Operating Systems

PAGE NO.:
DATE: / /

Motherboard:

* Motherboard comes with a capability to expand by adding more I/O ports.
* It's a fully-functioning computer, once it slots are populated.

x86 Instructions:

1. Data movement: MOV, PUSH, POP ..
2. Arithmetic: TEST, SHL, ADD ...
3. I/O : IN, OUT
4. control: JMP, JZ, JNZ
5. String: REP MOVSB
6. System: IRET, INT

Memory (RAM):

- same time access to any location
- semiconductor device.
- When a system is started, there is nothing on RAM.

Hard Drive:

* Head is responsible for reading & writing data onto HD.
* IDE, SATA, SCSI, PATA, SAS

SSD:

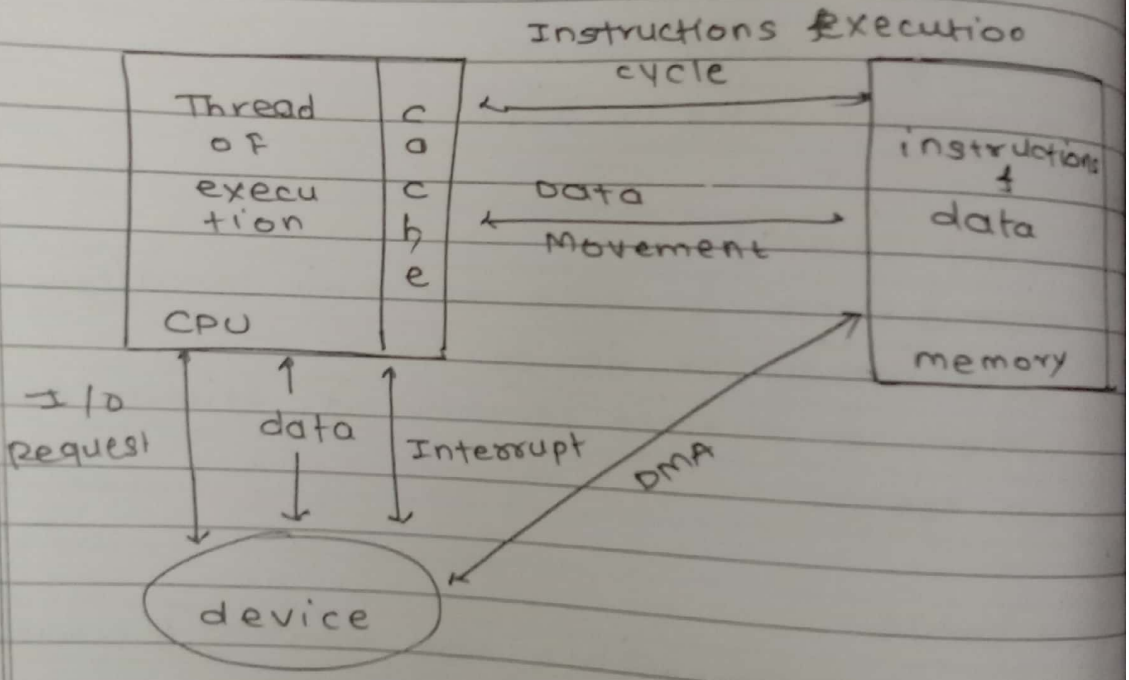
* Permant storage
* Reliable, no moving parts.

* Smaller access time = costlier storage (~~Sto~~ re

* Access time $\propto \frac{1}{\text{storage capacity}}$

* Register → compiler
  Cache ⇒ Hardware
  RAM, SSD, HD → OS

Von-Neumann Architecture:



Important:

* CPU transfers data b/w itself & RAM or No transfer b/w CPU & HD, CPU & mouse

* I/O devices transfer data to RAM & CPU instructions access data from RAM.

PAGE NO.:
DATE: / /

hat does processor do?

* From its being turned on to turned off, it does.

① Fetch instruction from RAM Location given by Program counter (PC)
② Decode instruction & execute it. While doing this may fetch some data from RAM.
③ While executing the instruction change/ update PC.
④ Go to 1.

oot loader:

- A program exists on (sector 0) of secondary storage.
- Loaded by BIOS in RAM & passes over control to.

ernal:

- Code that is loaded and given control by Boot loader intially when computer boots.

System Programs:

- Applications that depends heavily on Kernal and processor.

OS(kernal):

- Takes control of hardware
- Loads some initial applications("init") from HD into RAM & passes control to it

* BIOS code is Machine code.
* Compiler is used to compile applications on particular OS & hardware.

* BIOS, Bootloader are software.

* Debian/Ubuntu: Kernal + System programs + selected applications.

Event Driven Kernal

* Boot process
  BIOS → Boot loader → OS → "init" process
  ↓
  sector 0 of boot device

* PC on start has address of BIOS at start by default.

* Hardware devices (keyboard, mouse, hdi, etc) can raise hardware interrupts.

* on hardware interrupt, pc changes to location pre-determined by CPU manufacturers.

* ISR codes are part of OS code. OS occupi that ~~process~~ place in boot process.

* Whenever there is hardware interrupt, OS code will Run.

* Taking control of hardware = All ISR location are occupied by OS code.

* CPU fetches instruction from RAM.

PAGE NO.:
DATE:

Time sharing CPU

- CPU is timeshared blw diff. appn for OS

Multiprogramming & Multitasking

* Program: A binary file(machine code) on hard drive
* Process: A program that is executing
  - must exist in RAM to execute.
* Multiprogramming: A system where multiple processes exist at the same time in RAM
* Multitasking:
  Time sharing blw multiple processes in a multi-programming system.

Imp

1) A multiprogramming system is not necessarily multitasking.

* Events that make CPU run code at pre-defined places:
  1> Hardware Interrupts.
  2> Software interrupts (int)
  3> Exceptions:
     (divide by zero)

* OS doesn't run unless there's an event.

* Two types of CPU instructions
  1> Normal 2> Privilage

* Two modes of CPU operation
  - CPUs have mode bit (0 or 1):
  - If bit is in user mode; normal instr are executed by CPU.
  - If bit is in kernal mode, both norma & priviliged instructions are executed.

* OS code runs in kernal mode.
* Application code runs in user mode.

* Transition from user mode to kernal mode & vice versa:
  - It can be done by software interrupt instructions (ex. INT)
* Before INT instru, CPU is in user mod
  After executing, kernal mode.

* Hardware interrupts & exceptions happen asynchronously while software interrupts are caused by app^n code.

System Calls

* Any user or application program can call system calls to avail availabilities given by (By calling software interrupt instruction)

* Interface provided by kernal.

Process in RAM

- Memory is required to store some comp of a process:
  - code
  - Global variables
  - stack
  - Heap
  - shared libraries, etc.

PAGE NO.:
DATE: / /

Fork():

- If a process calls fork, it creates identical duplicates of it.
- calling function returns in two places
- caller is parent and new process is child. PID is returned to parent and 0 to child.
- Memory addresses of variables after fork are different.

Exec

- Takes path name of an executable as an argument.
- Overwrites existing process using the code given in executable.
- exec never returns if it succeeded as caller gets vanished.

DATE: / /

Compiler, linker, loader, Assembler

- System programs/ utilities most essential to make a kernal usable.

Standard C Library

- Machine code containing code of all the most frequently used functions.
- /usr/lib/x86-64-linux-gnu/libc-2.31.so
- .so = shared object file

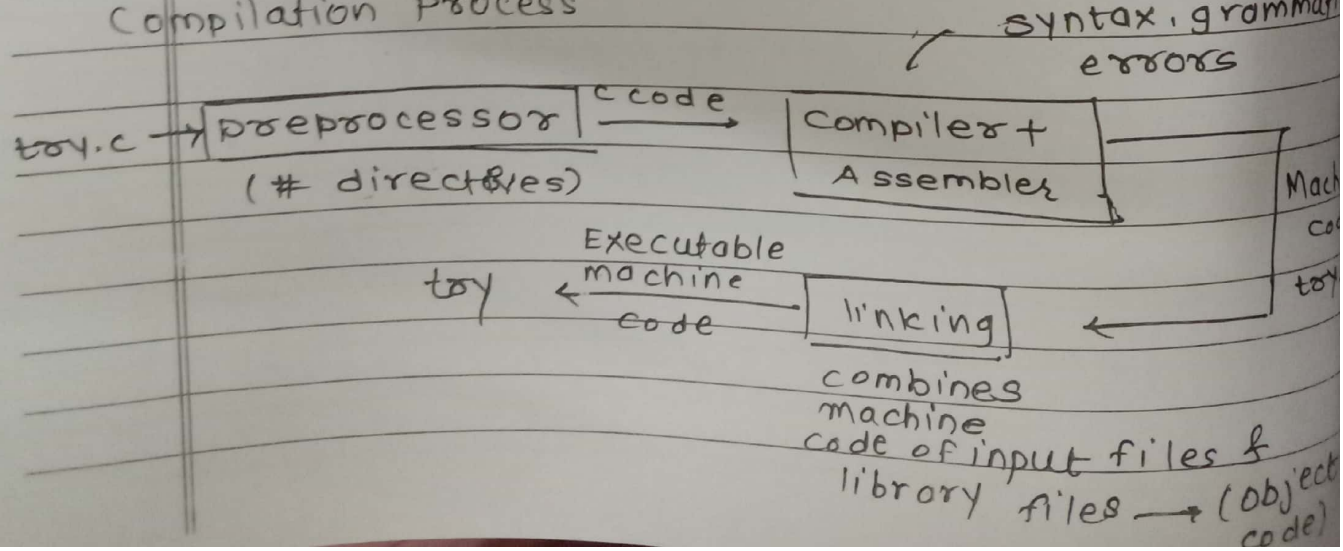
Compiler:

- Application program
- Input is file & output is also a file.
- High level language to machine code.

Assembler:

- Application program
- converts assembly language to machine co

Compilation Process



* Linker
- Application program
- on linux, its "ld" program.

Executable File Formats
- It must be dependent on processor + OS to execute.
- contains machine code + OS

- Windows: PE
- Linux: ELF
- Old Unixes: a.out (no extension)
ELF file
- ELF is used for both executable programs but also for partially compil

- a.out: ELF format (No longer use o

- objdump:
objdump -D -x /bin/ls
- shows all disassembled machine instructions & headers.

- hexdump
- shows files which are in hexadecimal

- readelf
- Alternative to objdump

or

- create statically linked library fil.
- ar -cvs libmine.o one.o two.o

Loader

- The execl()
- Loads an executable file in memory

Important

- memory management features of processor, memory management architecture of kernel, executable / object-code file format, duties of linker and job of loader are all interdependent and inseperable.

Calling convension

- compiler needs to map the features of C into processor's features and then generate machine code.

- Local variables are located on stack
- Sequence of function calls are presen on RUN time.
- compiler has to generate machine code for each function.

Function calls:
- LIFO order.
- Processors provide stack point (%esp)
- Push & pop provided by hardware.

System calls, compilers
- compiles generates a machine code whi can be played or supported by esp.
- There can't be any C code compiled which wil run on a processor with no esp.

* what goes on stack!
  1) Local variables
  2) Function parameters
  3) Return address of instruction which called a function.

Activation Record:

- Local vars + parameters + return address
- One activation record for each function call which get destroyed on return.

- ebp & esp used to denote activation record

x86 Instructions:

① leave

```
mov %ebp, %esp
pop %ebp
```

② ret

```
pop %ecx
jmp %ecx
```

③ call x

```
push %eip
jmp x
```

④ endbr64

Normally a no operator (processor wasting one CPU cycle)

* More paremeters, more registers and hence more sophisticated way to do calling convention.

Function which call makes

Function which gets called

PAGE NO.:
DATE: / /

Caller save and callee save Registers

① Caller save Registers
- compiler need to save register used by caller. They can be used by callee fun

② Callee save Registers.
- Registers will be pushed by callee functi it self.

③ How to return values:
- caller will have to pop.

* Caller Saved: EAX, ECX, EDx
* Callee saved: EBX, EDI & ESI

8086

Address Extension

* pa = va + seg * 16

Ex. (i) %ip + %cs * 4

Flags Register

- set by processor depending on previous instruction.

32 bit 80386

- Boots on 16 bit
- Running particular instructions switches to 32 bit mode (protected mode)

- 0x66 | 0x67 to toggle between 16 bit & 32 bit operands | address.

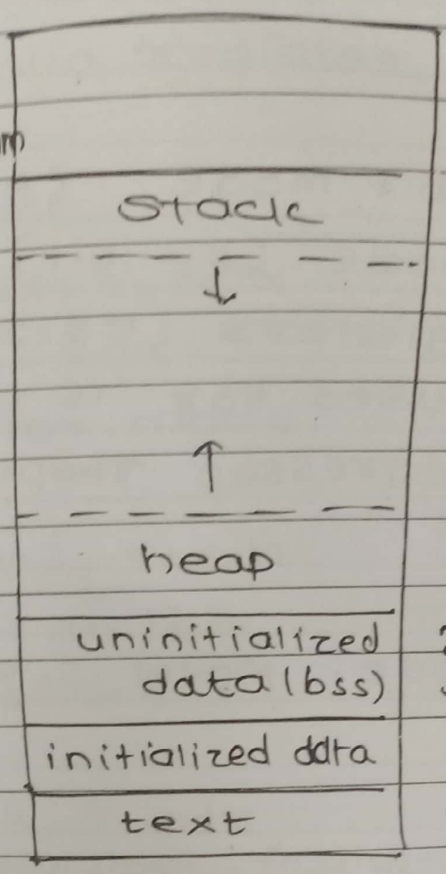
x86 Assembly code

- Intel syntax : op dst, src
- ATST (gcc(gas) : op src, dst (xV6)

* IDTR : CPU part IDT : memory

Memory Management in X86:

* Memory Layout of C program



data = global & static variables

* Text, data, bss are present in ELF file.
* stack & heap are not in ELF file.

Stack: Region for allocating local variables, function parameters, return values.

Heap: Region for use by malloc(), free()

created by OS

Basics of x86 Architecture

* CS:IP , SS:BP,SP, DS:ES , SI:DS
* Flags register can be set by processor depending on current instruction

32 Bit 80386:

- Boots in 16 bit mode (Real mode)
- Protected mode (32 bit)
- Segment Registers (cs.ss.etc) : 16 bit.
- 0x66/0x67 used for 16 bit and 32 bit operands/addresses respectively

Example

1) In 32 bit mode, MOVW is expressed as 0x66 MOVW.

Expected Use of segment Registers:

CS: Holds the code segment in which your program runs.

DS: Holds the data segment that your program accesses.

ES,FS,GS: These are extra segment registe

SS: Holds the stack segment your progra Uses.

- All 16 bit.

os sets values in registers.

Assembly code
- Intel syntax : op dst, src
- AT&T (gcc/gas): op src, dst (x86)
- uses b, w, l to specify size of operands i.e. 8 bit, 16 bit & 32 bit.
- operands are registers, constant, memory via register, memory via constan

x86 Instructions:

1. movl %eax, %edx
   ⇒ edx = eax
   ⇒ Register mode

2. movl $0x123, %edx
   ⇒ edx = 0x123 (storing number 0x123)
   ⇒ Immediate

3. movl 0x123, %edx
   ⇒ edx = (int32_t) 0x123 (operand at address 0x123)
   ⇒ Direct.

4. movl (%ebx) %edx
   ⇒ edx = (int32_t) ebx (Address of ebx pickup)
   ⇒ Indirect
   ⇒ Two fetches from memory

5. movl 4(%ebx), %edx
   ⇒ edx = (int32_t)(ebx + 4) ebx+4 addre
   ⇒ displaced

PAGE NO.:
DATE: / /

Instructions suffix | prefix

① mov %.ebx, %.eax # 32 bit
② mov %.bx, %.ax # 16 bit
③ movw %.bx, %.ax # 16 bit
④ mov $123, 0x123 # Ambigious
↓
operand

privilege levels

- x86 has 4 protection levels
  0: most privilege, kernal mode
  3: least privilege., User mode

- current privilege level with which the x86 executes instructions is stored in CPL field (2 bit) inside %.cs register.

- changes automatically on
  ① "int"
  ② hlw interrupt
  ③ Exceptions

- slw interrupt canbe used to create hlw interrupt

- ~~x64~~ ~~x86~~
- XV6: int 64 used for system calls.

PAGE NO.:
DATE: / /

Interrupt Descriptor Table (IDT):
- It is in memory table.
- 256 intries.
  - 0-31 are defined for slw exceptions
  - xv6 32 hlw interrupts in 32-63
  - Int 64 for system call.

* IDTR and IDT
- IDTR is in CPU (48 bit)
- Gives address of IDT
- Each entry in IDT is called gate.
- IDT is initialized by OS.

* Why segment: offset pairs in x86?
- segmentation: Compiler's view of program
- compiler generates object code assuming that different memory regions corresponding to the program are different segments in memory.
- segment: a continuous chunk.
- segment register gives the location of the chunk, indexloffset registe gives offset.
- Example
  CS: IP, CS gives location of segment & IP gives index in it.

Real mode and Protected mode:
- Not same as kernal mode & user mode.
- for booting: 16 bit
- on running particular machine instruction, CPU will change to 32 bit (protected mode)

* Real Mode:
- Address always correspond to real addr
- seg *16 + offset
- no support for memory protection, multitasking or code privilege levels (user/kernal)
- so no multitasking on 8086.

PAGE NO.:
DATE: / /

Virtual Memory

- virtual address is address issued by CPU execution unit which later converted by MMU to physical address.
- Employed by OS (with hlw support)
- virtual memory achieves ability to execute partially loaded program.

virtual memory

- seperation of user logical memory from physical memory
- virtual memory can be implemented by
  1) Demand Paging
  2) Demand Segmentation.

- It is the work of OS to map logical virtual memory with physical memory and backing store (ex. swap area).

Virtual Address Space

- Enable sparse address spaces with holes left for growth, dynamically linked libraries. etc.

- What is sparse address space?
  - We assign a page for stack and heap which can grow towards themselves. The area between them is what called as sparse address space.

sbrk() is a system call which asks kernel to allocate some region in virtual memory and return a pointer to that region.

PAGE NO

Shared Pages Using Virtual Memory

- ~~Shared~~ System libraries or shared via mapping into virtual address space.
- Shared memory by mapping same pageframes into page tables of involved processes
- Pages can be shared during fork(), speeding process creation

emand Paging

- Load a page to memory when its needed (on demand)
- Pure demand paging: Load ~~one~~ no pages at load time.

alid Invalid Bits in page table

- with every page table entry, valid-invali bit is associated.
- v : in memory - memory resident
  i : not-in-memory or illegal
- During address translation, if valid-invalid bit in page table is i, raises trap called 'page fault'

Backing store have space for storing all the pages in memory.

pure demand paging: Start execution of a code without page fault.

DATE: / /

age Fault

Hardware interrupt
- ⇒ change of stack to kernel stack
- ⇒ saving the context of process
- ⇒ switching to kernel code.

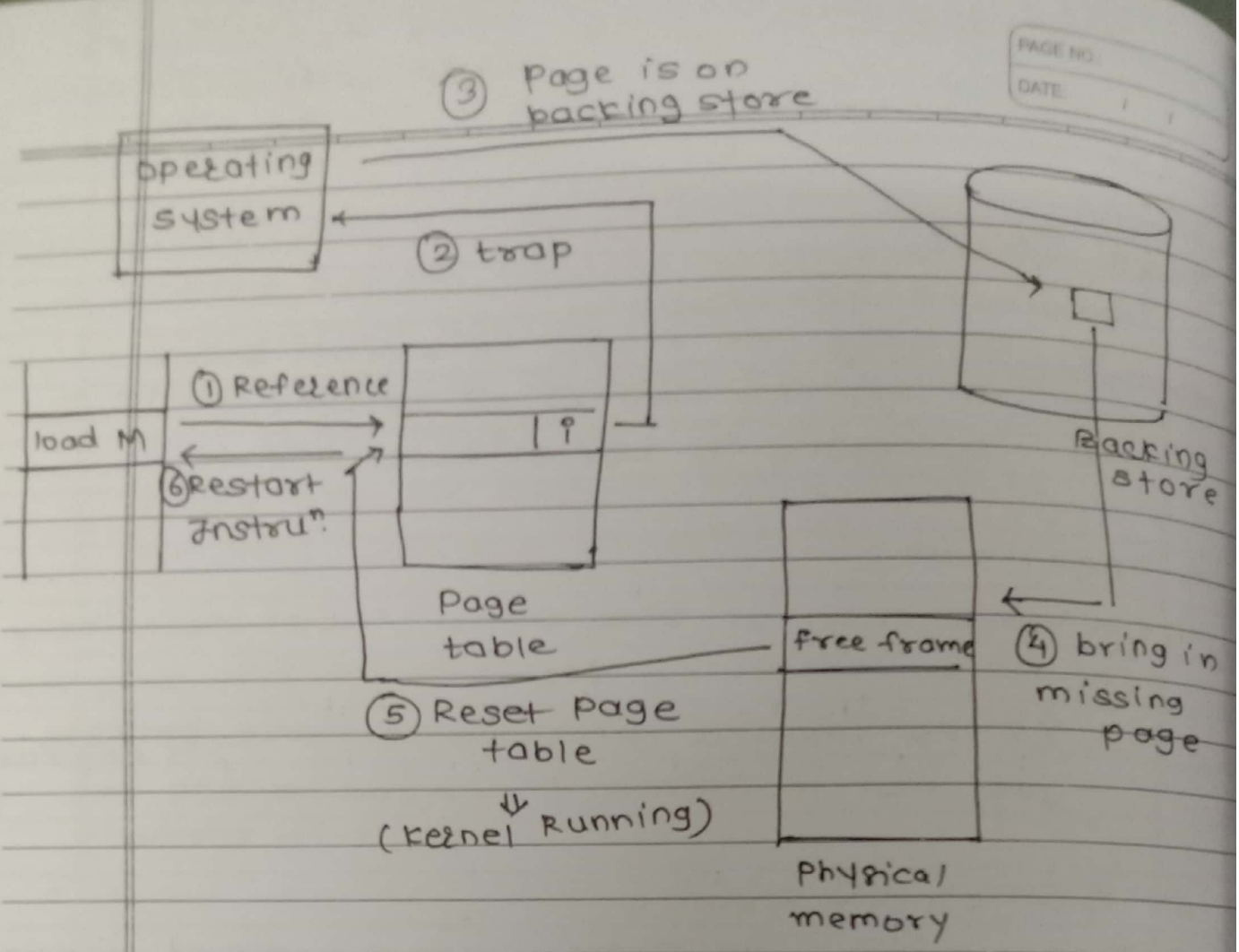
Page Fault:

1) OS looks into at Data stru. (table) most likely in PCB itself, to decide
   - If its invalid reference → abort process (seg fault)
   - just not in memory → need to get the page in memory

2) Get empty Frame
3) Swap page into frame via scheduled disk/IO operation.
4) Resets tables to indicate page now in memory
5) Set validation bit = V
6) Restart instruction that caused page fault.

Hardware support needed for demand paging
- Page table with valid/ invalid bit
- Secondary memory (Swap device with swap memory)
- Instruction restart

rocessor Related.



Page Fault Handling

1) Trap to OS
2) Default trap handling():
   save process registers & process state
   determine that interrupt was page fault.
   Run page fault handler.
3) Page Fault Handler(): Check the page referenc
   was legal and determine location of page
   on disk. If illegal, terminate process.
4) Find a free frame. Issue a read from
   the disk to a free frame.
   - process waits in queue for disk read.
5) Meanwhile many processes may get scheduled.
   - Disk DMA transfer data to free fram
   & raise DISK/IO interrupt.

6) while waiting, allocate CPU to some other process
7) Receive interrupt from disk I/O subsystem
8) Default interrupt handling():
- save registers & process state for other process
- determine it was DISK I/O interrupt.
9) DISK Interrupt Handler:
- Figure out interrupt was for waiting process
- Make process runnable.
10) Wait for CPU to be allocated to this process again
- kernel restores the page table of process, marks entry as "V".
- Restore user registers, process state & new page table, & then resume the interrupted instruction.

Performance of Demand Paging

- Page Fault Rate: $0 <= p <= 1$
If $p = 0$, no page fault
If $p = 1$, every reference is page fault.

Effective Access Time (EAT)

EAT = $(1 - p) *$ memory access time +
$p *$ (page fault overhead +
Swap page out +
Swap page in +
restart overhead)

Optimization : Copy on write

- During fork()
  - Pages of parent were duplicated
  - Equal amount of pages frames were allocated
  - Page table for child different From parent
- In exec()
  - Page frames of child were taken away & new frames were allocated
- Waste of time during fork() if the exec() was to be called immediately.

- Copy on write allows both parent and child processes to initially share the same pages in memory.
  - If either process modifies a shared page, only then is the page copied.
- COW allows more efficient process creation as only modified pages are copied.
- Vfork() variation on fork() syscall has parent suspend & child using cow address space of ~~contemporent~~ parent.
  - Designed to have child call exec()
  - More efficient

List of Free frames

- Kernel needs to maintain list of free frame
- At the time of loading kernel, list is created

* page frames in use depends on "Degree of Multiprogramming".

PAGE NO.:
DATE: / /

Basic Page Replacement (Two I/O)

1) Find location of desired page on disk
2) Find free from
   - If there is free frame, use it.
   - If no free frame, use page replacement algorithm to select a victim frame & write victim frame to disk if dirty.
3) Bring the desired page into free frame; update page table of process & global frame table/list
4) Continue process by restarting instruction that caused the trap.

(It doesn't fail on first time)

region_appages():
(2 level)
- Allocate page tables if required
- setup entries in page tables & page directories.
- If page table doesn't exist, allocate it and then compile.

walk_pgdir():
- Checks if pgdir has mapping with virtual address va.
  1) If it has mapping then return page table entry.
  2) Else allocate pagetable & return page table entry.

segdesc():
- Adds two additional entries in GDT for CS and DS but to be used in privilege level 3.

XV6 Paging:

* panic(): Enters in infinite loop or stops the kernel.
* At first time, r→next becomes nu on kfree() as in start there wer no pages allocated.
* All the addresses above 2GB will ge mapped from 0 to 4MB.
* First 4 bytes in page is pointer (r→n
* kinit1() gets 4K pages from 2049 t 2052 MB (allots) creates freelist of size 3MB(3072 p and assigns pages between 0 to 4MB vi

KVmalloc(): Allocate one page table for th machine for the kernel address spac for schedular processes.

lcr3(): Load cr3 register (Base of page

setupkvm(): Set up page dire or page table for the kernel and ~~returns address of this~~ and returns the address of that

* Before kvmalloc, our page directory w pointed by entrypgdir.
* kalloc(): returns a 4K size page from freel
* If there's a page directory entry, it creates two level page table.

Inter Process Communication (IPC)

- Cooperating processes can affect or be affected by other processes, including sharing data.
- Cooperating processes need IPC.

IPC

Shared Memory | Message Passing

Shared Memory:

- Two processes share the memory intellegently given/provided by kernel.

Message Passing:

- Kernel has to do more work.
- Two processes don't have anything in common but they request kernel to send or receive a message.
- Process is supposed to delete ~~the~~ or free the kernel resource after it is done using it.
- Processes communicate with each other with the help of OS //.

Pipe (1):

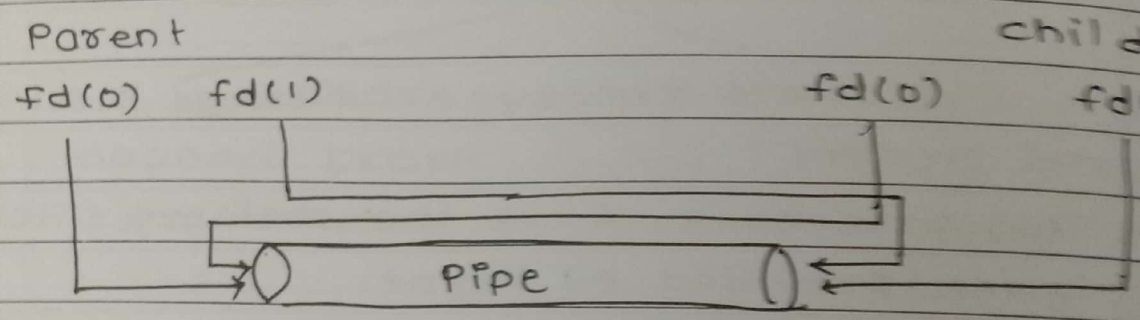
- ls -l | grep xyz
  (output of ls is input to grep)

Pipes for IPC

1. Unnamed or Ordinary pipes
2. Named pipes

Ordinary Pipes

- communication in std. producer-consumer sty
- Producer writes to write end and consume reads from read end.
- Works on FIFO style (unidirectional).



pipe() system call:

- When pipe() system call is called, it crea a notion of buffer (kernel) and two struct files and two pointer from file descriptor array of PCB gets pointed to those struc files and pipe() returns them.
- When fork is called on parent process w used the pipe() syscall, the file descripto gets copied and points to the same stru files as in above (& those struct files ha been pointing to some place in buffer).
- The above buffer becomes shared between parent and child.

* A process is required to close all unused end of pipe.

dup(): copies the given file descriptor & hence as a result of dup() we have to fds pointing to same file.

* pipe is not exactly same as i/p or o/p redirection to file.

Named Pipes
- FIFO
- Processes can create file that acts as 'pipe' & multiple processes can share file
- Bidirectional
- No parent-child relationship required & hence more powerful than unnamed pipe
- Not deted by OS automatically

* permission of Files
-rw-r--r-- (Regular File)
prw-rw-r- (Special File (piped file))

FaceLook
DATE / /

Shared Memory

* System File Shared Memory
  - Process first creates shared memory segment (shmget() function to be used)
  - POSIX: Portable OS Interface
  - shmid (Get ID of shared Memory)
  - shmat (Attach to shared Memory)
  - shmdt (Detach from —u——)

} FN used for shared memo

Message Passing

- Processes comm. with each other using send(), receive() syscalls.
- If P and Q want to communicate
  1) kernel needs to establish comm. link blw them
  2) exchange messages via send/receive.
- ftok() to generate unique key.
- mesget() creates message queue.
- msgctl() destroys message queue.

Message Passing using Naming:

- Pass message by "naming" the receiver
  1) Direct comm. with receiver
     (sender identifies receiver by its name)
  2) Indirect comm. with receiver
     (sender identifies receiver by its 'location of receipt')

Direct Communication

* Properties of link
  1) Automatically established
  2) Exactly one pair of comm. process for a pair of process
  3) Bidirectional (but can be unidirectional)

Indirect Communication

- Msgs received or sent through mailbox (just like sockets) {Unique ID for mailbox}

* Properties of Mailbox:
  1) Link established if two processes share common mb.
  2) One link for many processes.
  3) Each pair of process may share several communication links.
  4) Uni or bidirectional.

Message Passing Synchronization Issues

- Blocking / Non-blocking
- Blocking (Synchronous)
  1) Send has the sender block until msg is received
  2) receive has the receiver —∿— ~ is available
- NonBlocking (Asynchronous)
  1) Send has the sender send the msg & continue
  2) Receive has the receiver receive the msg (valid) or null.

Memory Management (Part II)

* MMU: Hardware features for MM
* OS: sets up MMU for a process, then schedules process.
* Compiler: Generates object code for a partic OS + MMU architecture
* MMU: Detects memory violations and raises interrupt → effectively passing control to

Static Linking
- All object code combined at link time and a big object code file is created (large execu files)

Static Loading: All code is loaded in memory at th time of exec.

⇒ Big executable Files, need to load functions ev if they do not execute (wastage of PM)

Dynamic linker:
At the time of loading, the program in memo the link-loader will pick up the relevant cod from the library machine code file (smal ex -able f

PLT: Procedure Linkage Table
Used to call external procedures/ functions whose address is to be resolved by the dynamic linker at runtime.

Dynamic Loading
- Load a part from ELF file only if needed during execution (small executable file)
- Delayed loading
- allocate memory if needed.

Continuous Memory Management

- Process: One continuous chunk
- Two partitions: 1? For process
  2? For OS (Usually at high addresses cuz IVT map to that location)

Strategies for finding a free chunk

6K, 17K, 16K, 40K holes. Need 15K

① **Best Fit** — Find the smallest hole, larger than process (16K)

② **Worst Fit** — Find the largest hole (40K)

③ **First Fit** — Find the first hole larger than process (17K)

External Fragmentation:

- Total unused memory space is enough to answe all the allocation requests. But memory is non-contiguous.
- Solution: Compaction.
- Compaction: Technique to collect all the free memory present in form of fragments into large chunk of memory.
  - It must update memory mana. info in PCB of each process (Time consuming)
  - Possible only if relocation + limit scheme of MMU is available.

Internal Fragmentation:

- Memory is divided into fixed partitions
- Allocate one or more chunks to a process, such that total size is >= size of process.
- Ex. each chunk is 50K.
  ① For 40K, allocate 1 chunk
  ② For 60K, — " — 2 chunks.

⇒ Leads to internal fragmentation.
- Memory wasted at 40K & 60K processes.
⇒ Solution to internal fragmentation.
- Reduce size of fixed partitions.

Paging

Job of OS
- Allocate page table for process, at time of fork/exec
  - Allocate frames to process
  - Fill in page table entries
- In PCB of each process maintain
  - Page table location
  - List of page frames allocated to this pr
- During context switch, load PTBR using PCB.

Disadvantage of Paging
- Each memory access results in two memory accesses.

Speed up Paging:
- Translation Lookaside Buffer
- Part of CPU
- Cache of page table entries
- searched in parallel for a page number.
- TLB hit: 1 memory access
  TLB miss: 2 memory access.
- All done by hardware.

Effective memory access time
Hit ratio x memory access time +
miss ratio x 2 x memory access time.

- Every process has its own page table unless you use an inverted page table.

DATE / /

Case Study (Oracle Solaris)

- Uses hash table (2 hashtables)
- One for kernel and one for all user processe
- Each hash table entry: base + span (# of pa
- Caching Levels:
  ① TLB (on CPU) ② TSB (in memory)
  ③ Page tables (in memory)

* First search is in TLB.

Swapping

Standard Swapping

- Entire process swapped in or swapped out.
- With continuous memory management.

Swapping with paging

- Some pages are "paged out" and some "paged in"
- Term "paging" refers to paging with swapping.

Activation Record:

- Local vars + parameters + return address
- One activation record for each function call which get destroyed on return.

- ebp & esp used to denote activation record

x86 Instructions:

① leave

```
mov %ebp, %esp
pop %ebp
```

② ret

```
pop %ecx
jmp %ecx
```

③ call x

```
push %eip
jmp x
```

④ endbr64

Normally a no operator (processor wasting one CPU cycle)

* More paremeters, more registers and hence more sophisticated way to do calling convention.

PAGE NO.:
DATE: / /

Segmentation & Multiple Base + Limit Pairs

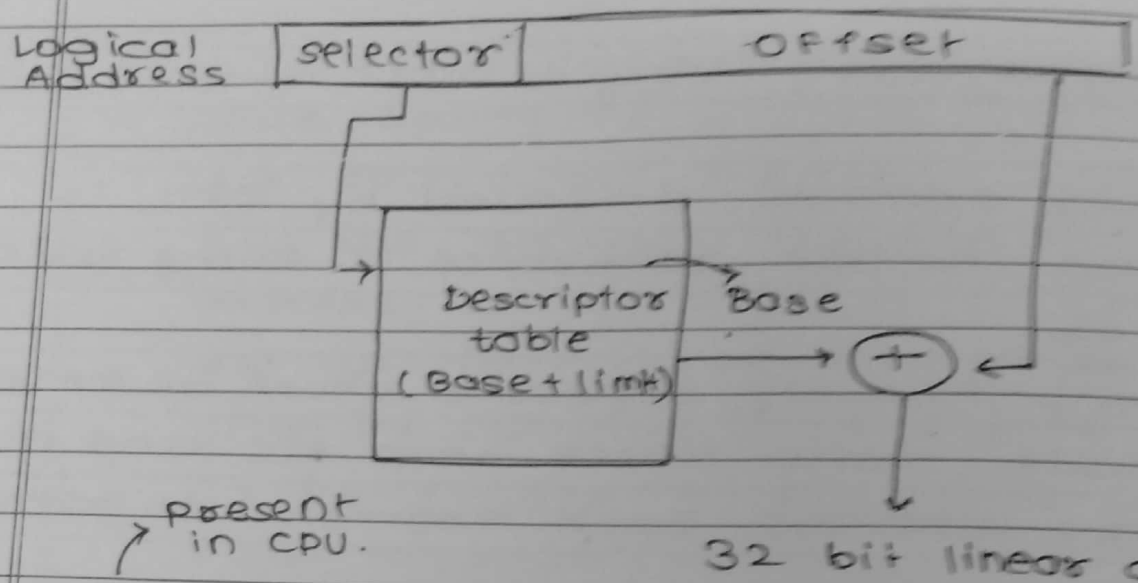
- whenever address is issued by CPU, it will also include reference to some base register.
- **compiler** Assume seperate chunk of memory for code, data, stack, heap etc and acc. calculate address. Each 'segment' at address C
- **OS** : Loads different section in different memory regions and acc. set diff base registers.

* **Imp**

- process is not a continuous chunk

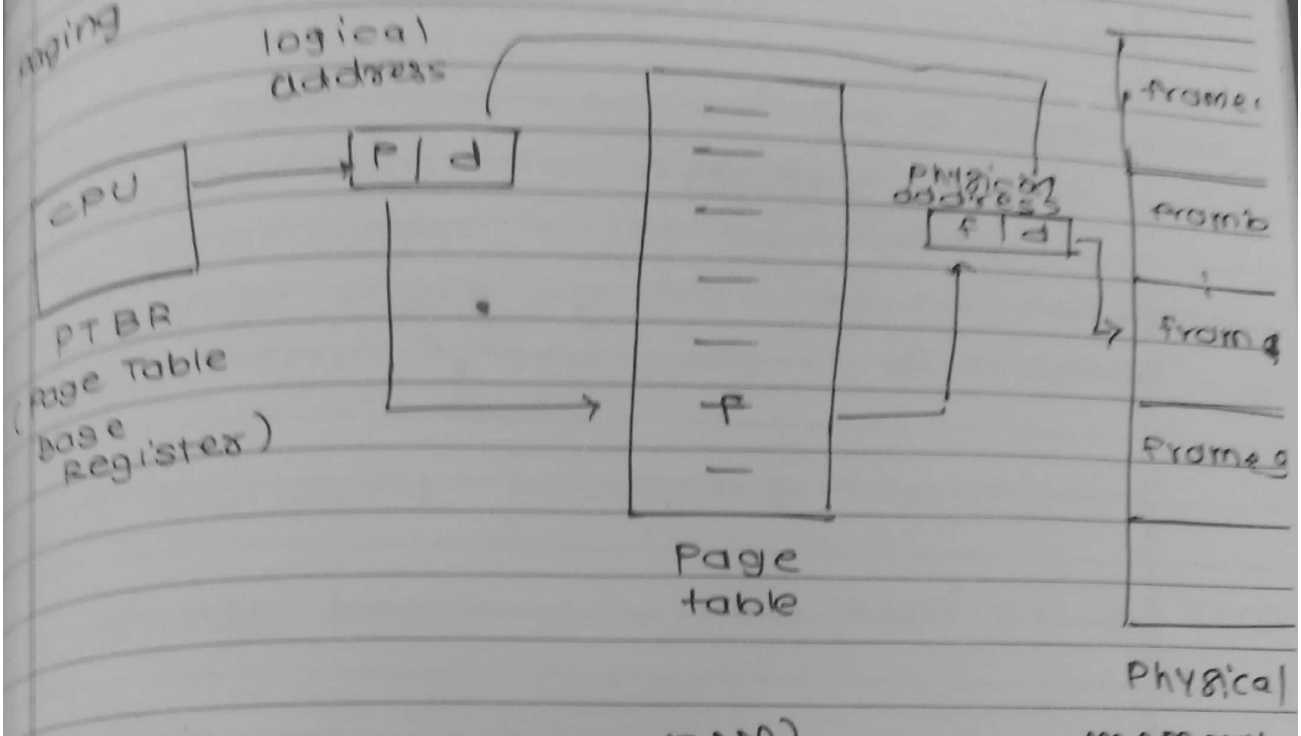
Multiple base + limit pairs, with further indirection

- Base + limit pairs can be stored in some memory location.
- One CPU register to point to the location of table in memory.
- segment registers are still in use, they give an index in this table.
- This is x86 segmentation.
- This is highly inefficient and slow as it doesn't involve MMU.



* GDTR: A register pointing to descriptor table
* Descriptor Table is in memory.
* OS stores value/ location of descriptor table in GDTR.

Paging



* Page table is in memory and set up by O PTBR. gives location of PT.

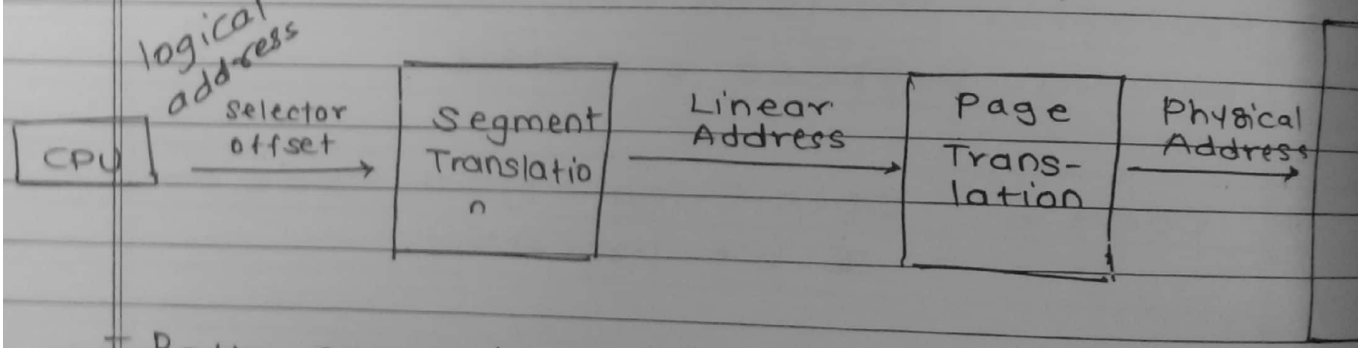
* CPU gives address in form p+d. The p is an index in page table which gives frame number. Frame number combined with offset gives value ~~of~~ from RAM.

* Page table needs to be continuous in memory.

* zero data segments in ax, DS, ES and SS.
* By default 21st bit will be disabled by BIO and if OS needs it, it will enable it.

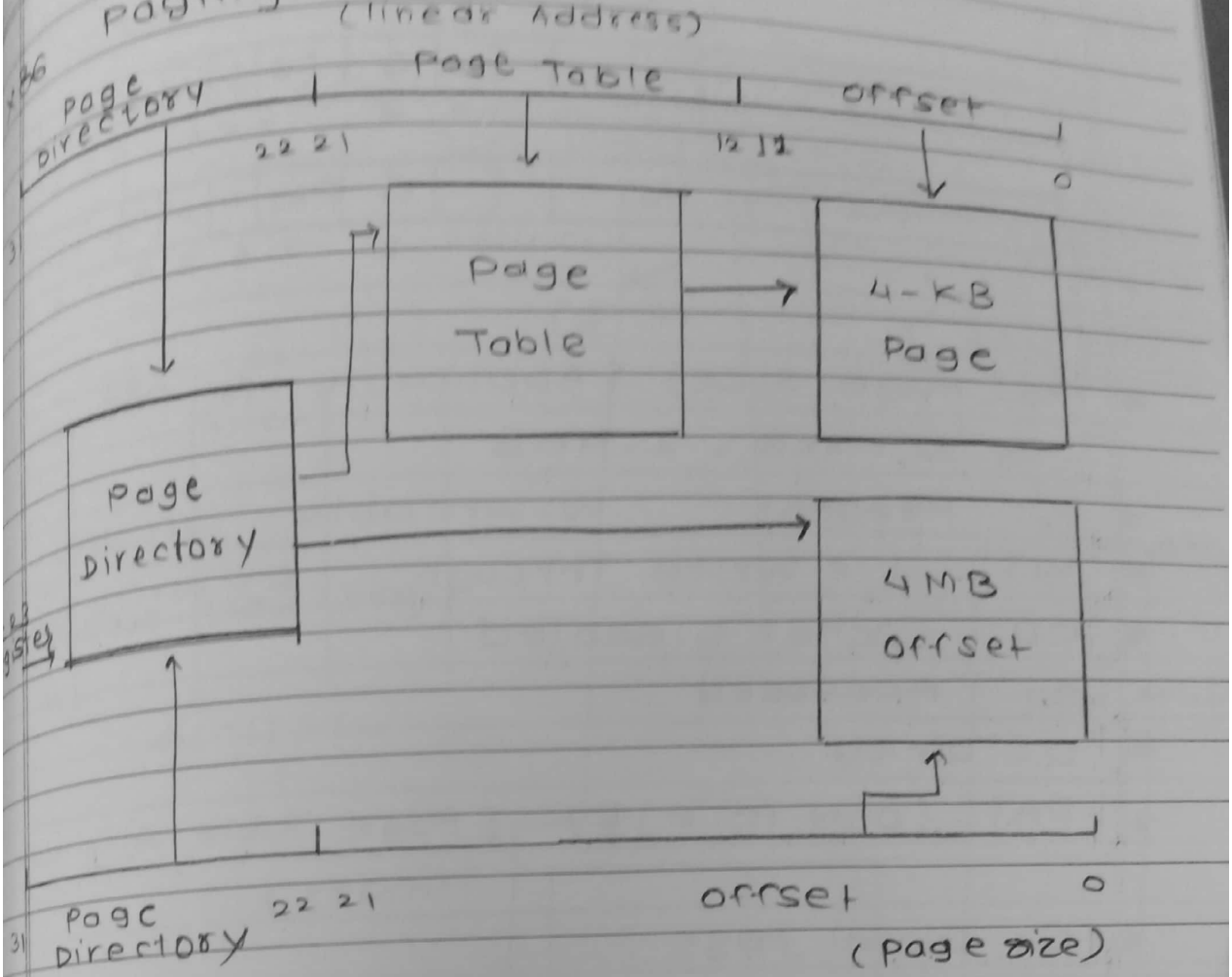
Protected Mode:
- Enables segmentation + paging
- Segment register is index into segment descriptor table. But segment : offset pair continue.
- Address translation happens using segmentation and paging.
- other segment registers need to be explicitely mentioned in instructions
  - MOV FS: $200.30

x86 address: Protected Mode Address Translation



- Both segmentation and Paging are used in protected mode x86
- x86 optionally allows one-level or two-level paging.
- Segmentation is must, paging is optional (needs to be enabled)
- Different OS use segmentation + paging in different ways.
- segment register is index into segment descriptor table.

paging

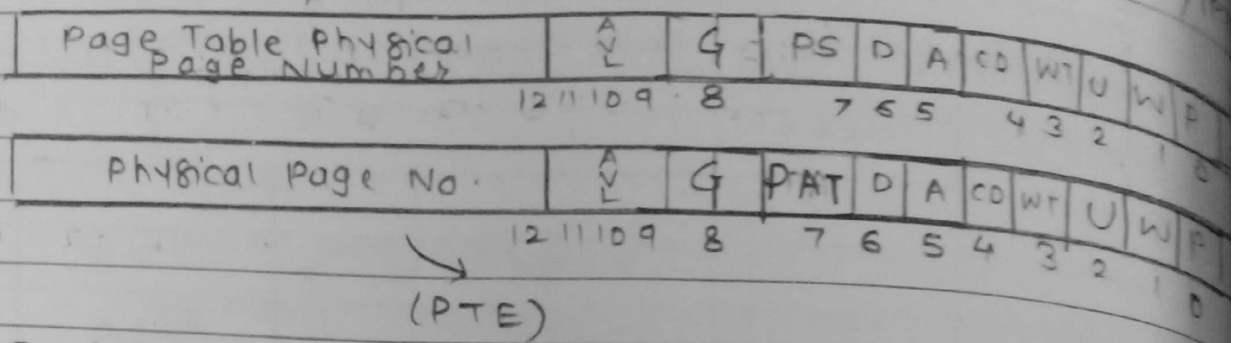


* One-level paging gives offset of 4MB
* Two-level paging — " — " — 4KB

* CR3 Register gives location of base of page directory (in case of two-level paging)

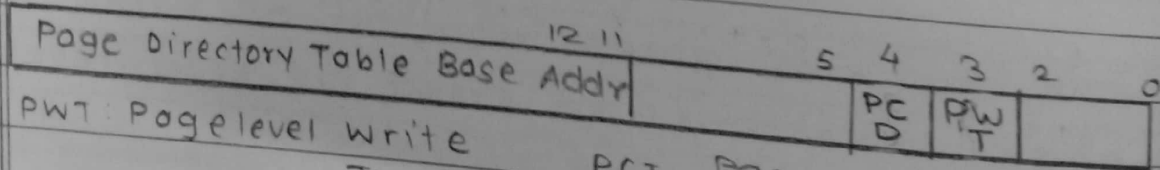
* Page Table and Page directory all are in memory.

Page Directory Entry (PDE) and Page Table Entry (PTE)



* PS (Page size) {Applicable in PDE only}
  ⇒ 0 : 4kB , 1 : 4MB
* P : Present W: Writable U : User
* WT: 1 = Write Through 0 = Write back
* CD: Cache Disabled
* A : Accessed
* D : Dirty
* PAT : (Only in PTE) {Page Table Attribute index}
* G : Global Page
* AVL : Available for system use

CR3 Register

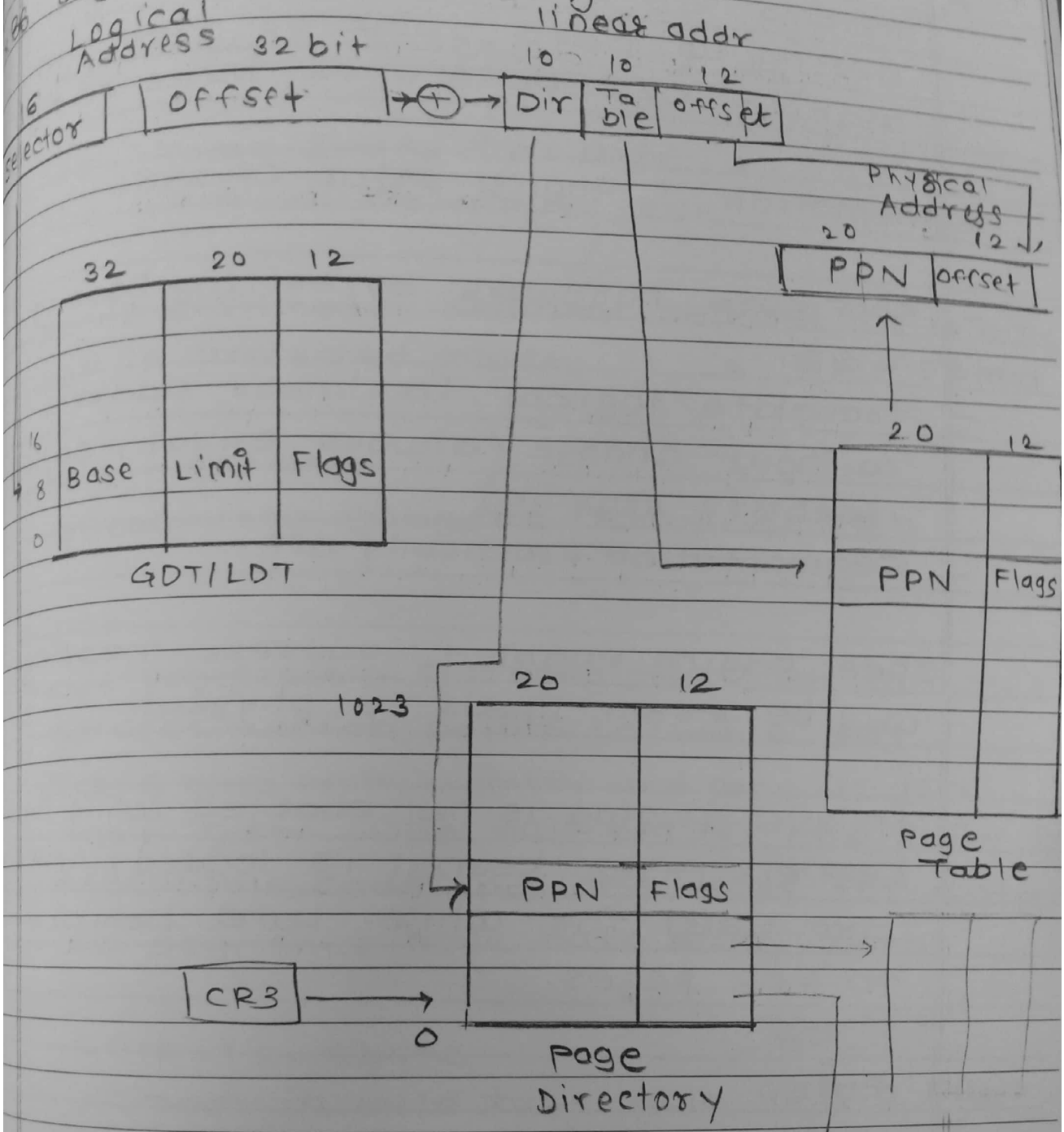


PWT : Pagelevel write Transperant
PCT Page-level Cache Disable

PWT: Page level write Transperant
PCD: Page-level Cache Disable

PAGE NO.:
DATE: / /

segmentation + Paging



selector value is implicit based on address being accessed.

Instruction: CS

Stack variable: SS

Data: DS

etc.