Memory Management Basics

stack, heap exists only at run time of program.

- Desired from a multi-tasking system
  - → Multiple process in RAM at same time (multi-programming)
  - → process should not be able to see/make change in each other's code
  - → Advanced requirements.
    - process can be put anywhere in RAM (pos should not be fixed)
    - → (gives flexibility i.e process can exists anywhere in RAM)
    - → Process need not be continuous in RAM
    - → Parts of process could be moved anywhere in RAM.

Different Times.

- → compile time
- → Load time
- → Run time

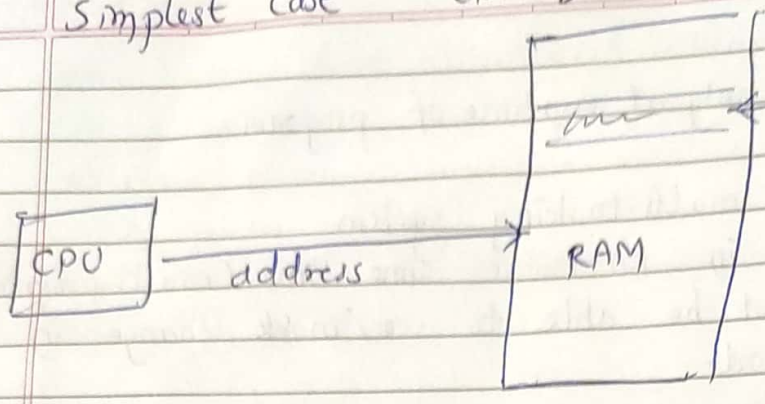
} Different action at different time depending on different os & processor. depending on that this leads to diff types of address binding

Diff types of Add binding
(determing address)

Run time add binding.

- Address of code/variables is fixed at the time of executing the code
- It may be decided at load time but it can be change even program is running.

Simplest Case ex: DOS



Example:

address	instruction
1000	mov $0x300, r1
1004	add r1, -3
1008	jnz 1000

(1000 add CPU send karel)

start la 1000 add issued ~~karel~~ hoil mg tyache contents fetch honar ani te execute honar
mg 1004 add Bsued ——

here $0x300 address is used while executing instruction

How does this impact compiler & os ?

→ Eth jevha a process run hot aste tevha address Bsued je hotat te Ram la directly miltat.

→ So, exact add of globals; add in "jmp" and "call" must be part the machine instructions generated by compiler

→ how compiler know addresses at compile time?
It assumes some fixed addresses for globals, code.

Os loads the program exactly at the same add specified in the executable file.
Non-relocatable code

Now program can execute properly.

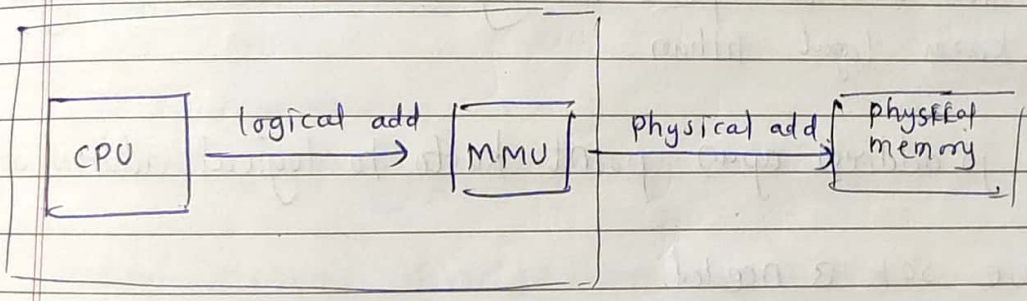
2 reg for mem add

As here compiler assumes addresses, so ~~they~~ it ~~may~~ will generate overlapping.
so support only one program (i.e, only one program in memory at a time)

Base / Relocation & Limit scheme

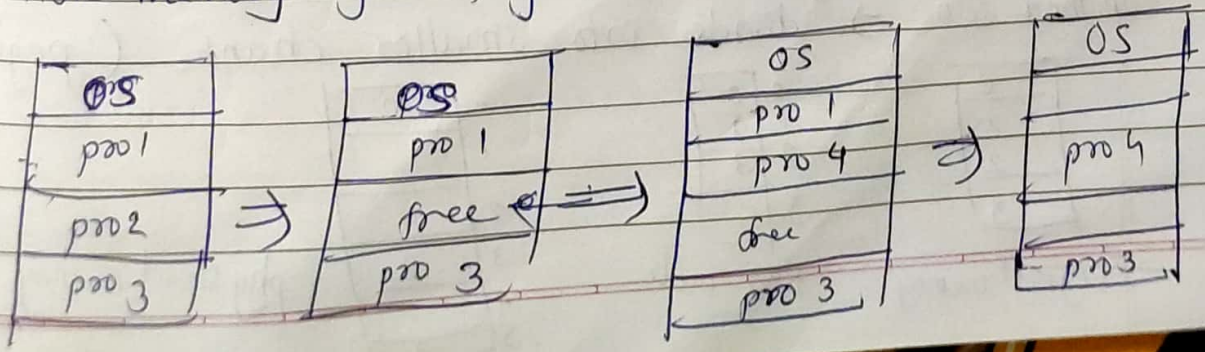
2 reg for memory add
- limit register (~~Base register~~)
- relocation register (Base register)

1st value check against limit reg.
If process is in RAM
→ location of process → relocation register (Base)
→ Total size of process → limit reg.



you could access code of your process & can make change in it.
Only one check → limit

Here memory get fragmented. | Variable partition scheme



Segmentation : Multiple base + limit pairs

Eth add Bsue chya veles kathal base & limit reg abe yach pn reference yeto.

text sathi vegal segment, data sathi regal, ——
means text regali pair, data sathi regali, heap, stack

No. of reg increased. → cost ↑

——

Multiple base + limit pairs, with further indirection

seg reg used as indices here.

Base + limit pairs in memory, one reg to CPU to point table.

Thod slow hot as pairs memory madhe astat ani tyamna access karav lagat ththan.

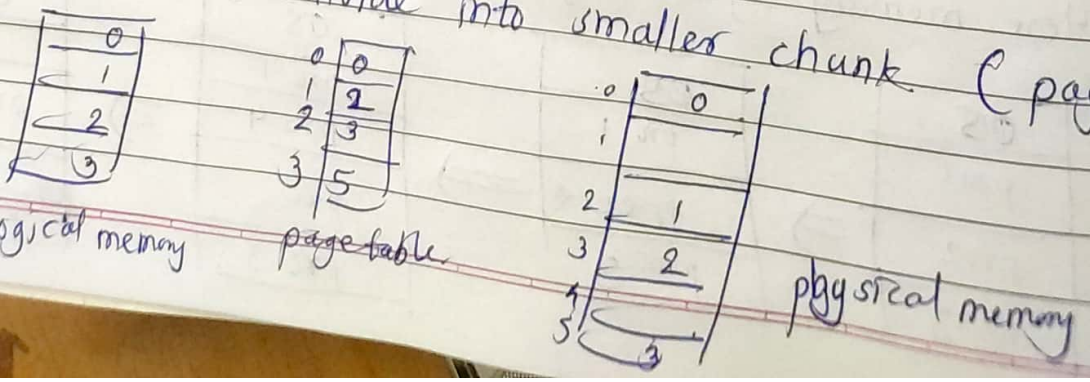
→ & ne jo address apan point karto to logical address asto

——

Suppose 50K B needed
and available 30K, 30K, 40K, they can't allocate (needed continuous 50K)
— External fragmentation.

soln
↳ compaction - chunk move karayach.

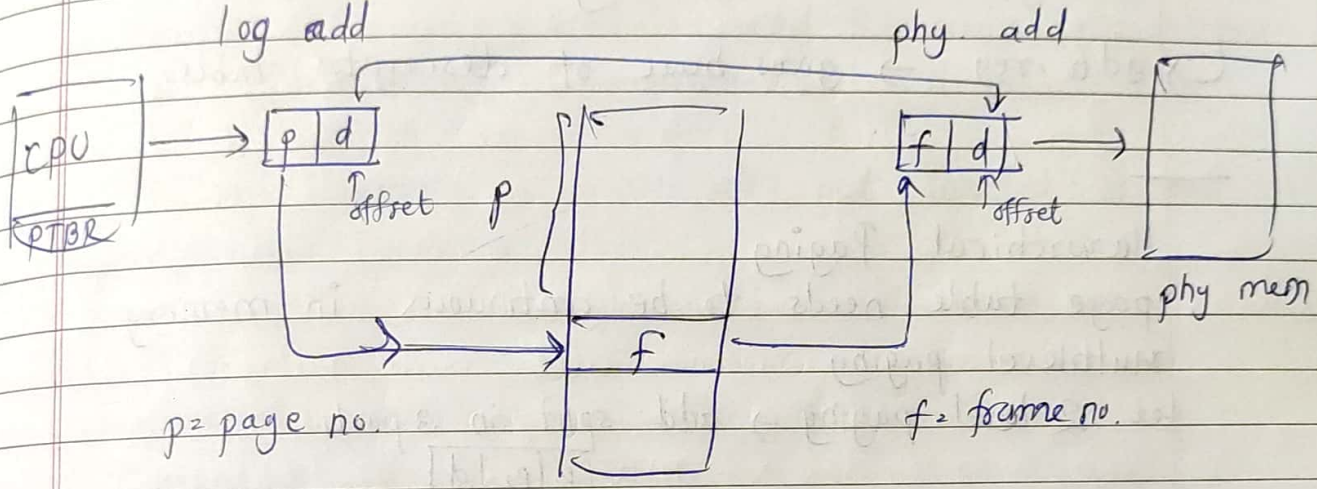
another soln → divide into smaller chunk (paging)



one page is continuous (further not divided into smaller chunks)

page table located in memory set up by OS

CPU has PTBR (page table base reg) (gives location of page table to CPU)



Process ^ equally ^ composed of size ~~divided~~ pages.

Actual memory divided into page "frames"

size of frame = size of page

SS, ES → they are used as indeces in segmentation table.

logical add = seg. reg. + add. compiler provides.

In Intel 32 support both segmentation & paging.

Process gives feature to use only one at time

xv6 bootloader

Load boot sector in RAM at 0x7c00
This add: documented by xv6 & qemu

.code 16 → (Directive to assembler)
(saying code for 16 bit)

descriptor table → array of base, limit

↳ gdtr reg → gives base of descriptor table

Hierarchical Paging
page table needs to be continuous in memory
Multilevel paging
let 2 level paging → add split in 3 parts
[p1 | p2 | d]

x86 paging

page directry | page table | offset
2 level → 4 kB
1 level → 4 MB

---

page directry / page table → in memory

PS (Page size) 0 = 4kB, 1 = 4MB

CR3 (Reg) → gives the base page directry table

Segmentation + Paging

Kernel role → creating GDT/LDT, making gdtr points to GDT.
setting up page directory, page table, making CR3 points to base of page directory.

Ones it is done, everything will be manage by hardware kernel doesn't need to come into picture again in memory manegment.

Here all memory violation also get detected in the segmentation + paging unit.

Segmentation + Paging setup of xv6
Base = 0, limit = 4GB
So, logical add = linear add (always)
(known as identical mapping)

Limit is 4GB so set to maximum possible length of RAM
32-bit prosessor → 2^{32} → 4GB
Bcz did not setup any segmentation, didn't divide in any chunk.
So segmentation is practically off.

Does segmentation setup to do identical mapping.

segmentation table is setup using global descriptor table (GDT) entry.

GDT entry → 64 bit

GDT → Global
LDT → Local
} in xv6 only GDT

```
gdtdesc:
  .word (gdtdesc - gdt - 1)   # 2byte
  .long gdt                   # 4byte   } total 6byte
                              # add of gdt
```

(Macro) SEG_ASM (we to do bit-wise operation & create GDT table entry)

SEG_ASM (STA_X | STA_R , 0x0 , 0xffffffff)

- STA_X | STA_R → permission
- 0x0 → base = 20
- 0xffffffff → limit = 4GB

with this type of setup, (base=0, limit = 4GB) every address in kernel will have identical mapping (remain as it is)

CR0 → register
last bit PE = 1 then protection mode enabled.

ljmp $(SEG_KCODE<<3), $start32

- $(SEG_KCODE<<3) → in code seg
- $start32 → EIP

from now onwards run in 32 bit mode.
previously running in 16 bit

left shift by 3 → Bcz there are 3 bits in every seg selector at the right side & these 3 bits are not part of seg. selector.
These 3 bits → TI (2) Table index
RPL (1 0) Request privilege

go in 32 bit
Here Base will be added to address translation going thru GDT table, But still it will be identical mapping

movl $start, %esp // start → 0x7C00
∴ esp = 0x7C00
stack goes downwards, so we will use stack from 7C00 to 0.

C code
read kernel code from disk → put in RAM → do paging setup → run kernel

readseg → read from disk into RAM.

func bootmain → job B to load kernel from disk in memory
xv6.img → concatenation of bootblock kernel

ELF
uint entry; → 1st instruction to be executed in ELF

ph off → program header offset (gives add)
ph num → ph number (no. of entries)

we will load kernel at this (vir. add)

vaddr → virtual add → 0x80100000
paddr → phy → 0x00100000

we can't like this any particular other ELF file but kernel can demand, that he wants to occupy particular region.

EXTMEM = 0x100000
KERNBASE = 0x80000000 (First kernel virtual add)
KERNLINK = (KERNBAS + EXTMEM) → add where kernel is linked

---

elf = (struct elfhdr *) 0x100000;
here we can write this add (points) because already set MMU properly (0 to 4GB)

SECTSIZE = 512 (normally) (Default)

readsect → read single sector at offset into destination

ELF_MAGIC → 0x464C457FU // \x7FELF in little endian

stosb → feels memory location with particular value

Eth ph→memsz > ph→filesz
my stosb la call karun 0 fill karnar

entry = (void (*)(void)) (elf→entry);
↓
type casting particular value in function ptr

entry.S → code from kernel (upto the paging is off)
4KB → 2 level
4MB → 1 level

Setup Page table
Only two entries 0 & 512 → both will point to frame 0.
4MB X 512 = 2GB

→ dir | offset | →

NPDENTRIES 1024
NPT — " — 1024
pde_t — 4 Bytes

PDXSIFT = 22

Flags set.
PTE_P → 0001 , PTE_W → 002 , PTE_U → 004
↑ present ↑ writeable ↑ User
PTE_PS → 80 (7th bit) page size 4MB)

CR4_PSE 0x0000 0010 // → enable page size extension which is 4MB

---

entrypgdir → part of kernel code
kernel code compiled linked by linker
so the add all generated in ELF file are going to be all virtual add.

---

kernel.ld → directive to link the kernel

Virtual to phy

V2P_WO (x) ((x) - KERNBASE)
P2V_WO (x) (c +)

Todo flags for enabling paging

CR0_PG 0x80000000 // paging
CR0_WP 0x00010000 // write protection to enable

KSTACKSIZE → 4096 // size pre-process kernel stack

.comm stack, KSTACKSIZE // directive to assembler
allocate data of size KSTACKSIZE

kvmalloc → reinitialize page table (over
seginit → segment descriptor (over) kern

jmp *%eax → writing like bez wants jmp to particular
fix phy add.

Process

Process related data structure in kernel code,
→ different for diff kernel

PCB (Process Control Block) (common in all kernel)
1 PCB for each process

PCB identifier for process in kernel data structure

struct task_struct → in linux kernel
struct proc → in xv6

Fields also in PCB vary for diff kernel.
Process state → running, sleeping, terminated etc.

List of open files in PCB
fd[] → array of ptrs.

fd (file descriptor) → index in the array of ptrs maintained in PCB.

(kernel in linux) in array 1st 3 are already pointing to files which are opened by kernel.

0 → std i/p
1 → std o/p
2 → std error

So 1st file db we get 3

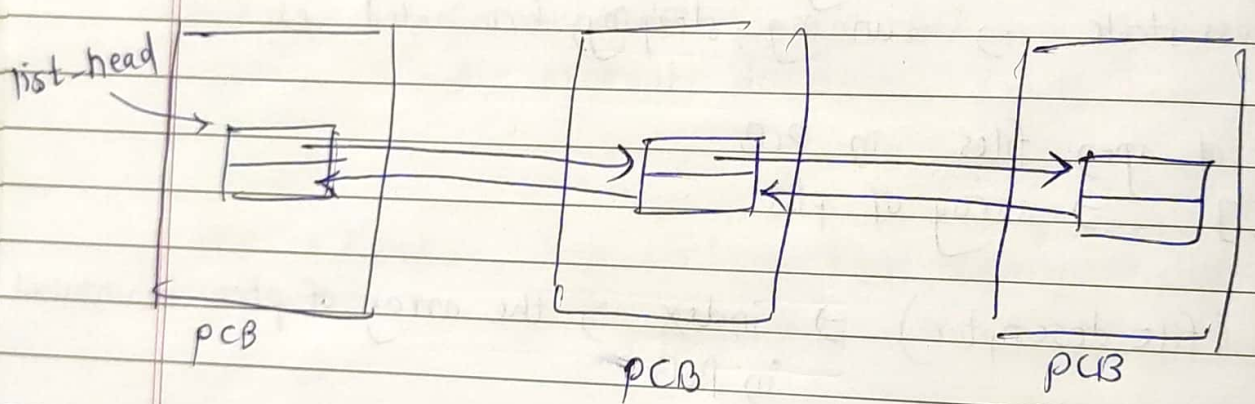
All these data structure in kernel space

all process in xv6 is ptable (global variable) maintaining array of proc[NPROC];
ptable.proc[]

Process Queue/Lists inside OS

In xv6 only array so shedular has to do more work look for all process and decide which one to shedule

In Linux



like doubly linked list (linking internal member)

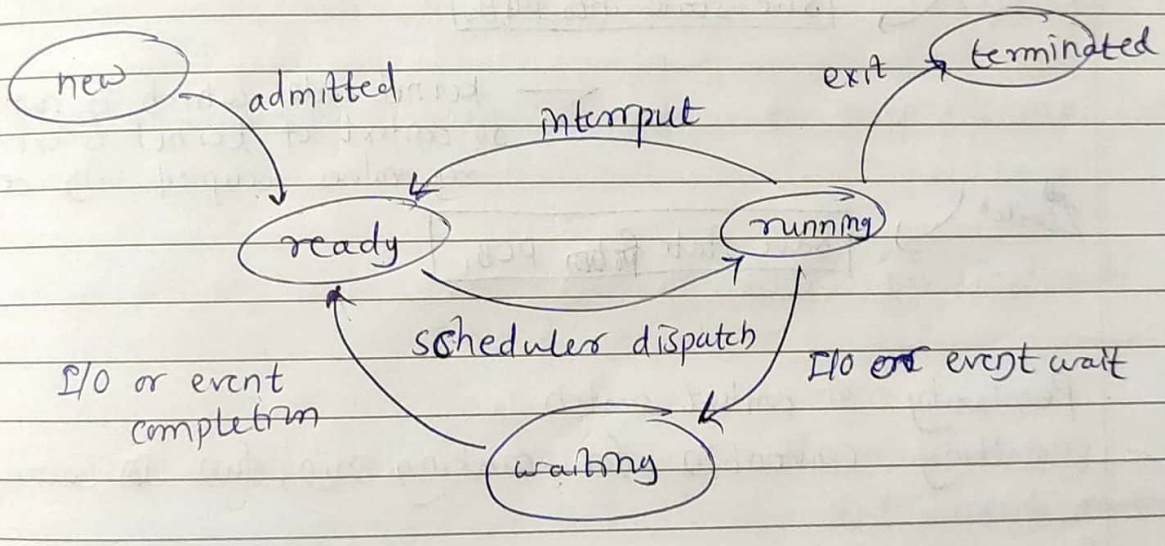
Process saru astanna I/O request ali (like scanf) tr mahit nahi ki keyboard var key kadhi press karar mhanun tyala queue madhe takun kahi mean nahi karat.
So, tyala tithun cut karun wait queue la taktat ani complete jhal ki ethun cut karun punha queue madhe

Steps diagram of process state

only PCB is completly constructed by the kernel PCB is moved to ready queue & state also marked as ready (means can be picked up scheduler for scheduling)

If now scheduler selected, change state to running then pass the control to process
It has to be execute on particular processor

For termination it must call exit
~~after it will scheduler and will new process~~



ctrl + C → will generate signal
when signal occurs kernel will notify process & process will its own signal handler and if not kernel will run its own default sig handler.

Here kernel particular code will run, looks for current process & terminate it.
(Eth process exit ne terminate nahi hot to kernel code ne hote)

Ata illegal memory access ka seg fault yevun termin
hot tevha, hardware interrput hot mg te run hotta
check karel ki kuthali process ne illegal memory
access keli ani tyala terminate karel.

— "Giving up" CPU by a process or blocking

sys_read
file f = current → fdarray [fd];
PCB

Context switch

1st switch → save state into PCB$_0$

← kernel code which is running
so context of kernel is executing
∴ reg. values occupied with kernel

2nd switch → save state from PCB$_1$

Peculanty of context switch
calling convention for making run, fun in same process
Cntext switch one process take place of another
→ Eth caller sangel mala he he reg use karaayachet
remaining callee use karel.
Pn contex switch madhe tas nast purn use hotat
by another process

So if code of context switch madhe lihala tr te calling convention ne hoil su,

tyacha code assembly madhe asto.

Process in xv6

Max possible virtual add → KERNBASE (0x8000 0000)
2GB

2GB + 1MB → onwards is kernal code/ text, data

PHYSTOP → 20 to 24 MB

Kernel needs to create mapping to map kernel code, data, process date

Kernel loaded at 0x100000 (1MB) physical add.
0 to 0x100000 → BIOS & devices

Kernel is loaded at lower regin of phy memory as some systems may not have that much memory

---

A process → 2 stacks
- → user (appln) stack
- → kernel stack

per process 2 stack

3rd stack
→ ~~used by kernel~~ Kernel stack used by scheduler

THB → is not for per process (each process) Generic stack

struct proc
uint sz; // highest vir. mem add ~~available~~ of Part code

should be, sz < 2GB

guard page → 4K, stack → 4K

The whole setup has to be done as a part of fork()
pgdir → pointing to page dir

Handling Traps

Kernel needs to get all data related to event → that info has to be made available to kernel by hardware
Some kind of data passing needs to be done betn kernel & processor.

Current privilage is → in cs (in 2 bits) in the field CPL

If your in protected mode
cs will not be multiplied by 16 (left shift by 4)
IDTR & IDT
Entries in IDT called Gate (Gate is code seg eip pair)
↑ (in memory)

IDTR register gives IDT base add & IDT limit

uint var : 2;
var is defined of 2 bytes

value for cs offset descriptor privilege level

SETGATE (gate, Btrap, sel, off, d)

SETGATE (idt [i], 0, SEG_KCODE <<3, vectors [i], 0);

(particular entry we trying to set) add of idt

Btrap

SEG_KCODE = 1 <<3

actual add of fun to be executed

vectors [] → in vectors.S with 256 entries pointers.

DPL_USER → 3 (ring)

in xv6 same for all

```
.global vector64
vector64:
  pushl $0          // push 0
  pushl $64         // push no.
  jmp alltraps      // jump to alltraps
```

CPL → % cs & <= DPL

levels set in descriptor

all levels set to 0, except sys calls level 3

checking that you should always go on low level not on higher.

On int instruction / interrupt CPU does this

Save %esp & %ss only if PL (privillage level) < CPL

processor gets kernel stack using task segment descriptor. (extra reg stores, ss esp for kernel)

IF → interrupt enable flag

set %cs & %eip values to the descriptor changes to refer table (set to value from vector table)

sys call has to return error no to the calling proces

struct trapframe → struct of reg pushed in stack in reverse order.

so we can access content of stack by pointing tf. and we access all data by structure trapfram
tf → pointing to relavent location in kernel stack

SEG_KDATA<<3

Trapframe is a part of kernel stack
It can access return value, trap no, old value of reg, ss, esp, everything

intrapframe in struct trapframe {

```
uint esp;
ushort ss;      // 2 bytes
ushort padding6; // 2 bytes
}
```

These 2 bytes additionaly added to insure that the content of struct actually match the size of trapfram built with assembly code

actual size is 2byte
2 bytes was pushed as 4 bytes, so access it as 4 bytes so padding.

Bootloader call main
main → call tvinit → set up all the gates in idt array also set particular value of syscall (says it should jump from level 3 to 0)
→ goto vectors.S
Do similar job
push 0
push current value of interrput
jmp to all trapes

In all traps push more things (reg)
→ call trap
Now we have trapframe already on stack.

In trap fun
if sys call then
(myproc() → tf = tf ; // ~~particulary~~ currently running process
calls syscall() ; //

---

who sets trapno → It is on trapframe. It was push on trapframe by vectors.s (as each level pushing it's own number)

---

struct trapframe {} in c file was defination of struct, while in assembly code we created a instance of that struct.

INT make processor look up in IDT, jump to location given by IDT

pushing ss, esp (in initialize phase of trapframe) → B option (optional remember)
If switch from 3 to 0 then push, then only iret it will pop

Interrupt only can occur at the end of instruction not in between

For all harware interr - istrap = 0
soft Btrap = 1

on 0 clears FL_IF (Interr flag)

kernel use ~~only~~ general purpose register

code making system call like read, open, write
code of read, — in usys.S

make SYS_read, SYS_open, SYS_write
system call number copied in %eax
then call int

jump on vectors
push 0
push 64
jump all traps

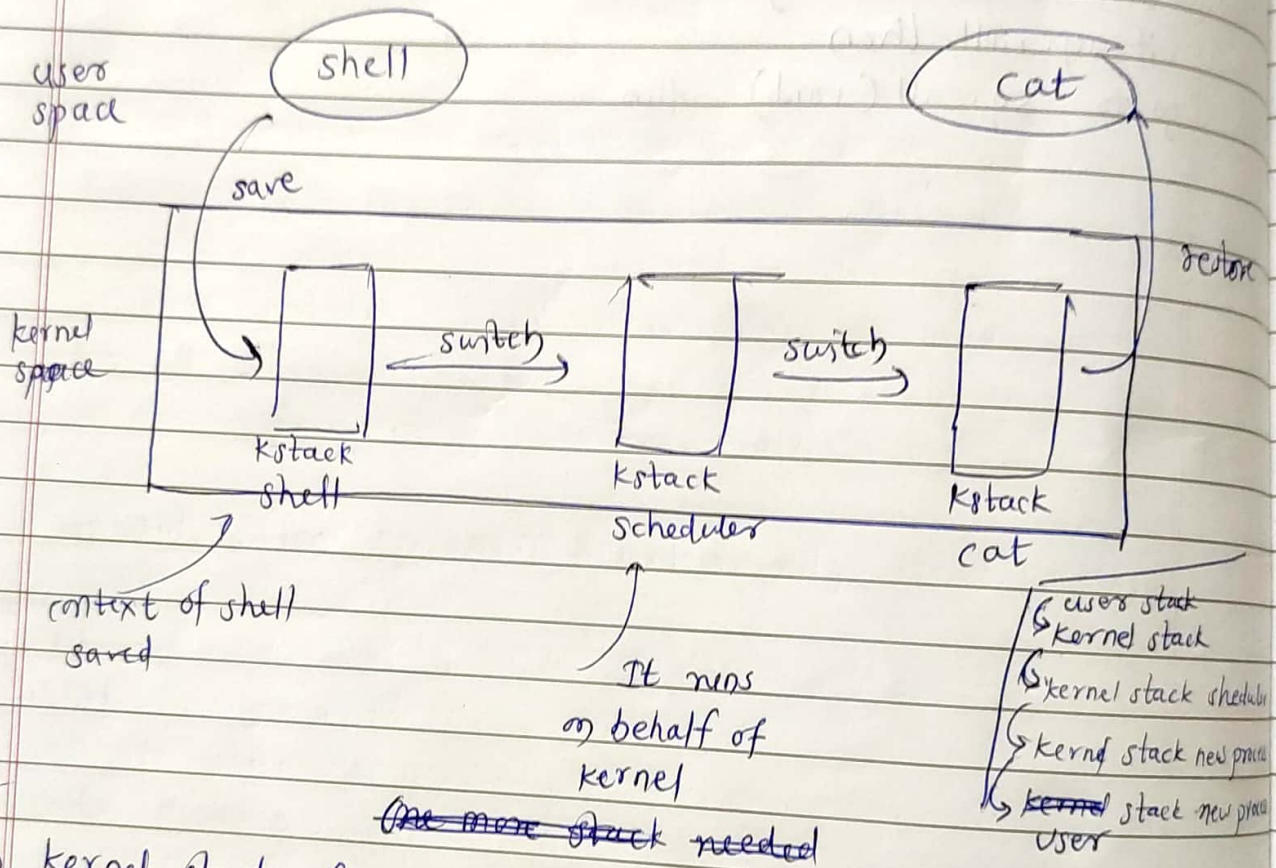
→ alltraps
pushing reg
pushal (pushall)

then call trap
then check for sys_call
if sys_call then
go to syscall (void) fun

xv6 Sheduler

Transition from process P1 executing on CPU to process P2 executing on CPU, in betn kernel code i.e, sheduler will execute.

4 stacks need to change!



↳ kernel stack of process, it change to stack of sheduler then only sheduler can run.

↳ Again stack needs to change to kernel stack of cat (process)

---

mpinit() → detect other processes
mpmain() → start execution

mpmain calls scheduler()

mpmain will run on all processor → ~~with~~ sheduler will run on all processor

kernel code runs on every CPU independently and sheduler process independently

only data structure which represents collection of process → struct {
// array of struct ~~of~~ proc
} ptable;

sheduler()

sti() → set Interrupt flag (Enable interrupts)
asm → directive to write assembly code in C

Iterate thr' proc array
check if p→state is RUNNABLE (ready) ~~if not then~~

c→proc = p; // c pointer to cpu
// remembering process is related to cpu

struct context {
edi;
esi;
ebx;
ebp;
eip;
}

// Per-CPU state
struct cpu {
...
struct segdesc gdt[NSEGS];
}

another/~~2nd~~ gdt table per-CPU

P→state = RUNNING; // only change state

loading → reading from stack put value in proper reg.
save → push values in stack

DATE

this will likely to be read

swtch(&(c→scheduler), p→context);

as passing add may it will change later

p is process to be sched...
c → struct cpu

~~In swtch.S~~
~~global swtch~~

~~eax = old, edx = new~~

~~pop & put value in esp & return~~

p → context → struct context
c → scheduler → also struct context

---

swtch → assembly code in swtch.S

// void swtch(struct context **old, struct context *new)

↳ here it will save current registers on stack, in struct cntx
at add in *old
switch stack to new & pop previous saved reg

eax = old, edx = new

---

in switching stack
*old = updated old stack
esp = new = edx

So now stack pointer becomes add given by p→context and &(c→scheduler) this add was save on earlier stack.

So now stack started pointing to kernel stack of new process

ret → pop from stack one more time (eip will be pop)
// here will pop from esp → fun where to return
→ pop & return to that location

& here it will not return to thB (switch in sheduler)

sched() ← call by
- exit
- ~~yeild~~ yield
- sleep

process executing → timer interrput occurs → came to trap → check condn → call yield.

In yield()
put myproc() → state = RUNNABLE

shed calls swtch
swtch (&p→context , mycpu()→sheduler);

context of currently running process will be pushed on its own kernel stack.
So context will be pushed &p→context will be made to point it.

mycpu() → scheduler → already pointing kernal's it's own stack.
from here you load ther context of kernel.

context of kernel → same context that was saved when switch was called from the scheduler

So calling this, thB will not return here.

It will return to swtch in scheduler

Now scheduler fun run in loop again once again call swtch

If that process was context swtch out as a part of shed (→ yeild), then that process will ~~continue~~ → return in shed.

So one possibility of returning process B here.
so one possibility of start executing ~~star~~ process start from next line of swtch in yield

so it will return yield
from yield it will return to trap

so whichever process B schedule will 1st return from kernel & then will start returning to application code.

In sleep

```
p->state = SLEEPING;   } put process in sleep (chang state)
sched();                 call shed
                         // like blocking
                         //so loop of scheduler will not select it
```

There B wakeup!
Find out particular process
p->state = RUNNABLE ; // change state

IPC (Inter Process Communication)

process
Independent → run their own don't need anyone

cooperating → can affect or get affected by other (including data)

Two type → shared memory → common memory
→ message passing → message pass from kernel (kernel has to do more work)

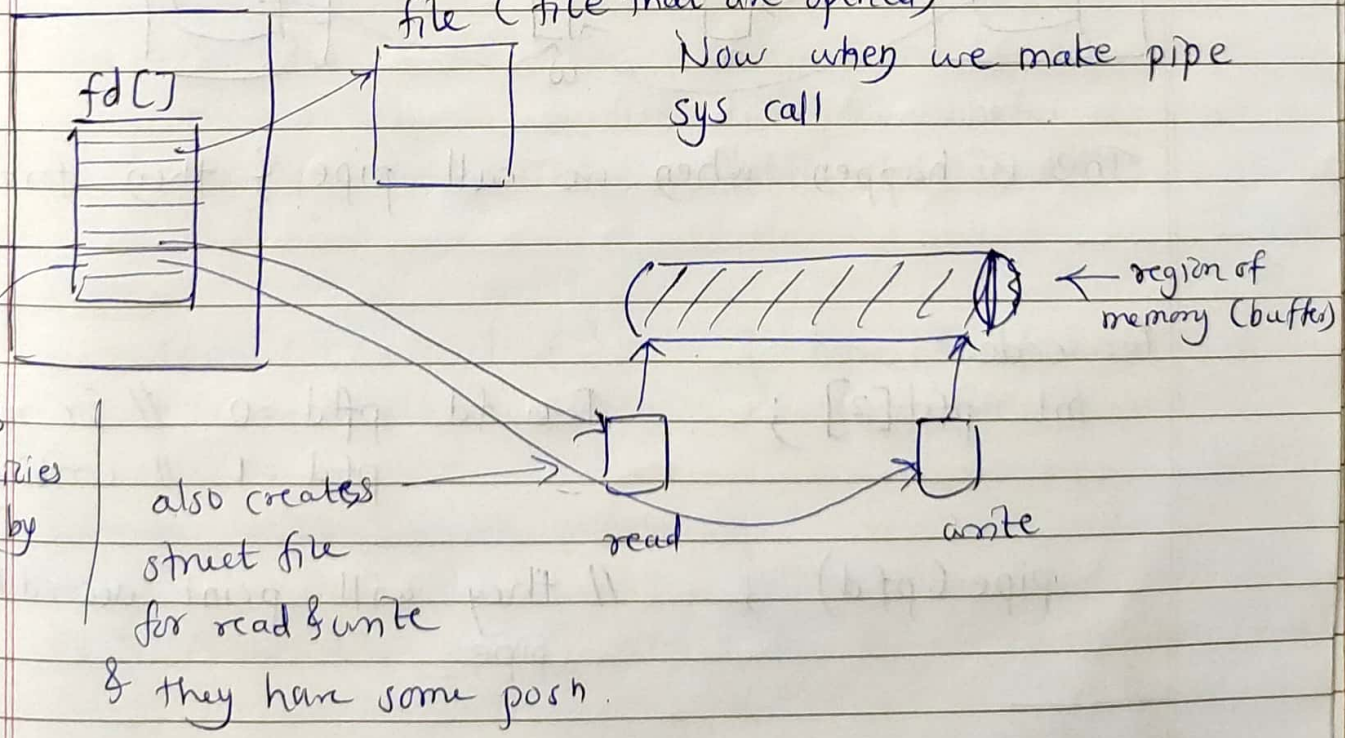
In shared memory kernel simply availables common memory

---

pipes → ordinary/unamed (also given FIFO)
→ named

---

for process it creates PCB, which has file descriptor:



So this two indizies written by kernel

also creates struct file for read & write & they have some posh.

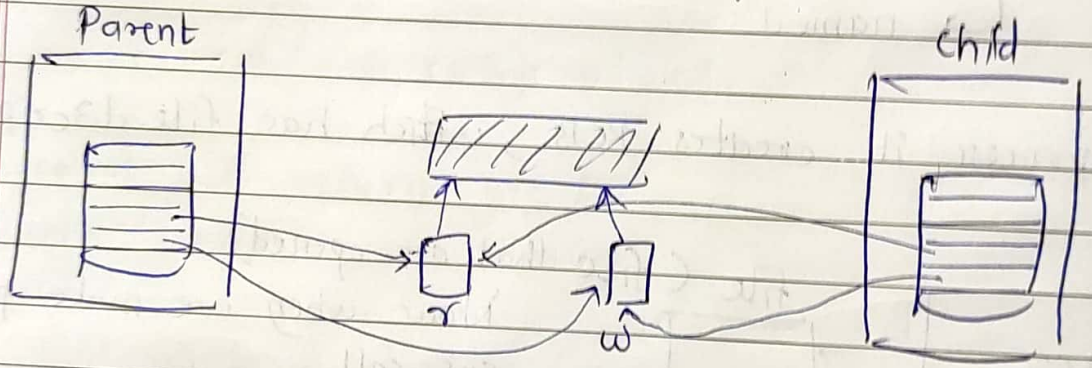
And kernel make pointer from fd[] to pointing read write

~~So this~~

In fork PCB gets duplicated.

Suppose pipe is called & then fork is called & in fork PCB get duplicated then

Then fd[] of child PCB also points to read write from same indices
ie, buff is shared betn two (parent & child)
Buff is part of kernel ~~memory~~ not a part of process data.. They have only access to read & write thr' fd[].
OS job to insure read & write happen in FIFO



This is happen when we call pipe & then fork

---

for code

int pfd[2]; Two fd pfd=0 // for reading
pfd=1 // writing

pipe(pfd); // they will point accordingly to pipe

---

while writing or reading from pipe
only one end must be open.
It's unidirectional communication mechanism

So for writing,
close (pfd[0]); // require to close unused end
write (pfd[1], string, no. of bytes);

for reading,
~~read~~ close (fd[1]);
read (pdf[0], baf, no. of bytes to read);

---

kernel often does delay write

---

close(0);
fd = open (filename, O_RDONLY);

Ata eta adhi 0 fd close kelay & nantr file open kelay tr tya file la 0 file descriptor bhetel.
As open try locate 1st available slot in fd.

Now,
scanf ("%s", buf); } here scanf will read
printf ("%s \n", buf); } from file.

~~This is redirection of stdout~~

---

ls > /tmp/file (output of ls stored in /tmp/file)

---

shell does → created child but calling fork →
in child before exec, simply close(1) and open
/tmp/file → so output got written in /tmp/file

< ⇒ std i/p redirection
> ⇒ std o/p redirection

---

dup → duplicates file descriptor

---

cat /etc/passwd | grep jay

Here, cat will not write entire output to the pipe & then grep will read

cat & continuously writing to the pipe & grep & continuously reading from the pipe

Both cat & grep exist at same time

- Named pipes → FIFO
process can create a file ~~cat~~ acts as pipe.
more powerful than ordinary pipes.

int mkfifo (const char * pathname, mode_t mode);

mkfifo /tmp/xyz
ls -l /tmp/xyz
→ prw-rw-r-- 1
↑
at start p which not present for normal file, kernel knows it & special file

Shared Memory

shmget (IPC_Private, size, S_IRUSER | S_IWUSER);
- IPC_Private → key
- size → size

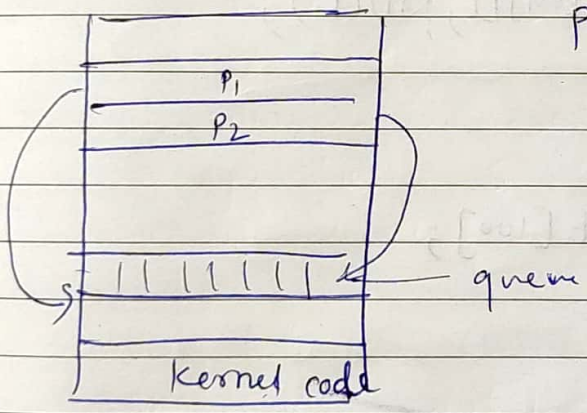
ptr = (char *) shmat (id, NULL, 0);
- ptr → to shared mem

Now can read & write from it
sprintf (ptr, "string");

To detach shared mem from its add space
shmdt (ptr);

Message Passing

P_1, P_2 2 processes and they want to talk using msg passing



Kernel creates / makes available queue of msgs to processes.
Now, P_1 & P_2 can access to queue & they can communicate.

Kernel job is to create communicating link bet processes

---

key = ftok ("str", no); // creates unique key

↑ To generate same other process should know these two parameters.

msgid = msgget (key, 0666 | IPC_CREAT);

// msgget creates a msg queue
// with help of id, can access to queue

~~msgsend~~ // msgsnd to send msg)
msgsnd (mgsid, &message, sizeof(message), 0);
↑ data

In other file read.c
msgrcv (msgid, &message, sizeof(message), 1, 0);
↑ saying that want msg of type 1.

// To destroy msg queue
msgctl (msgid, IPC_RMID, NULL);

struct msg_buffer {
long msg_type;
~~long~~ char msg_text[100];
} messag;

sys_pipe()

argptr() // use to fetch arguments passed by user in the system call.

argptr(0, (void*)&fd, 2*sizeof(fd[0])) < 0

pipealloc(&rf, &wf) → // modify rf & wf
// allocate array & make
// rf & wf point to array

fd0 fdalloc(rf) // → will allocate index in file dB table & make ptr point to rf.

---

pipealloc()

*f0 = filealloc() → gives struct file

kalloc() → give a page frame of 4K

---

struct file {
enum { FD_NONE, FD_PIPE, FD_INODE } type;
;

struct pipe *pipe;
struct inode *ip;
}

Treating pipe also as file & file also as file. So one of ptr we will be use
This for pipe & file is actual pipe for file

Type will be decided on variable type

When we read from pipe it will call sys_read()

sys_read() → fileread()

fileread (f, addr, n)
- f: file ptr
- addr: char *ptr
- n: no. of bytes to be read

fileread () {

```
if (f->type == FD_PIPE)
    return piperead(f->pipe, addr, n); // read from pipe
```

Similar to next cond, check for FD_INODE
Then it will read from file acted on disk.

piperead (struct pipe *p, char *addr, int n)

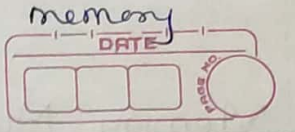
// code
// waiting if the queue is empty

Then in loop reading from pipe & putting one byte after one in addr[i]

wakeup(&p->nwrite); // wake up writer
// Bcz there can be writer which was waiting bcz queue was filled.

relase(&p->lock);

ipcs // shows all existing shared memory



pipewrite ( )
// Do almost same thing
// wait if pipe was full
// write data into array of pipe

wakeup (& p→nread);
release (& p→lock);
return n;
}

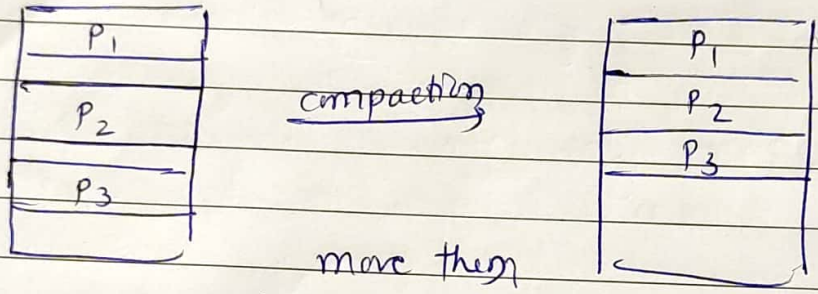
sys_write → filewrite → pipewrite
calls
calls

Memory Management

variable partition scheme
strategies to find free chunk → Best fit
→ worst fit
→ First fit

External fragmentation problem
30K, 40K, 20K chunks free
Need 50K
→ Total free = 90K, still can't allocate 50K

Compaction



more than
to be continuous

Fixed sized partitions
→ lead to internal frag
→ 4K normal size

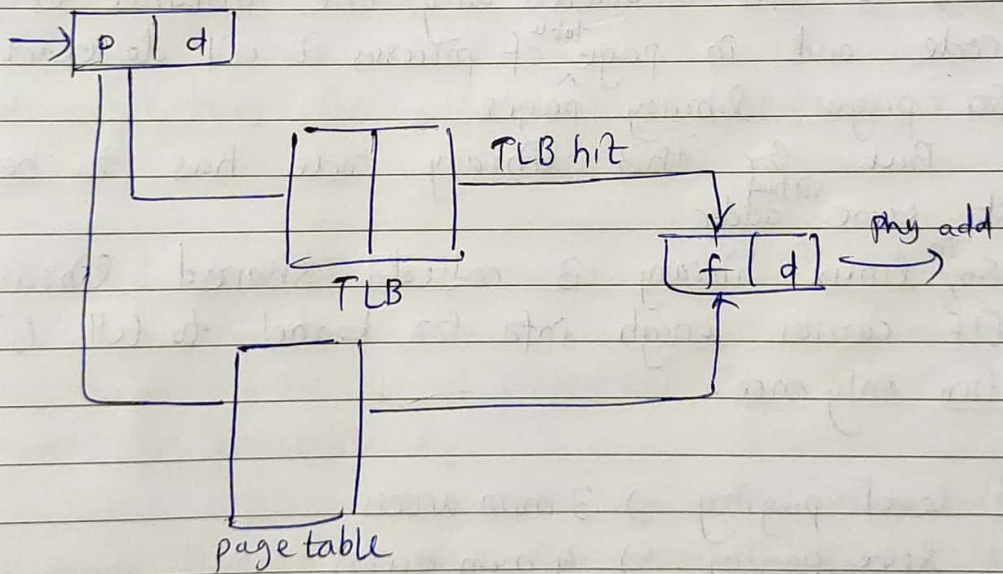
~~after~~ During context switch of the process, OS load the PTBR using PCB.
once this done, Then process B ready to run.
All the memory add translation for process will be done by hardware. (Process will run as if it is running its own)

OS has to do some global level task.
→ maintain a list of all page frames

TLB entries can be maintain by → Least freq use
most freq use
Irrespective of it TLB speeds up.

- DB advantage of pagin
  memory access results in 2 memory access
  - → 1 actual memory access
  - → 2 page table

Translation lookside buffer (TLB) → 2 entries
↳ It's subset of page table entries
↳ doesn't frame no. eg ~~to~~ page number to number contains less,
So it has to maintain both page no. & frame no.



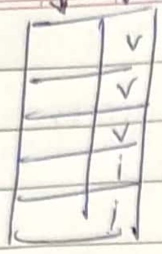
First try to look up in TLB if get it is TLB hit & if don't it's TLB MISS, then look it for in page table

Now if TLB hit → 1 memory access
TLB hit → 2 memory access

TLB access & updation is done by hardware automatically. There is no kernel involve here.

Memory protection with paging

frame number ↓ ← valid-invalid bit



// with valid & invalid bit, can raise interrupt (memory vi...

// illegal memory ac...

// kahi veles tevdhy size che page table kernel alloc... ~~masel~~ kami masel shakat asnar, tr remaining la invalid d...

---

shared pages (eg library with paging)

suppose P_1, P_2, P_3 uses same library code while normally allocating pages for process it will alloc... as seprate library for P_1, for P_2, P_3 (End up allocati... same library pages multiple time)

But If kernel can identify which part of ELF file is library code and which part of file is not library c... Then it will allocate only one instance of library code and in page table of processes it will do exact mappi... for pages library pages.

But for this library code has to be map at same virtual addr.

So, In linux library is called shared library. ELF carries enogh info for kernel to tell load library fun only once.

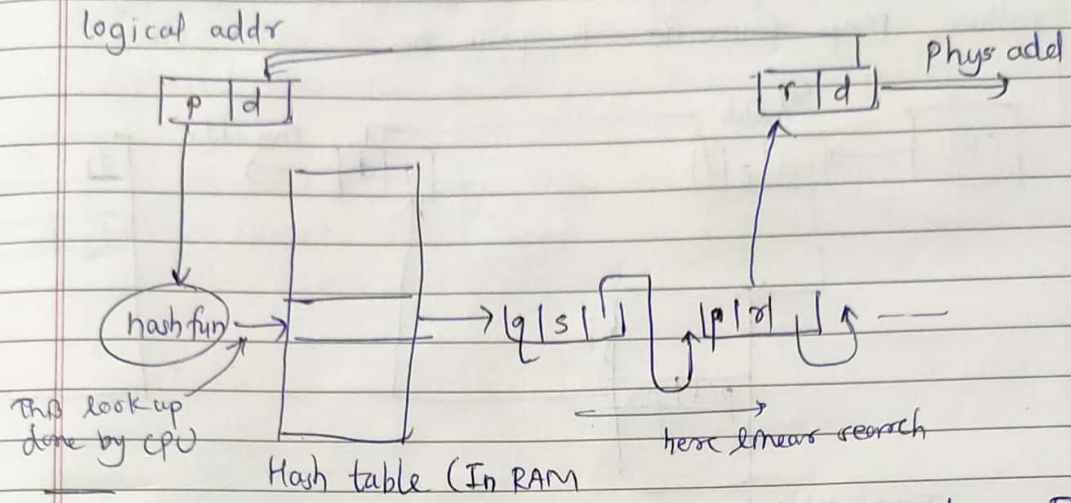
→ 2 level paging → 3 mem access

→ 3 leve paging → 4 mem access

Hierarchical Paging

Scanned by CamScanner

Hashed page table



p → page no. It doesn't use as index. It pass thr' hash fun. That will be index in hash fun.
In hashing for problem of collision → chaining is used.

Here entries for q & p collided. From p, r is used.

Inverted page Table

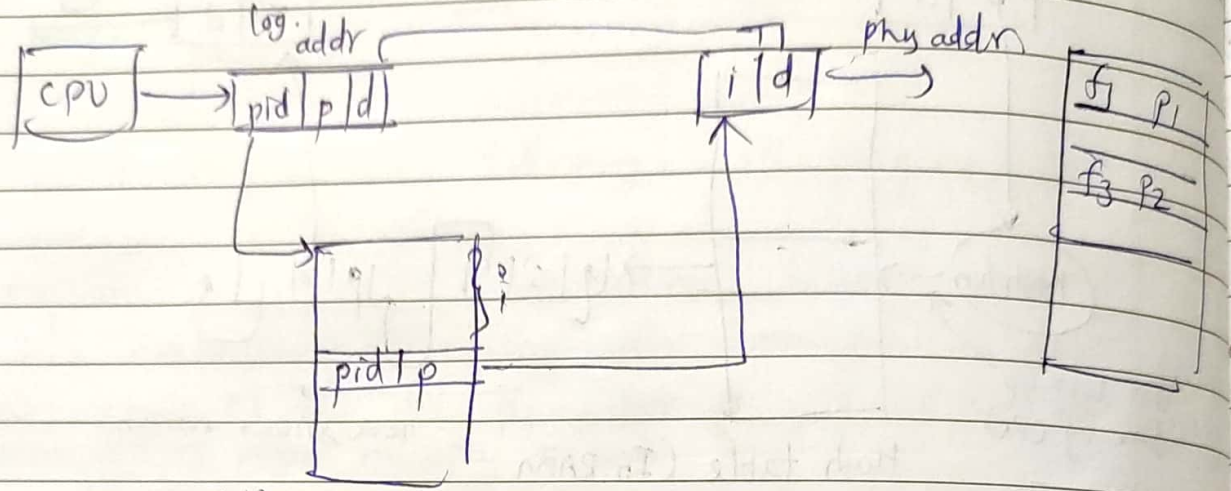
we have process P_1 → page table P_2, pro P_2 → p table P_2, pro P_3 → page table P_3

Now all of they think as entire ram belongs to them, that why they have mapping to the same RAM.

So, more no. of pages, mapping all to same RAM
All mapping to same page frame in RAM.
RAM is same.

So think inverted.
No matter how many process → they still have to be contained within same memory.

so didn't map page no. to frame no. (As page no. for every process) rather map frame no. to page no.

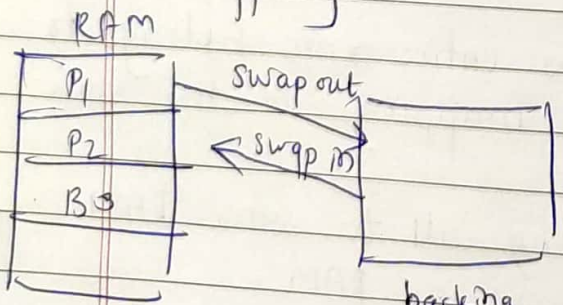


Problem is that f_1 can belong to P_1, f_3 to P_2 so also store pid & p.no.

This is one global table not perprocess table. There are no perprocess table.

pid → not generate by ~~kernel~~ CPU
→ loaded by kernel when process executing

Swapping



Suppose there is no space for P_4 in RAM, But a time we can run only one process, so we can swap P_1 & allocate P_4 so we can allocate pages more than efficiency of RAM.

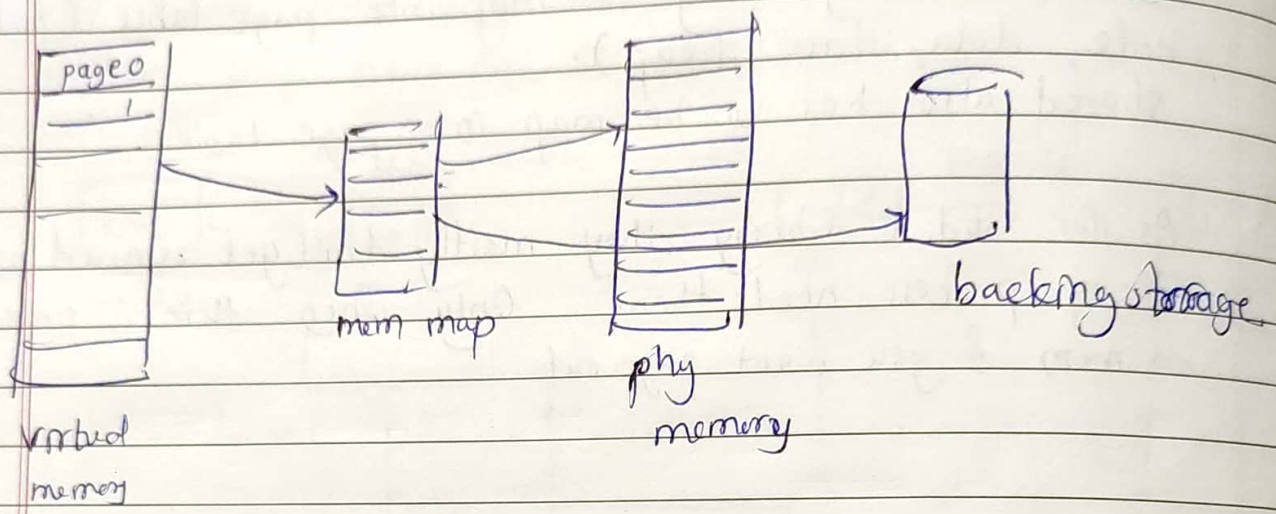
of process

Entire memory img is map into page table (including code, data, stack, heap)
shared also has to be map into page table.

As for std C library they mostly don't yet removed as every process needs them. Only when there is pressure on mem & you want page out.

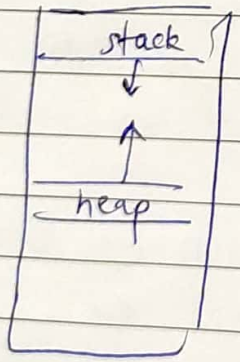
sdrk → sys call → ~~allocate~~ ask kernel to allocate certain region in virtual mem & return ptr to it.

Virtual Memory



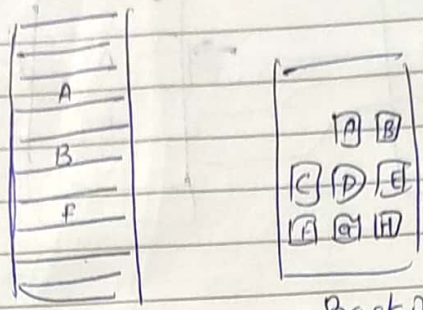
vir > phy. So job of OS to creates mem map (like page table) it will not only map in RAM but some also in backing storage.

Virtual add space



→ Normally yasathi pura space adhich allocate keli asti pn in virtual add space stack la 1 page & heap 1 page and ~~they~~ later size can be increased (left for growth)

Demand Paging



phy mem. Backing storage

Pages in phy mem also in backing stg as we never know which one go out & which one come back in. So in backing stg → all loaded.

---

valid-invalid bit

i → not in mem or illegal → raised trap called page fault

---

memory 0×100, 0×110, 20

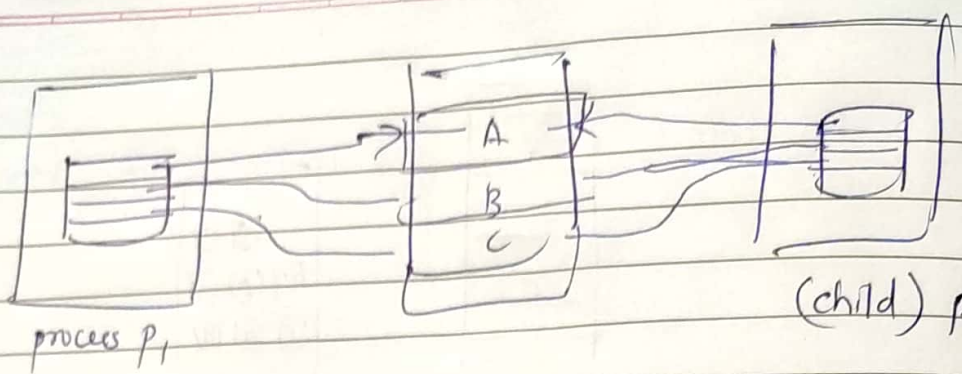
Yamul overlapping hoil & this will lead to page fault.

Jr ekhadi inst partially over asel ani mg page fault ala tr te kas revert back honar.

↳ for demand paging → CPU has to insure with something goes wrong it has to revert back as if nothing is happen.

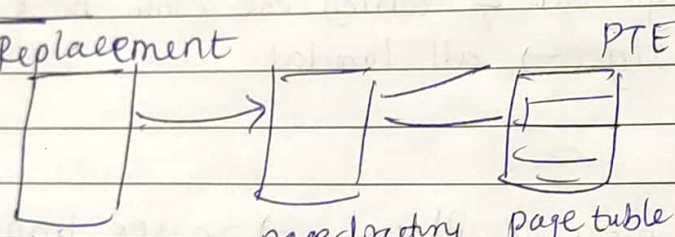
---

i → while bringing page from backing storage to phy. mem → Involes read from → takes some milisecond → So, Meanwhile scheduler will schedule another process.



~~Initially~~ ~~sent~~ In ~~PCB~~ P_1, P_2 all the pages get shared
In PCB there should one ~~extra~~ bit.
Indicating there should copy on write

Page Replacement



one more bit,
PTE has ~~dirtybit~~ dirty bit get set by hardware when you modify corresponding page.
only modify pages only to disk.

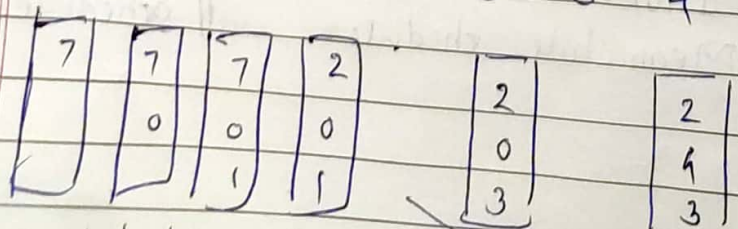
Page replacement algo.

1) FIFO

2) optimal page replacement

next hya entries madhe 0 sarvat aadhi ahe so tyala replace kel

7 0 1 2 0 3 0 4 2 3 0 3



Eth check kel tr next chya reference madhe 7 lait ahe. so tyala replace kel

1 la replace kel

optimal is not possible

least recently used (LRU) → approximation of optimal.

---

stack algorithms

for n frames

No. of page fault >= n+1 frame no. of page fault

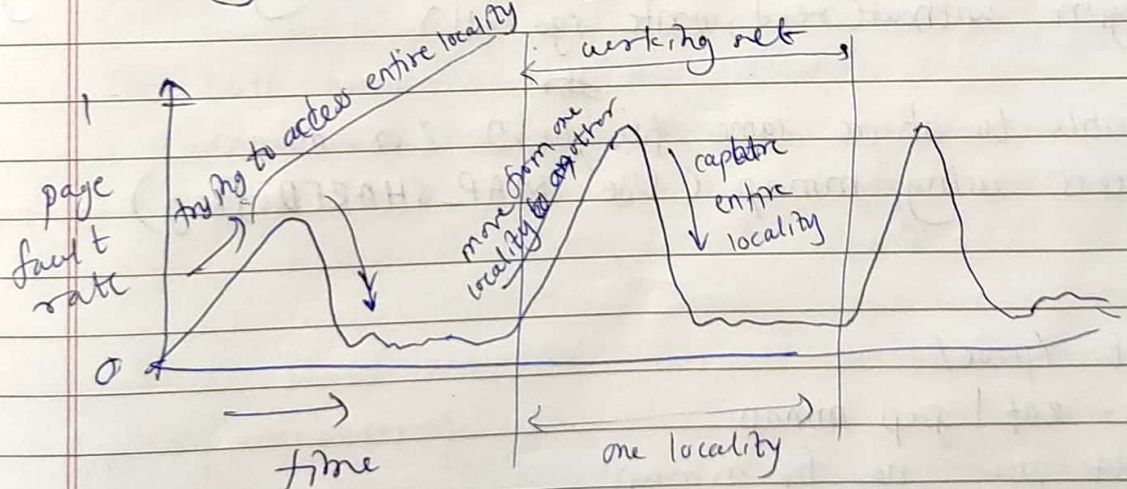
Thrashing

CPU utilization is low when

- less no. of process
- more no. of process, but most time in page fault

---

working sets and Page fault rates



```
fd = open(argv[1], O_RDWR);
data = mmap(NULL, 4096, PROT_READ|PROT_WRITE, MAP_SHARED, fd, 0)
```

PROT_READ|PROT_WRITE → marks as read & write

MAP_SHARED → can be shared with other process

NULL → map 1st 4k bytes into memory region

and returning ptr to region.

When mmap call is did, → mapping done only conceptually
→ so basically page table entry created for process
→ having storage in the file
→ Now get access in memory region

```
munmap(data, 1024); // undo mapping
close(fd);
```

Here you read & write from file directly from memory region without (read, write sys call).

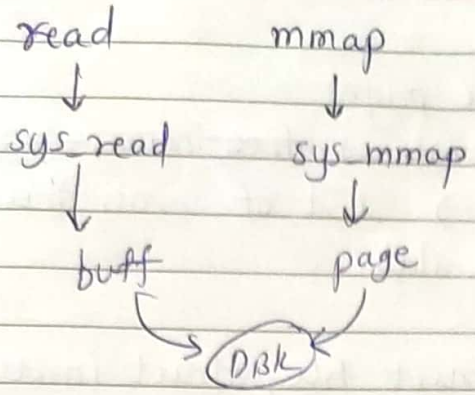
---

possible to share same file betn 2 or more process using mmap (Use MAP_SHARED flag)

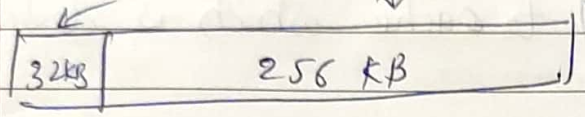
---

cat /proc/
ps - eaf | grep mmap
→ get proc no. for mmap

→ cd no.
cat maps → memory map of process



Buddy Allocation ← physically continuous pages



Suppose need 30 KB

Break 256

128 128

64 64

32 32 → stop and allocate 32 KB to it

Ith kuth mem. we ahe kas identify karayach

→ kernel will set set limit

→ suppose here set lower limit to be 1KB ~~that~~ is i.e below it won't get divided.

So, 256 / 1 = 256 So it will have 256 bit long bitpattern.

Suppose here ~~32~~ KB B at start, so at start all starting 32 bits will be set to 1 from 0.

Slab allocator

Slabs → set of continue pages
when kernel loaded → ~~combines free frames (continuous)~~
→ create list of free frames → list of conti free frames
→ combine some frame into slab.

---

kernel often needs struct proc, struct file, struct inode, struct buffer etc → called as kernel object.

Set of slabs is allocated to cache which is dedicated to one type of object.

Take a slab → divide it into equal size chunks of that ~~size~~ object type

---

let say, So here preallocating lot of, ~~let say~~ struct proc here and when there is request to get one of them, you simply take one and give it back

---

Here preallocating → so wastage of mem.
Advantage → caches are already there
when loaded kernel → kernel created slabs
→ from slabs kernel created proc cache, file ~~cache~~ Inode cache, IO cache etc
→ so very fast, very quick
→ No internal fragmentation but wastage if don't use all completly (don't use all cache completly)

Threading

Thread create →
seprate concurrent flow of execution ß created

set of inst executed independently.
process ß also thread

concurrency → progress at same time
parallalism → execution at same time

→ suppose f_1, f_2 ahe tr my f_1 kahil mili sec my f_2 mili sec sathi as Eth as nahiy ki f_1 purn execute hotay my f_2 hotay.
Eth parallaly dogh sobat ~~sutu~~ ~~astat~~ run hotat (multi core)

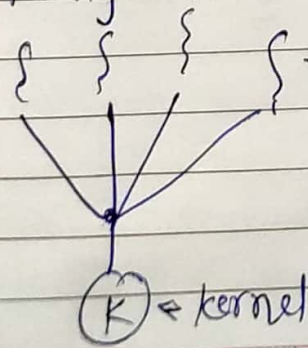
Process creating ß typically heavy weight, (amount of time)
Thread light

Going do fork, exec → ß fork going to duplicate everything

→ Part of fun → going to duplicate only some part of it

Pratically it switch to context switch while scheduling diff threads.

Many - one model

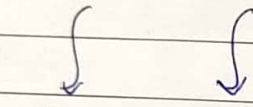


Trouble with thß ß
Suppose ~~thß~~ ~~one~~ thread user thread ~~make~~ does ~~call~~ scanf,
→ make sys_call → thß syscall will suspend the process → so process get blocked → So what happended to other threads → they also get blocked.

One-one Model → need to many resources

```
main {
    pthread create (
}
}
```

```
pthread create (— f1 —)
    clone (f1 —  )        library
}
```



If don't have clone → then many ~~to~~ one

```
pthread create (— f1 — )
    thread {
        context c
        stack ps
        // other things needed
    }
}
```

```
th = new struct
setup th
switch ()
```

Read → clone
getcontext, makesetcontext
setcontext

Thread pools
→ create threads without scheduling

Scheduler activations for thread.
suppose 4 user threads, 2 kernel threads
many to many

Application
```
f(){
  scanf();
}
g() {
  recv();
}
```

→ after allocating thread to process
→ when scanf → sys_call → thread block
→ they call sched → goes to scheduler → schedules process

As here we don't know thread get block, so that we can create new thread

On thread B alway ON which should not block → scheduler

SIG_KILL → Note says Not possible to handle sigkill

```
int *p = 1234;
main () {
signal ( SIGSEGV, seghandler );
*p = 1000;          // writing at any random pointer
                    // it will handle in thB (segfault)
```

we can modify that fun

but if segfault occurs before thB func line of code, then that segfault will be handle by default signal handler.

```
int *p = 1234;  // initialize any random loc
void seghandler (int signo) {
    printf ("Seg fault occured \n");
    return;
}

int main() {

    signal (SIGSEGV, seghandler);
    *p = 100;   // Try to write at random location.
    return 0;
}
```

Here we are override default handler.

~~Basic~~ B

The default action for SIGSEGV is to terminate your prog. But here we ~~are~~ override.

So after this line of code, for every segfault our code will execute.

Q- Why signal handler goes to infinite loop?

As we override, so for every instruction that triggers a sigsegv, this (our) handler is called & the instruction is restarted. But here our handler does nothing, ~~it doesn't fix anything, to fix~~ It did nothing to fix what was wrong in the first place with faulting instruction.

When instruction is restarted, it will fault again. That's why it goes in infinite loop.