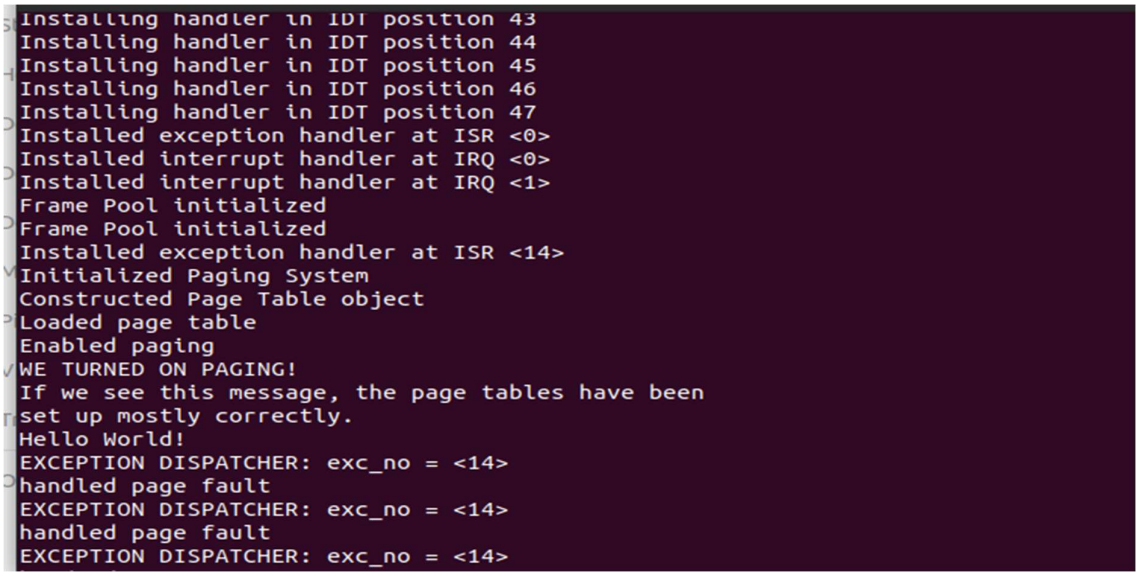
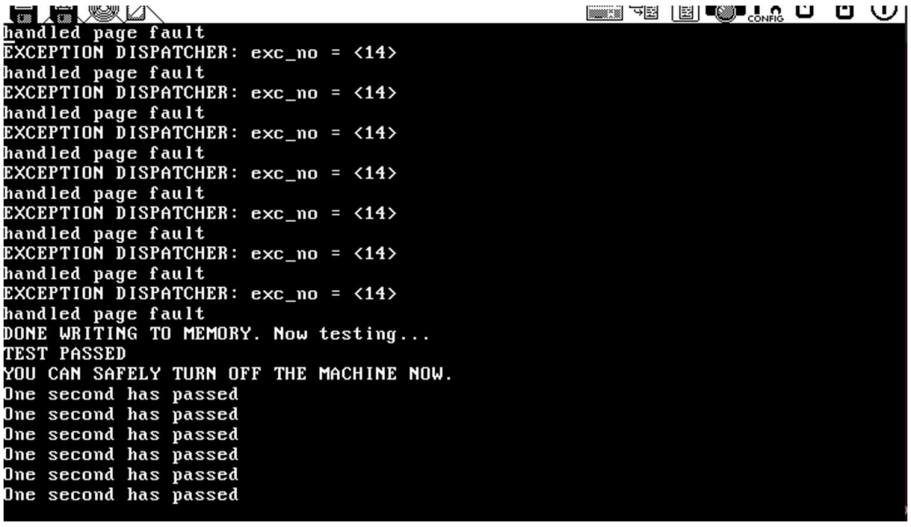
Procedure:

1) The handout is clearly studied and
2) The first 4MB is directly mapped
3) Page tables are setup correctly
4) Paging is enabled
5) The CR0, CR2, CR3 registers are properly utilised
6) The page table entries are correctly masked for address space beyond 4MB
7) The testing is done in new development environment using the following commands
8) make, ./copykernel.sh, bochs -f bochsrc.bxrc
9) The following photo shows the result of the output.
10) The detailed paper work is attached at the end of this document.
11) The github link is https://github.tamu.edu/kausht14/OS_MP3

Result:





① Memory - 4MB — directly mapped
& directly allocate in the page table

② Beyond 4MB needs implementation

③ only one process
First 4MB of each address space
→ maps to 4MB of physical memory

④ Create a page table object (done) to access the
directory → page table pages → memory frames

⑤ ① Store the page table object on the stack
→ ② create page table before paging enabled
to prevent page fault

⑥ Then load the page table by using current variable
to store current table

⑦ Once loaded enable paging by using the
CR0 register → paging_bw.asm & paging_bw.h

⑧ handling faults later

⑨ get_frames & release frames is available

10) Kernel port - 2MB & 4MB & is in direct mapped memory

11) process-mem-port above 4MB & freely mapped

→ define init paging & store into private variables values passed

→ Once init done, kernel sets up first page table object

→ Give a frame port to directory

↓

we need to configure port first

→ After init, port init, load the page table

do Reg * CR3 = & page table-directory

→ make sure during context switch the new page table page is loaded

→ Switch to paging using enable_paging() by setting CR0 register

→ Only do handle_fault() after initialisation of page table is done.

2 00111001 3

0010 1001

0011 0

→ Below 4MB → directly mapped

→ whenever above 4MB, page fault occurs &

page-fault handler should

1. get_free_frame from free-frame pool
2. Should allocate this to process
3. Update the page entry
4. Update page table entry } if multiple references of page faults
5. CPU again fetches the instruction

→ Store the page directory, pagetable pages & management info for the process frame pool in kernel pool

→ management info for kernel is however stored in kernel pool itself because directly mapped (0-4MB)

→ Dont forget CR0, CR3 CR4

CR0 - page enabling

CR3 - page directory table register

Structure of PT entry can be used for setting use/dirty bits if allowed

31 ... 12	11 ... 9	8 ... 7	6	5	4 ... 3	2	1	0
Page frame	Avail	Reserved	D	A	Reserved	U/S	R/W	Present

U/S - User or Supervisor level

R/W - Read or write

Present - use bit

0 1 0 1

→ A page fault triggers exception "14"

→ pushes a word with the exception error code onto the stack

→ REGS type argument tells about the exception

→ Lower 3 bits of word pushed onto the stack

Value	2	1	0
0	Kernel	read	page not present
1	user	write	protection fault

↳ error-code folder to check

0 to 1024

→ 32-bit address of the address that caused the page-fault → Store in CR2 / ✓

and read using CR_2 → read_cr2()

① Set the VMS address Valid bit to high

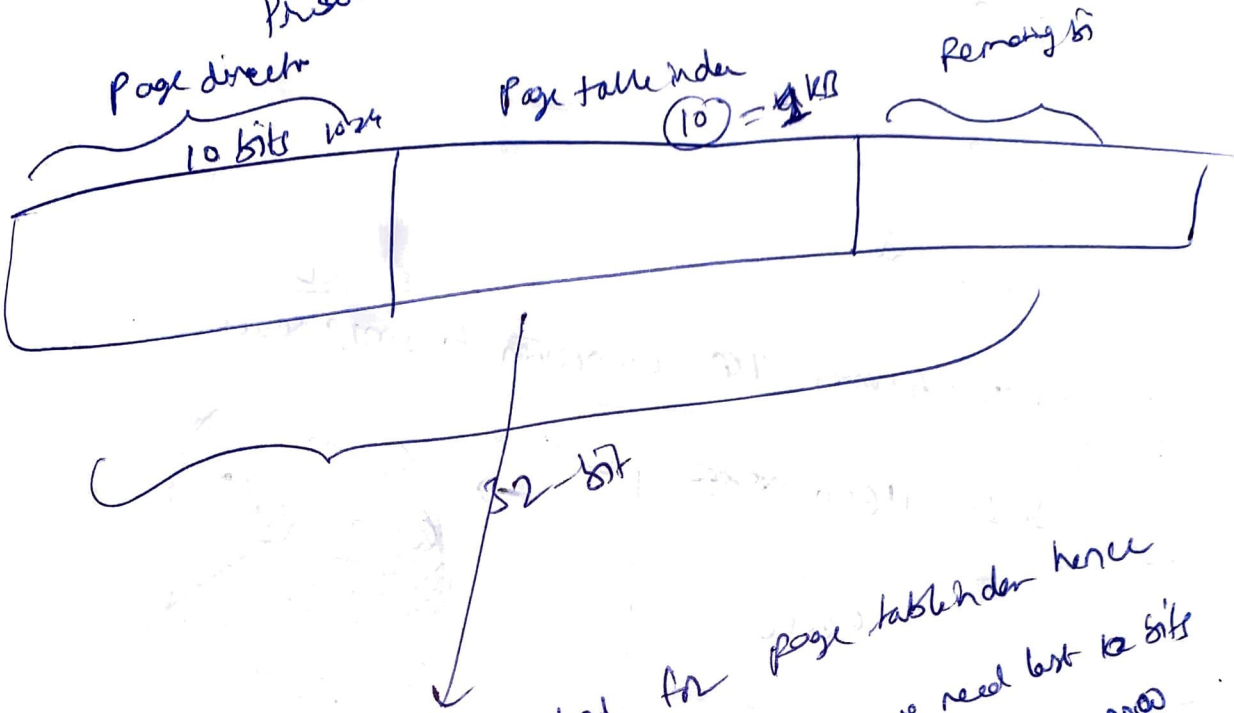
② mark the invalid bit initially for above VMS - 32MB

③ Then page-faults occur, puts in page table & then sets to Valid, mark the page as present

return from exception

for UMB

Present bit – 0 → not present



only this needed for page table index hence we can shift by 12 bits initially & we need last 12 bits

20