach A with R-bit not set and it will evict page-A.

Hence, this algo. always terminates.

=

* The clock Page Replacement algorithm :->



- on Page fault, inspect page at the clock hand, check R-bit
  - R:0 (evict the page)
  - R:1 (clear the bit, advance hand)
- It differs only in implementation from second chance algorithm not by concept.

* The LRU (least recently used) Page replacement algo.

- This algorithm is based on the observation that the page which is heavy in use will probably be used again in future.
- and page which is unused from a long time will remain unused in future too.
- Hence, on page fault, evict the page which has not been used for a long time.
- But LRU is not cheap, this algorithm required to maintain a linked list to track the pages lifecycle where we have to maintain the least recently used

page in the rear and most recently used page at front.

- The problem is on every memory reference, we have to modify the list. removing/deleting a page, adding a new page and moving to front is an expensive operation.
- But there are other ways to implement LRU with a special hardware :→

(i) maintain a 64bit counter "C" for each page in the page table and when a page will be –

referenced increment the counter & while any page fault it can scan the page table and remove the one with Counter value minimum

(iii) for n-pages, machine can make (n x n) matrix in its hardware setting all value to zeros initially. and when K^{th} page will be referenced then K^{th} row's all bits will be set to 1 and K'^{h} column will be set to 0.

At page fault, the row with lowest binary value will be evicted.