MEMORY MANAGEMENT IN OS → Long Session

SESSION 6

LINK IN THE DESCRIPTION FOR ALL SESSIONS

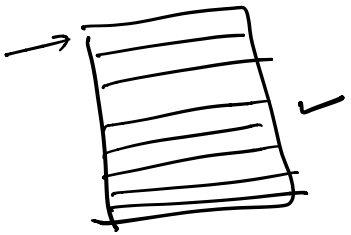
PREVIOUS SESSION

- Process synchronization
- Race condition
- Critical section problem
- Software and hardware based solutions
- Semaphore and mutex
- Producer consumer problem
- Reader writer problem

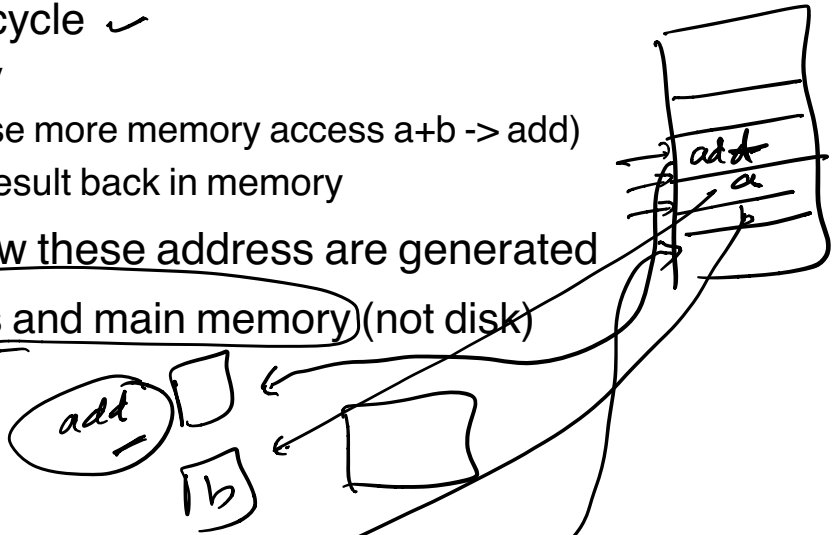
INTERVIEW QUESTIONS

- Difference between 32-bit and 64-bit system
- Logical memory vs Physical memory
- Normal paging vs demand paging vs pure demand paging
- Segmentation and paging
- Virtual memory and its advantage
- Paging vs segmentation
- Physical address vs virtual address
- Inverted paging

MAIN MEMORY

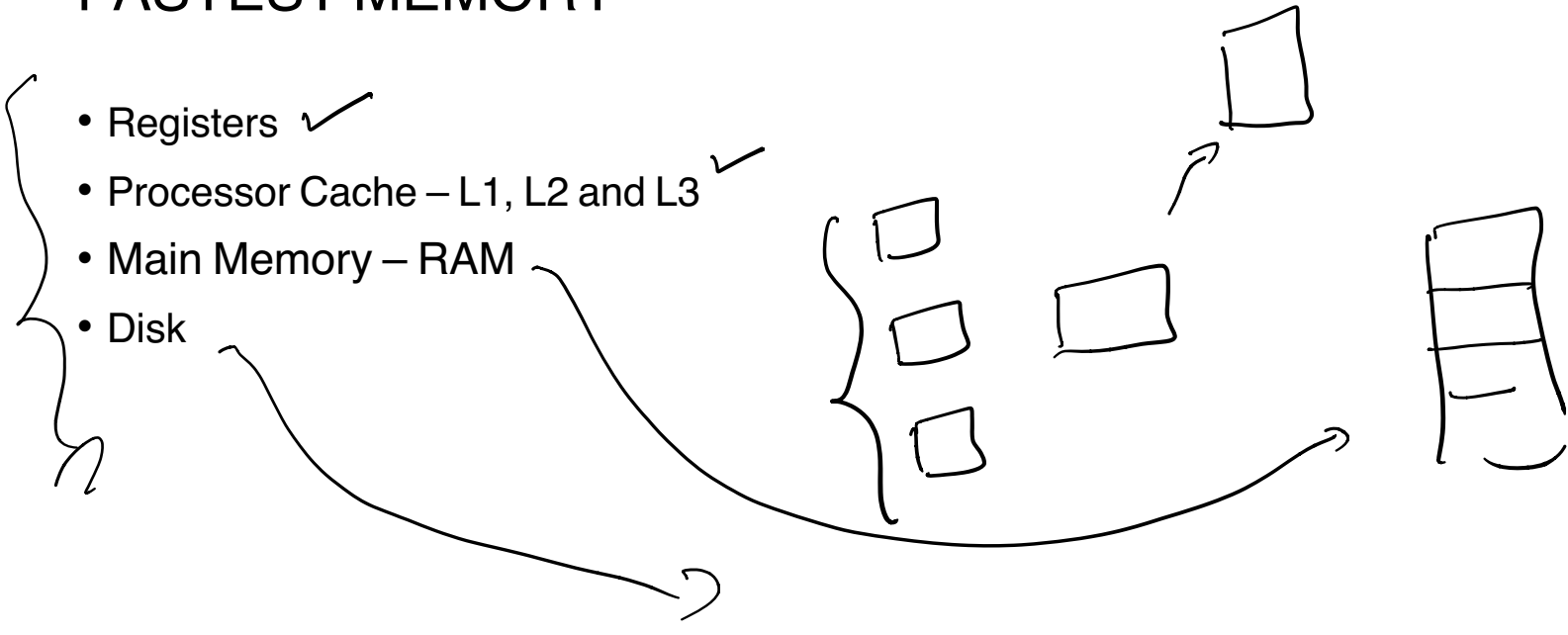


- Memory consists of a large array of words or bytes, each with its own address.
- Typical instruction-execution cycle
  - Fetch instruction from memory
  - Instruction decoding (may cause more memory access a+b -> add)
  - Execute instruction and store result back in memory
- Memory unit doesn't know how these address are generated
- CPU can only access registers and main memory (not disk)



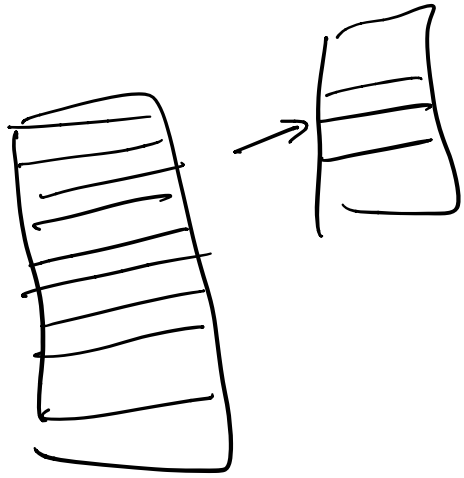
FASTEST MEMORY

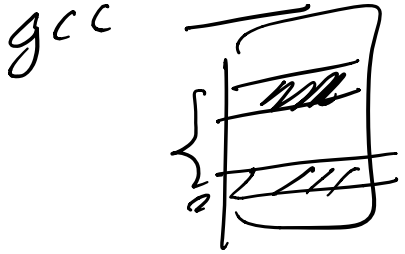
- Registers
- Processor Cache – L1, L2 and L3
- Main Memory – RAM
- Disk



ADDRESS BINDING

- Compile time
  - Starting address is known
  - Starting location changes ?
- Load time
  - Address not known at compile time
  - Final binding is delayed till load time
  - Relocatable code
- Execution time
  - If process can be moved during execution?
  - Binding is delayed till run time
  - Used in most modern OS

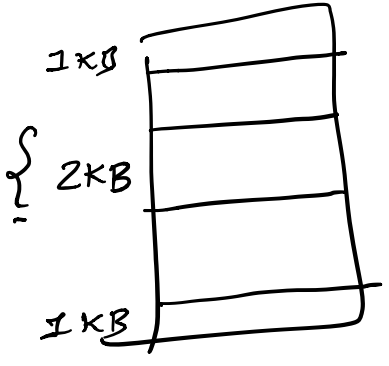




CONTIGUOUS MEMORY ALLOCATION

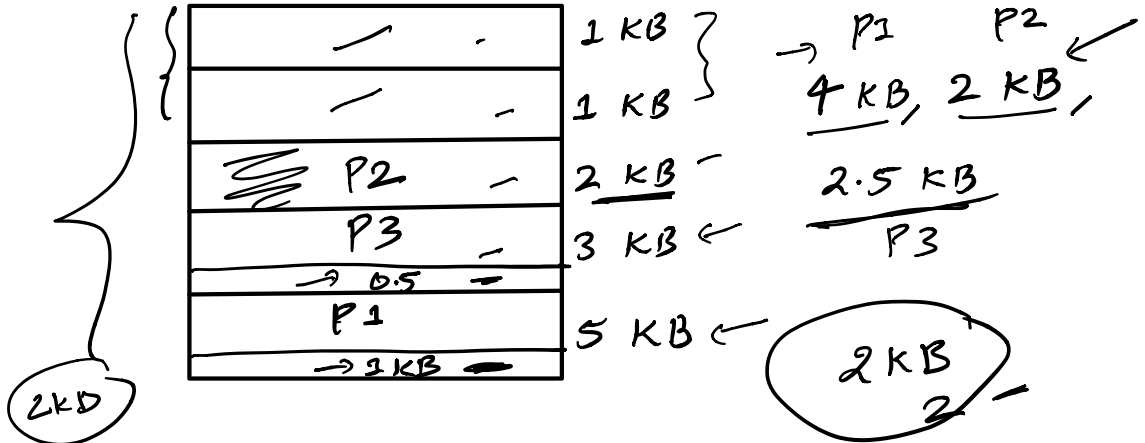
- Traditional approach
- Allocate contiguous memory blocks for a process
- Two approaches
  - Fixed partitioning
  - Variable partitioning

P_1 2KB



FIXED PARTITIONING

- Memory is divided into partitions of fixed size which cannot change
- Each partition can only contain one process
- Find a suitable partition and allocate it



DISADVANTAGE

- Degree of multiprogramming bounded by number of partitions
- Size of process is bounded by size of largest partition
- Suffers from internal fragmentation

2 KB, 6 KB

VARIABLE PARTITION METHOD

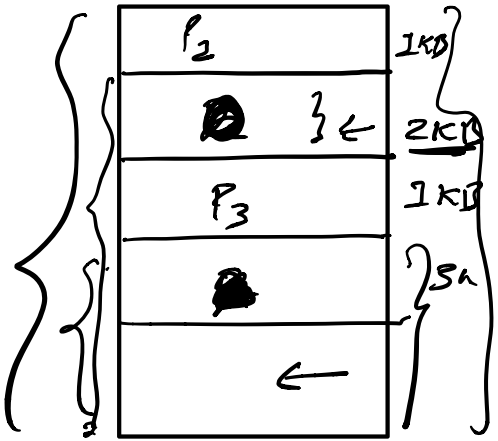
- Partitions are created dynamically as process are loaded
- Free partitions -> holes
- Initially, one hole whose size is of main memory
- OS keeps tracks of these holes
- If process size > selected hole, split hole into two
  - Give one to process
  - Other is returned to the OS
- Process terminates, hole is released.
- Two adjacent holes are merged

$P_1 \rightarrow 1$ KB

$P_2 \rightarrow 2$ KB

$P_3 \rightarrow 1$ KB

$P_4 \rightarrow 3$ KB



10 KB

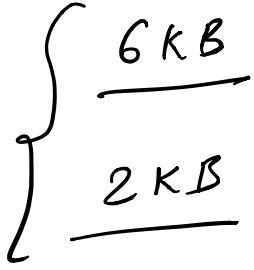
7KB

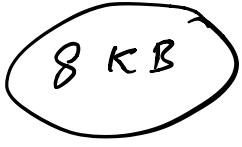
Hole: ~~10 KB~~ ~~5 KB~~

- No internal fragmentation
- Degree of multiprogramming

Disadvantage

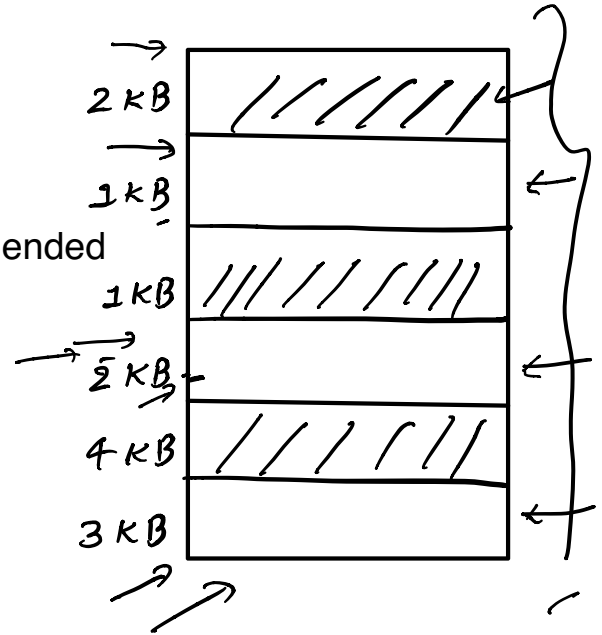
- Size of the process is limited by size of biggest hole
- Leads to external fragmentation





ALGORITHMS FOR VARIABLE PARTITIONING

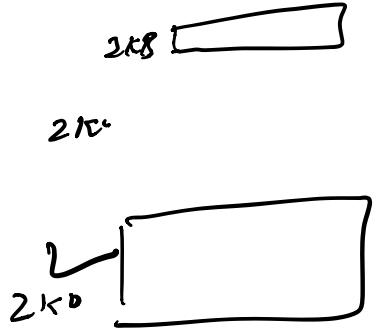
- FIRST FIT
  - Scan from beginning
  - Allocate the first hole that is big enough
- Next Fit
  - Scan from location at which previous searched ended
  - Allocate the first hole that is big enough
- BEST FIT
  - Allocate the smallest hole that is big enough
  - Produces smallest leftover hole



Selected ≥ process Size

P → 2 KB

- WORST FIT
  - Allocate the largest hole that is big enough
  - Produces largest leftover hole
  - Leftover hole may be more useful than best-fit created hole



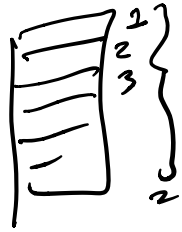
Example - https://youtu.be/fz9v-SZt3cQ

LOGICAL VS PHYSICAL ADDRESS

PCB

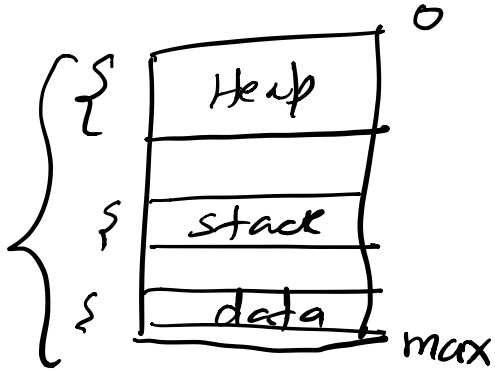
Program Counter

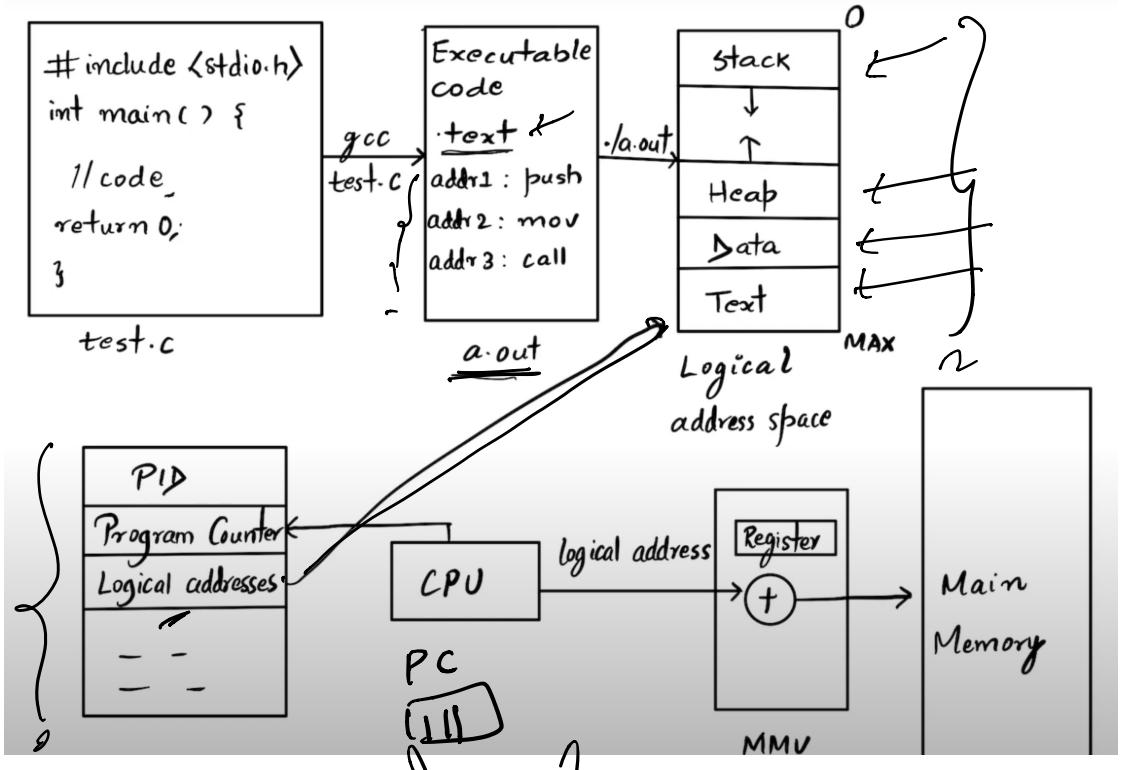
- Address which the CPU sees -> logical address
- Address contained in the program counter
- Address that exists in main memory is called physical address
- User programs never see the physical address, they only see logical address



addr = malloc(int)

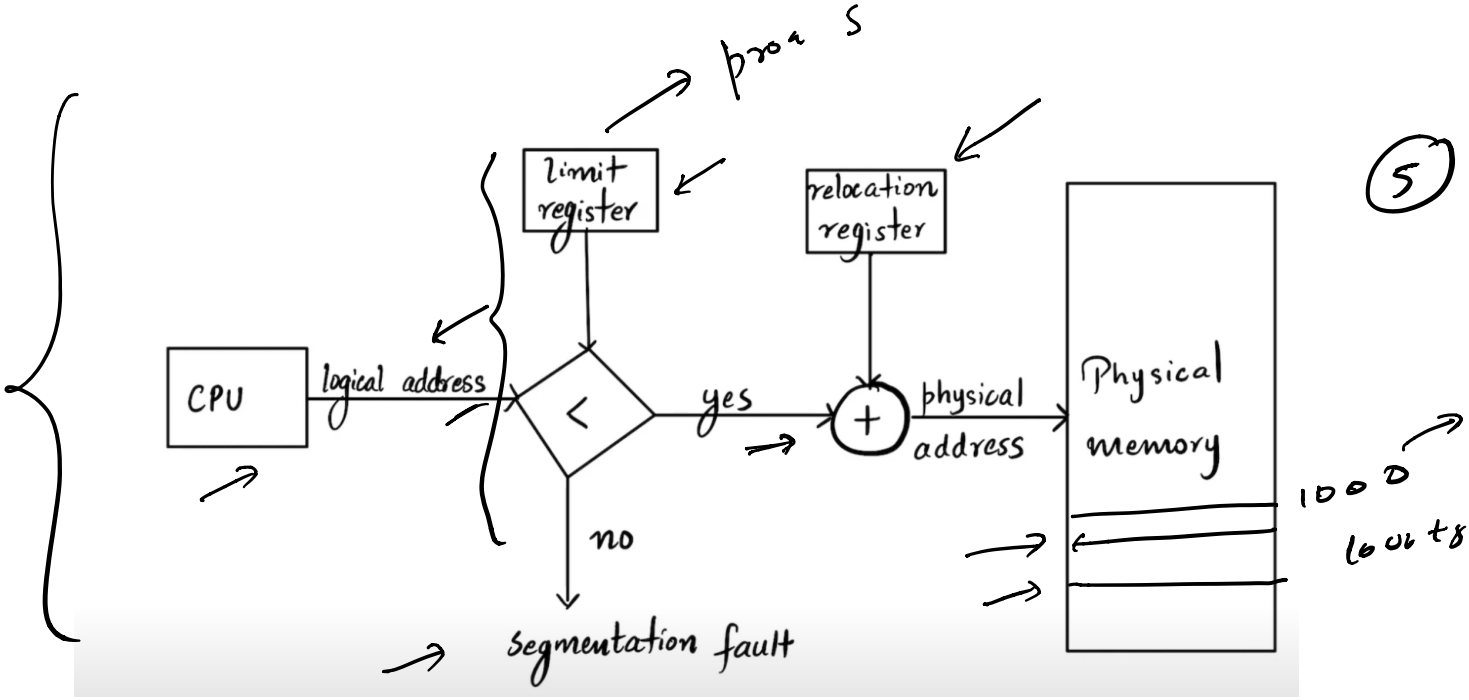
Print("%p",&addr) -> logical or physical ?




#include <stdio.h>
int main() {
// code
return 0;
}
test.c
gcc
test.c
Executable code
.text
addr1 : push
addr2 : mov
addr3 : call
a.out
./a.out
Stack
Heap
Data
Text
0
MAX
Logical address space
PID
Program Counter
Logical addresses
CPU
logical address
Register
+
Main Memory
PC
MMU

MEMORY MAPPING AND PROTECTION IN CONTIGUOUS MEMORY ALLOCATION

- The logical address need to be converted into physical address for instruction execution
- Also, one process should not be able to access other process memory
- Use limit register and relocation register
- Limit register = size of the process
- Relocation register = Starting physical address of the process

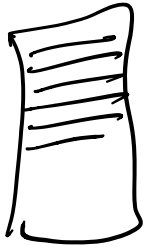

proc 5
limit register
relocation register
5
CPU
logical address
<
yes
+
physical address
Physical memory
1000
1600
no
segmentation fault

NONCONTIGUOUS MEMORY ALLOCATION TECHNIQUES

- Limitation with contiguous memory allocation
  - Internal fragmentation – Fixed partition
  - External fragmentation – Variable partition
- Use compaction to solve the external fragmentation
- But compaction is very expensive because of copying operation (I/O)
- Other solutions?

- Allow process to have noncontiguous physical address space
- Break down logical address of a process into chunks and allocate them
- Noncontiguous memory management techniques
  - Paging
  - Segmentation
  - Paging + Segmentation

PAGING TECHNIQUE

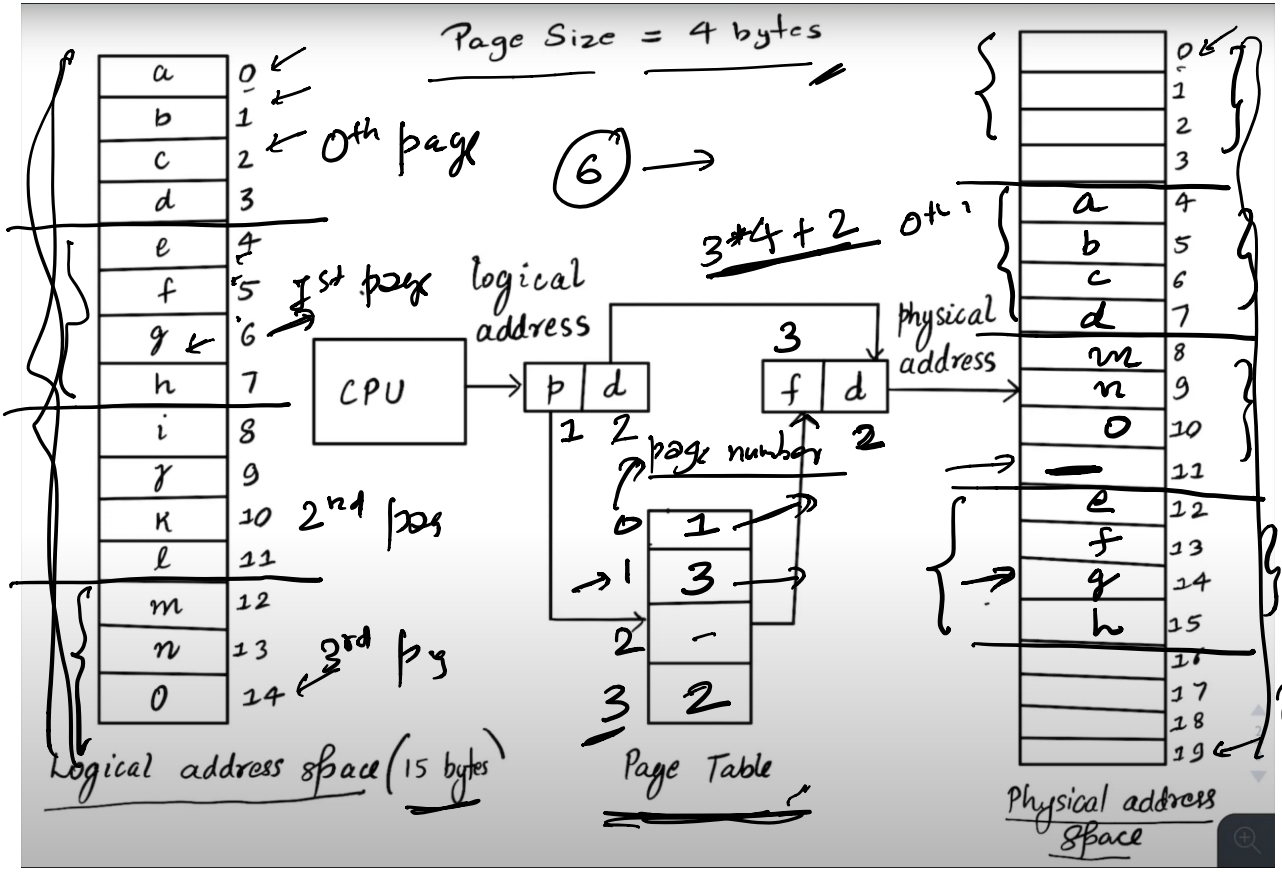


- Assume byte addressable memory
- Each byte is a unique address

- Break down the logical memory into partitions called pages
- Break down the physical memory into partitions called frames
- Page size = frame size
- Use page tables to map logical address to physical address
- Note: In simple paging, all the pages are bought in memory before process starts executing.

N pages

- If there are N pages, then at least N frames must be available for process to run. See demand paging
- Logical address – page number + page offset
- MMU combines page table mapping with page offset to arrive at physical address


Page Size = 4 bytes
a 0
b 1
c 2
d 3
e 4
f 5
g 6
h 7
i 8
j 9
k 10
l 11
m 12
n 13
o 14
0th page
1st page
2nd page
3rd pg
6
3*4+2
CPU
logical address
p d
1 2
3
physical address
f d
page number
2
0 1
1 3
2 -
3 2
Page Table
Logical address space (15 bytes)
Physical address space
0 1 2 3
a 4
b 5
c 6
d 7
m 8
n 9
o 10
- 11
e 12
f 13
g 14
h 15
16 17 18 19
fra 0
fra 1
fra 2
fra 3
fra 4

3*4 + 2

- Page number = logical address / page size
- Page offset = logical address % page size
- Can be simplified if page size is power of 2
- Number of pages = logical memory size / page size
- Number of frames = physical memory size / frame size
- Frame size = page size
- Avoids external fragmentation
- Some internal fragmentation (Fixed size partitioning)
- Same as using a table of base/relocation registers

$\frac{6}{4} = 1$

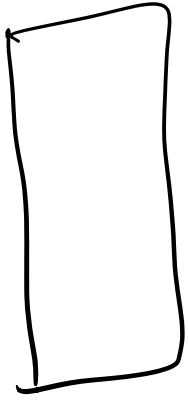
6 / 4 = 2

$\frac{15}{4} = 3..$

$\frac{20}{4} = 5$ 4

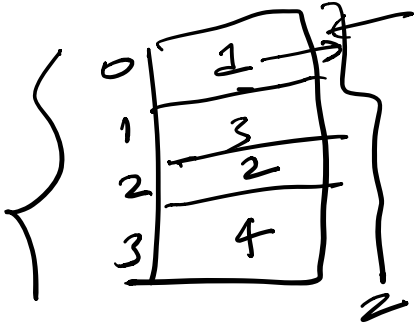
CLEARLY UNDERSTAND

- PAGE
- FRAME
- PAGE NUMBER
- FRAME NUMBER
- PAGE SIZE
- FRAME SIZE
- LOGICAL ADDRESS SPACE
- PHYSICAL ADDRESS SPACE
- PAGE TABLE
- PAGE TABLE SIZE

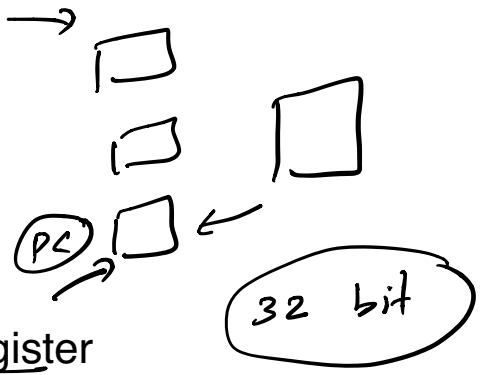


fra

4*



32-BIT VS 64-BIT SYSTEMS



- 32-bit -> CPU has 32 bit registers
- 64-bit -> CPU has 64 bit registers

- Logical address is stored in program counter register
- 32 bit -> max logical address size -> 32 bit

- Maximum address generated with 32 bit = 2^32 = 4GB (binary basics)
- You cannot run a 5GB game on 32-bit processor system

34

32 bit → 2^{32}

PAGE SIZE AS POWER OF 2

- Page size = 2^n
- Logical address space = 2^m (m-bit system)

Binary basics:

- N bits -> 2^N different values from 0 to 2^(N-1)
- 2^N values -> N bits

3 bit

8 → 3

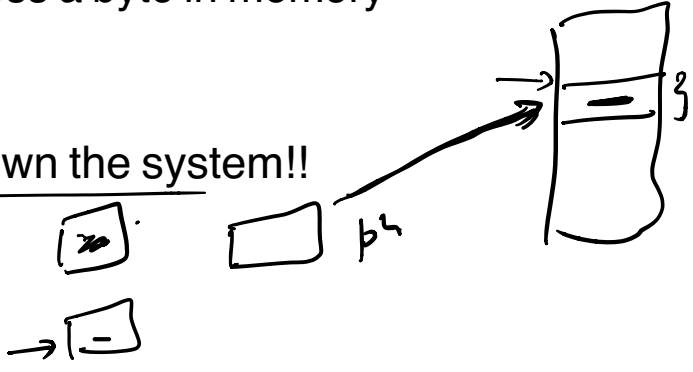
0 - 7 → 8

$\log_2 8 \rightarrow 3$

- Total pages = 2^m / 2^n = 2^(m-n) pages = 0 to 2^(m-n)-1
- How many bits you need to represent 2^(m-n) page numbers?
- (m-n) bits to represent page number
- Logical address = page number + page offset
- m = (m-n) + page offset -> page offset size = n bits
- Page offset can vary from 0 to 2^N-1
- Similar approach cab be followed for frames

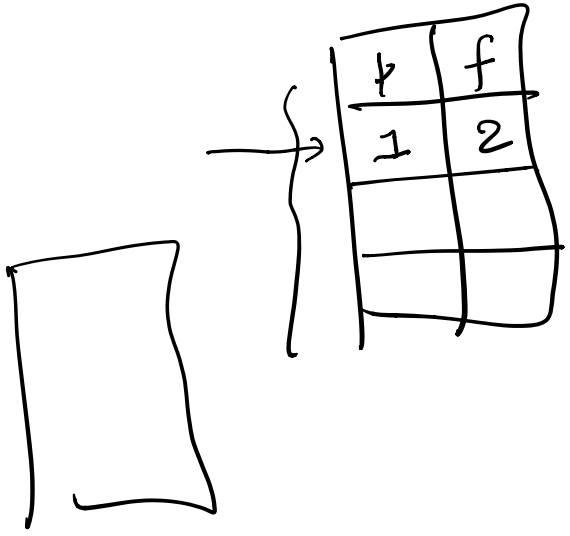
ISSUE WITH KEEPING PAGE TABLE IN MAIN MEMORY

- The page table is kept in main memory, and a points to the page table. Changing page tables requires changing only this one register, substantially reducing context-switch time
- Threads vs Process context switch time (Thread lecture)
- Two memory access required to access a byte in memory
  - Page table access
  - Access the required byte
- Doing this for every byte will slow down the system!!



TRANSLATION LOOK-ASIDE BUFFER (TLB)

- A small, expensive cache that sits close to CPU
- Key, value data structure
- Key -> page number
- Value -> Frame number
- Total entries = 64-1024
- TLB hit and TLB miss



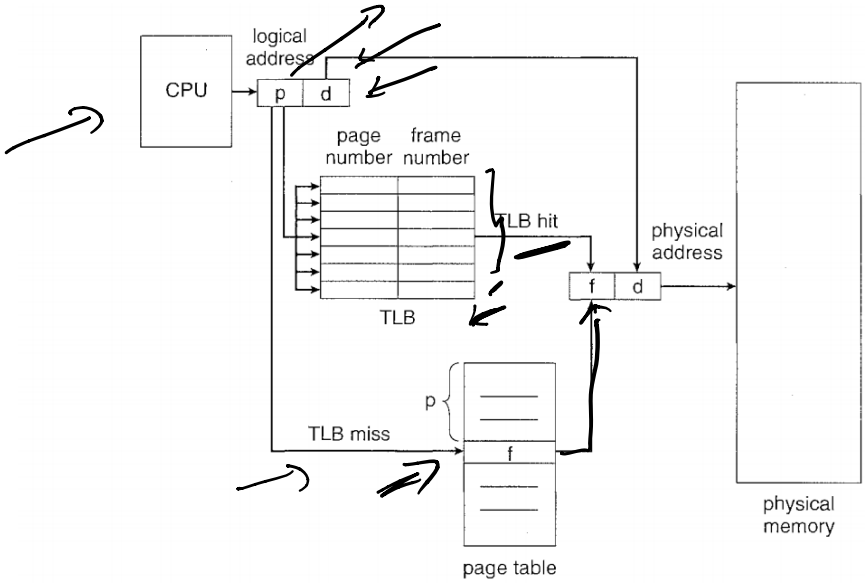
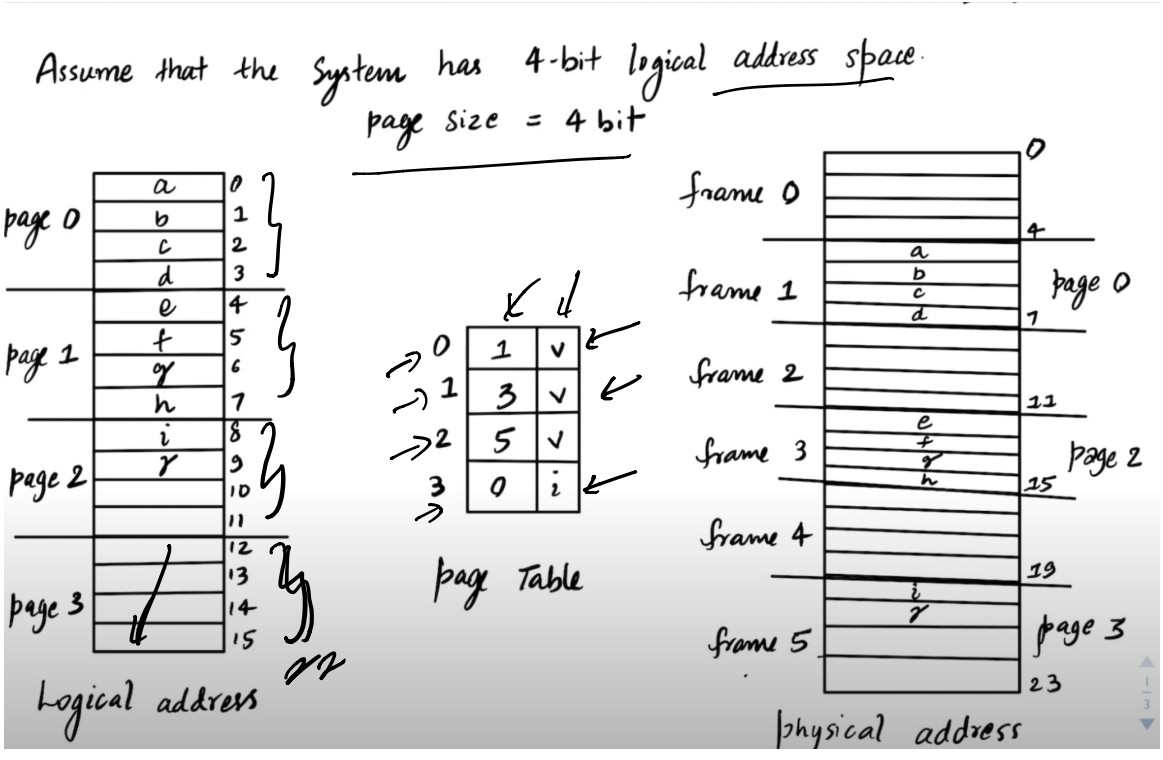


Figure 8.11 Paging hardware with TLB.

MEMORY PROTECTION IN PAGING

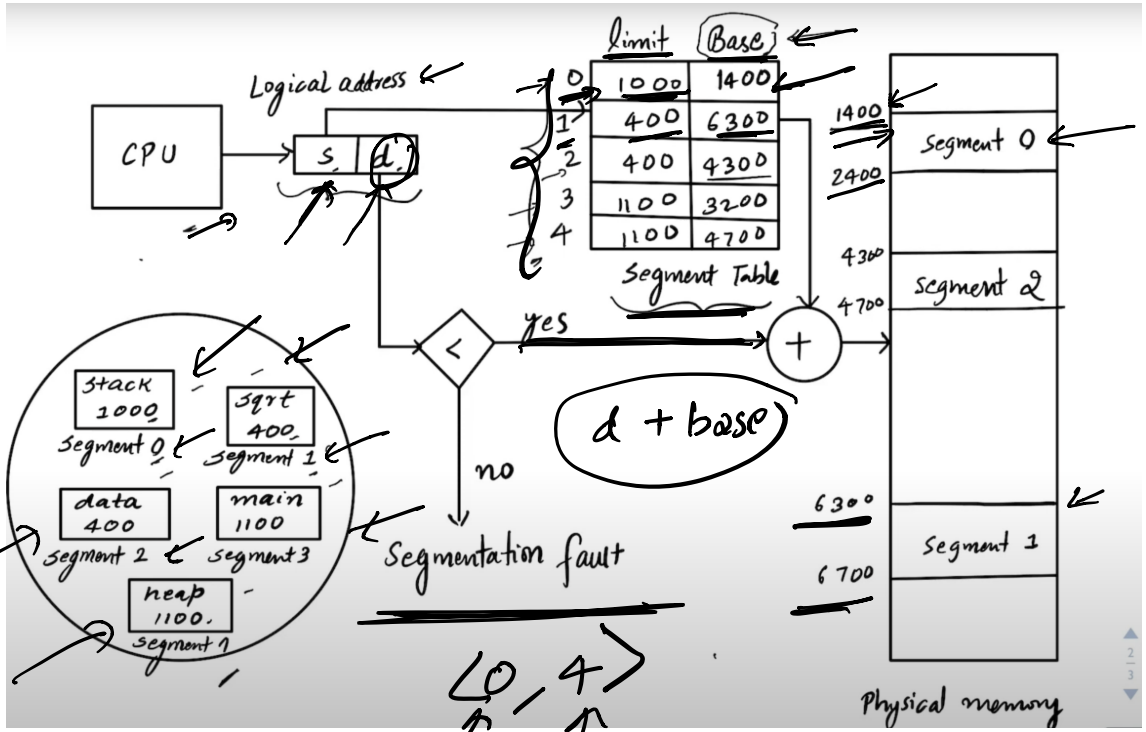
- Protection bits can be associated with along side frame number in page table
- Valid-invalid bit -> page exists in logical memory or not
- Dirty bit -> Page has been modified or not
- Cache enabled/disabled -> where caching is enabled for page or not
- Reference bit -> whether page was referenced in last clock cycle


Assume that the System has 4-bit logical address space.
page size = 4 bit
page 0
page 1
page 2
page 3
a
b
c
d
e
f
g
h
i
j
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
Logical address
page Table
frame 0
frame 1
frame 2
frame 3
frame 4
frame 5
page 0
page 2
page 3
Physical address

SEGMENTATION TECHNIQUE

- Users views a program collection of logical segments
- C program
  - Code segment (main(), sqrt())
  - Data segment
  - Heap segment
  - Stack segment
  - Segment for C library
- Segmentation is a memory management technique that supports the user view of memory.

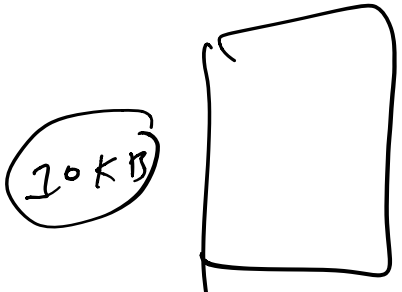
- Logical address -> Collection of segments
- Segment name and length
- Logical address = segment name + offset


Logical address
CPU
s
d
limit
Base
0 1000 1400
1 400 6300
2 400 4300
3 1100 3200
4 1100 4700
Segment Table
yes
no
d + base
Segmentation fault
stack 1000 Segment 0
sqrt 400 Segment 1
data 400 Segment 2
main 1100 Segment 3
heap 1100 Segment 4
1400
Segment 0
2400
4300
Segment 2
4700
6300
Segment 1
6700
Physical memory
<0, 4>
1400 + 4

STRUCTURE OF PAGE TABLE

Hierarchical Paging

- Logical address space = 2^32 or 2^64
- Page size = 4KB or 8KB
- Page table size itself can be huge and will need to be broken down
- Divide the page table into in smaller pieces



TWO-LEVEL PAGING ALGORITHM

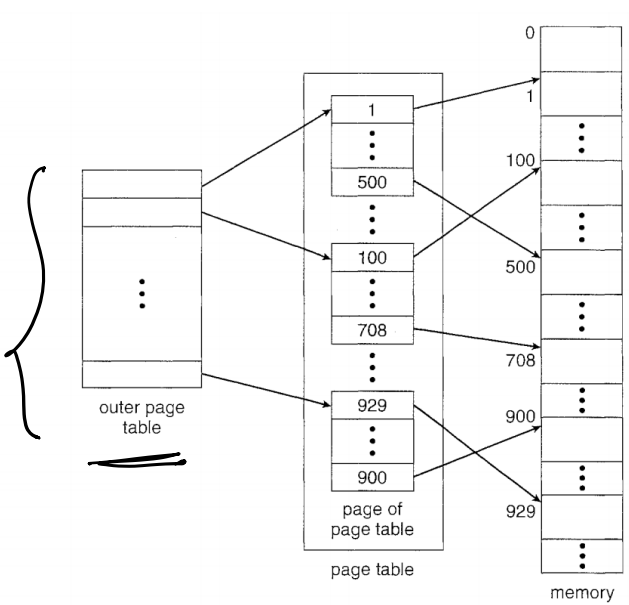
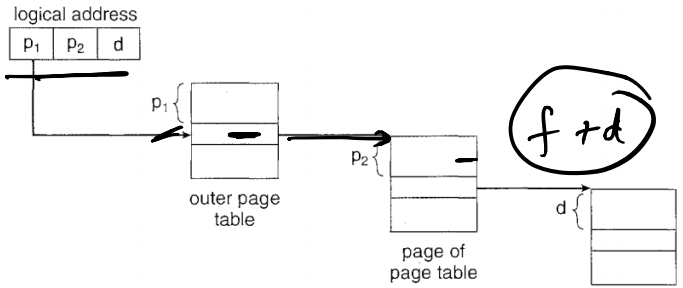
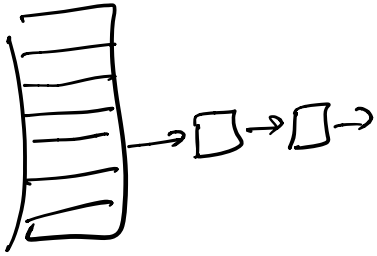


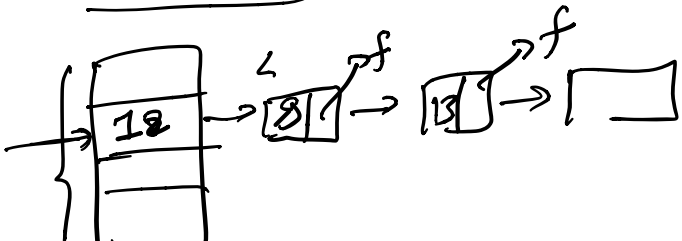
Figure 8.14 A two-level page-table scheme.



Hashed Page Tables

- Hashing using chaining concept
- Page number is hashed
- Each node in link list has following
  - Page number
  - Frame number
  - Pointer to next element
- Page table implementation as an array need contiguous memory

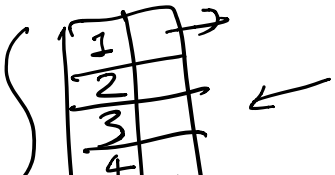




Inverted Page Tables

- Each process has its own page table
- Each page table may consist of million of entries
- These tables may consume large amounts of physical memory just to keep track of how other physical memory is being used.

- In inverted page tables, we have one entry for each frame.
- Size of inverted page tables depend on the number of frames
- We need only one table now



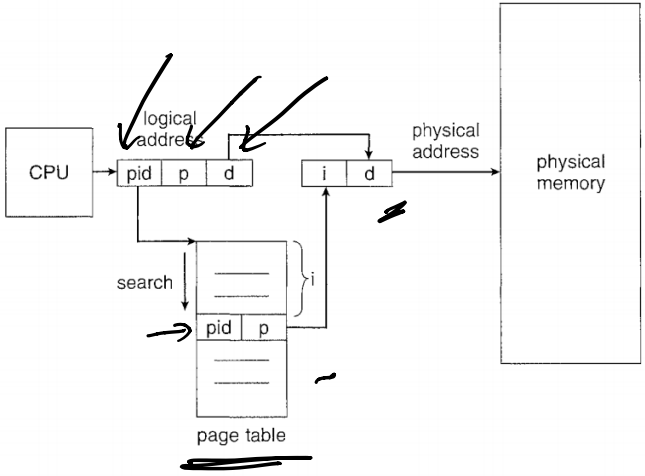
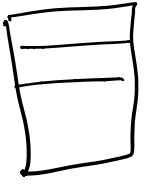


Figure 8.17 Inverted page table.

<process-id, page-number, offset>.

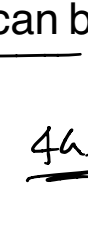
VIRTUAL MEMORY MANAGEMENT TECHNIQUES

- Normal paging – all instructions should be in memory before program can start
- Do we really need to load the whole program at once?
  - Error handling code are used very less
  - Arrays -> allocated more memory than we use
  - Certain functions are used very less
- Even if entire program is needed, it will not be needed at the same time.
- What is the maximum size of the process in simple paging?



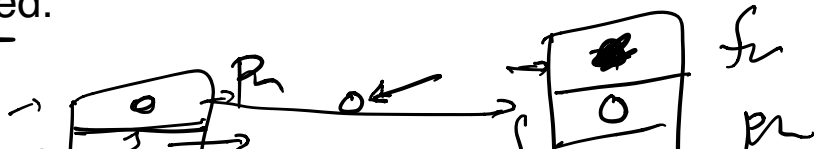
VIRTUAL MEMORY, VIRTUAL ADDRESS SPACE

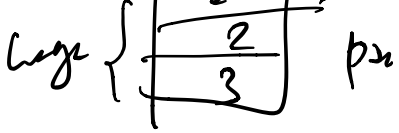
- Virtual memory management technique -> size of the process can be larger than size of main memory
- Disk is also used to store pages of the process.
- Virtual address space -> logical address space
- Virtual memory = concept of logical memory + using disk for storing pages
- When we say virtual, we mean that disk is being used for storing pages.
- Logical address is called virtual address as they are also not real.



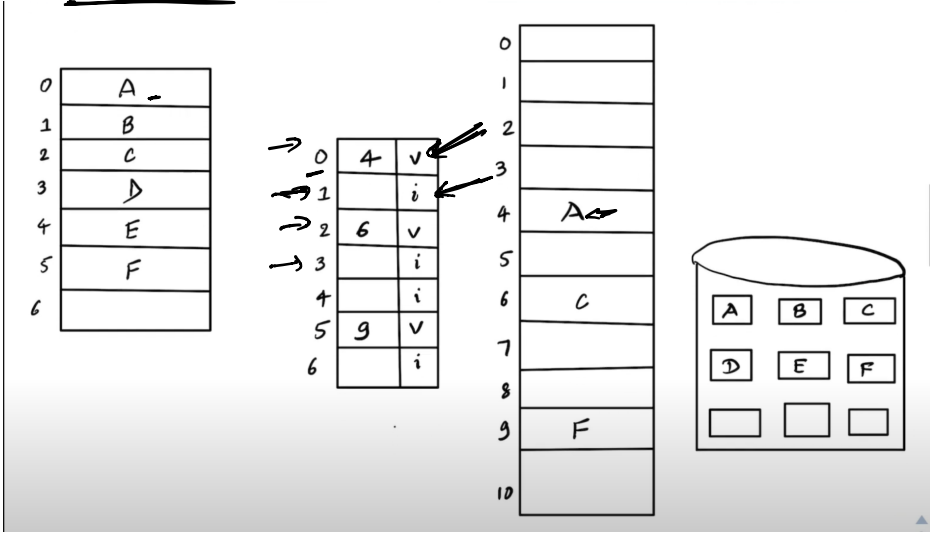
DEMAND PAGING IN OPERATING SYSTEM

- Load the page only when they are needed -> demand paging
- When a process is created, some pages are loaded into main memory by pager (frame allocation algorithms)
- CPU starts executing the program. Two things happen.
  - Page is their in main memory
  - Page is not there in main memory but on disk - page fault
- In page fault case, the kernel has to bring the page from disk to main memory.
- Pure demand paging -> no pages are loaded initially. When page fault happens, then a page is loaded.

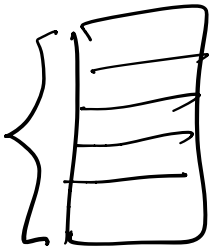


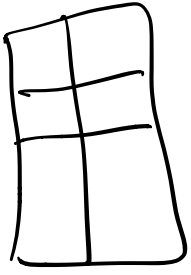


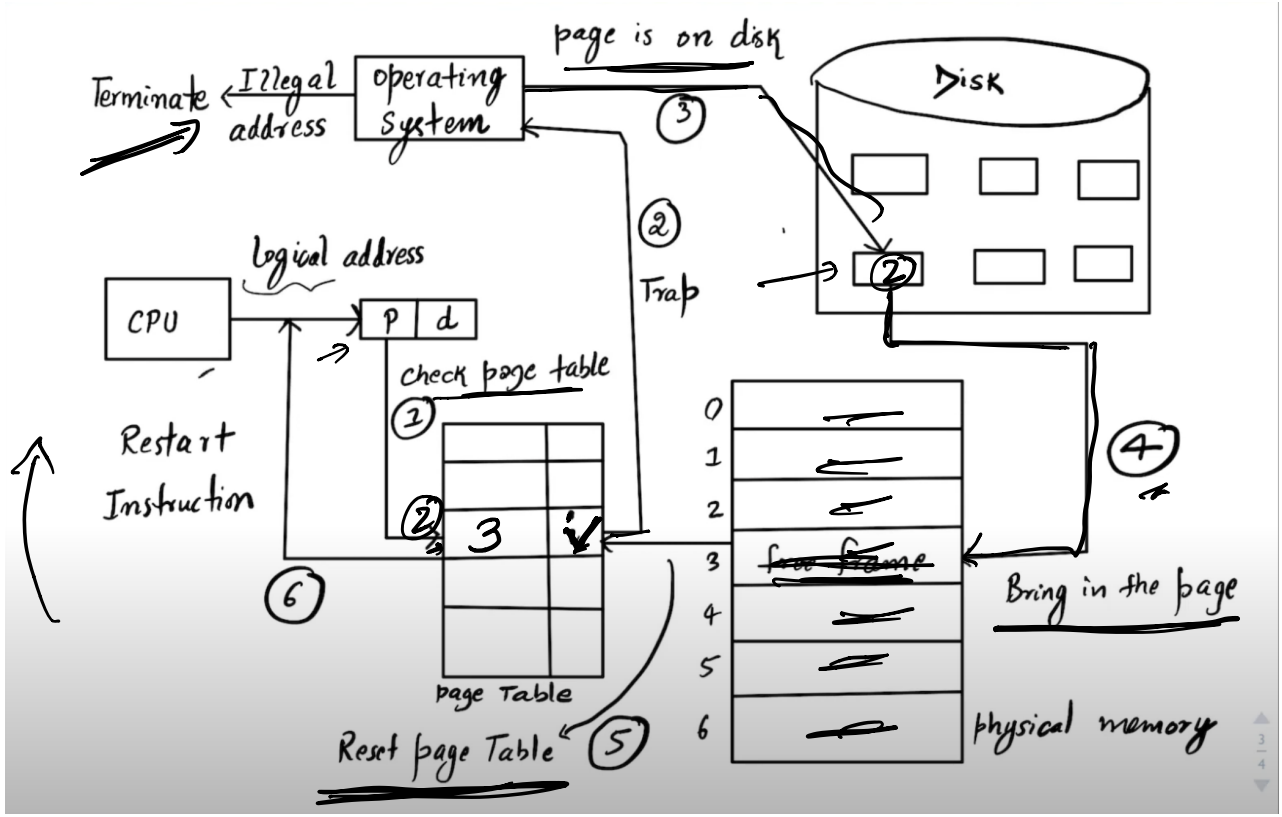
- How does kernel know about page fault?
- An extra bit called valid-invalid bit is used



- Valid
  - Page is legal
  - Is in the memory
- Invalid
  - Page is not legal ? Separate table in PCB keeps track of it
  - Page is legal but not in main memory ?

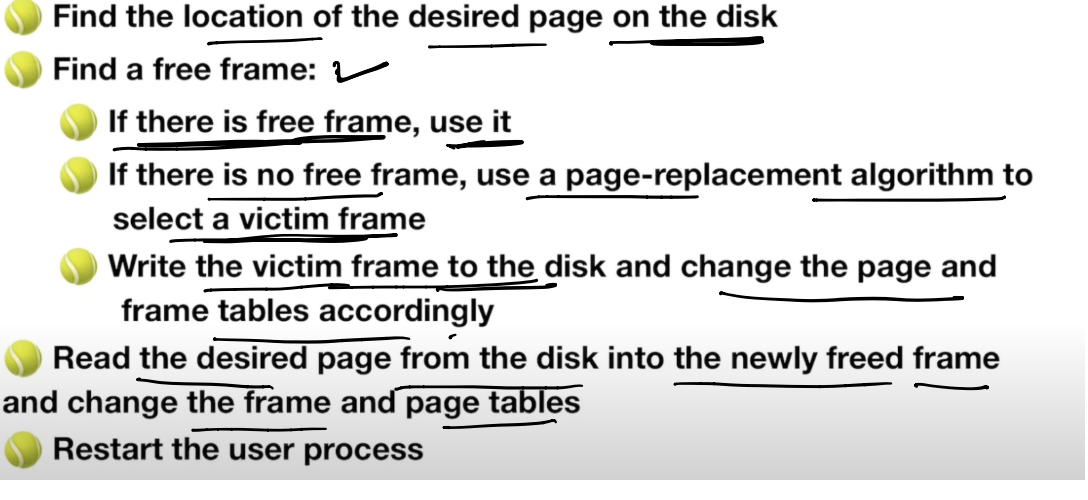


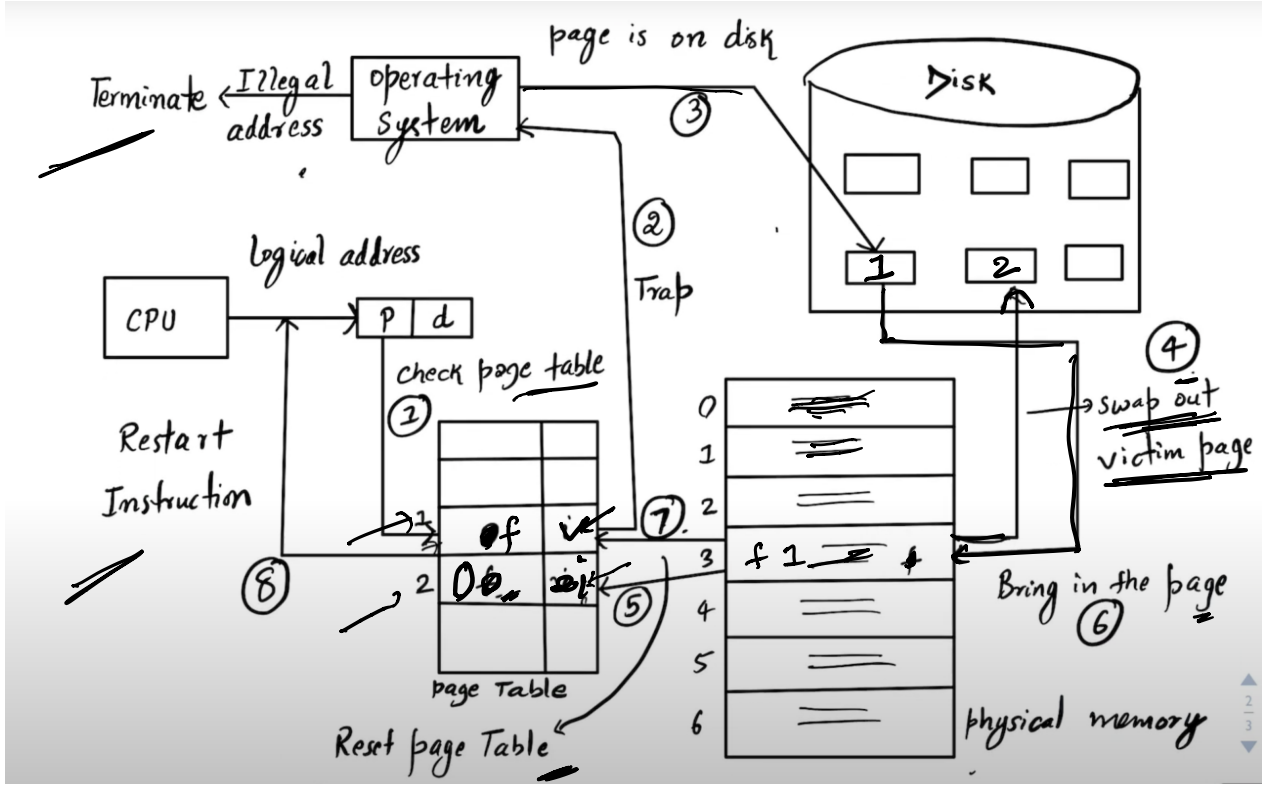



page is on disk
Terminate
Illegal address
operating System
Disk
3
2
Trap
2
Logical address
CPU
p
d
check page table
1
Restart Instruction
2
3
i
0
1
2
3
4
5
6
4
6
Bring in the page
page Table
Reset page Table
5
physical memory

PAGE REPLACEMENT TECHNIQUES

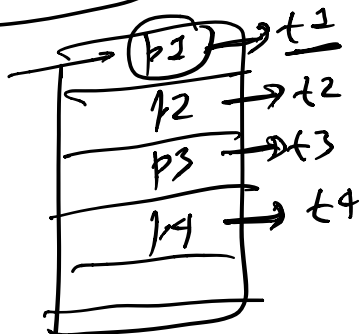
- What if all the frames are full when a page fault happens?
- Use page replacement technique to replace a page in the main memory




page is on disk
Terminate
Illegal address
operating System
Disk
3
2
Trap
Logical address
CPU
p
d
check page table
1
Restart Instruction
4
swap out victim page
0
1
2
3
4
5
6
7
8
5
Bring in the page
6
page Table
physical memory
Reset page Table

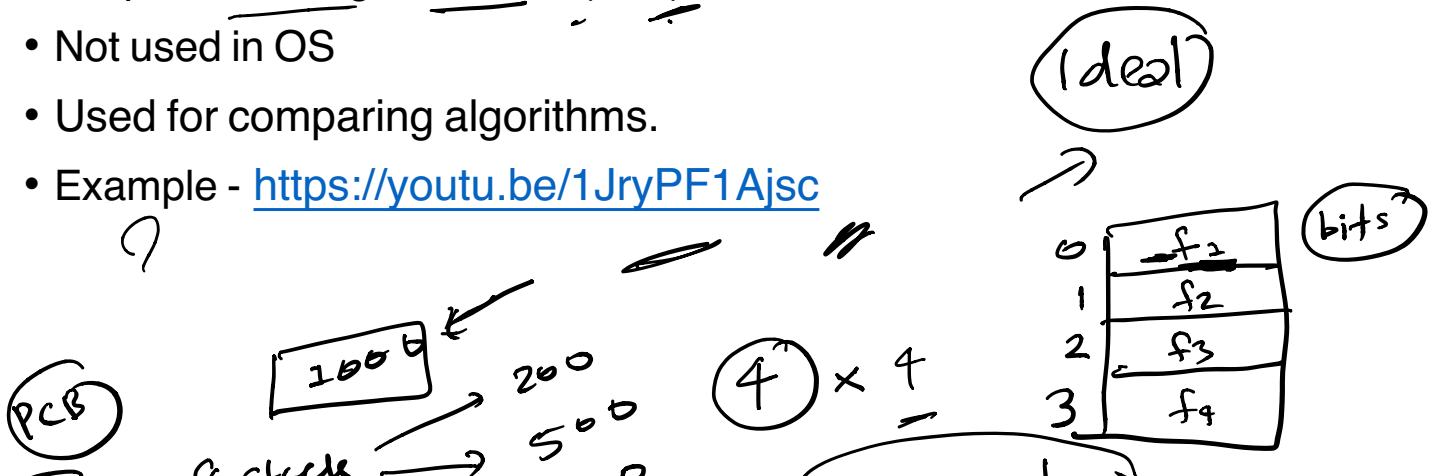
FIFO PAGE REPLACEMENT ALGORITHM

- Oldest page is replaced first
- First in, first out
- If number of frames is increased, one would expect page faults to go down.
- Using FIFO, it's not always true. This is called Belady's anomaly.
- Example - https://youtu.be/a2yJKKh48a4



OPTIMAL PAGE REPLACEMENT

- An algorithm with lowest page-fault rates
- Replace the page that will not be used for longest period of time
- Requires looking into future (SJF)
- Not used in OS
- Used for comparing algorithms.
- Example - https://youtu.be/1JryPF1Ajsc



LRU PAGE REPLACEMENT TECHNIQUE

- Replace page from past that has not been used for longest period of time.
- Optimal page replacement looking backwards
- Used in most modern OS
- Doesn't suffer from Belady's anomaly
- Example - https://youtu.be/ul3dFPkvU-I

FIFO

Request	4	7	6	1	7	6	1	2	7	2
Frame 1	4	4	4	1	1	1	1	1		
Frame 2		7	7	7	7	7	7	2		
Frame 3			6	6	6	6	6	6		
	PF	PF	PF	PF						

OPTIMAL

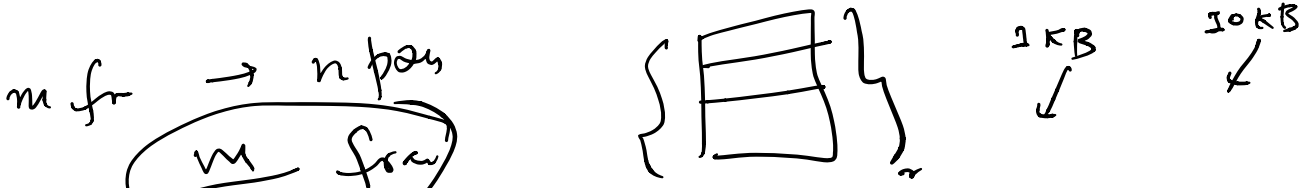
Request	4	7	6	1	7	6	1	2	7	2
Frame 1										
Frame 2										
Frame 3										

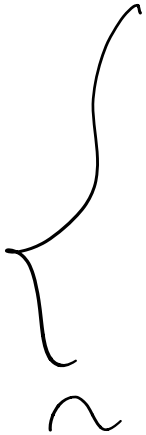
LRU

Request	4	7	6	1	7	6	1	2	7	2
Frame 1										
Frame 2										
Frame 3										

THRASHING

- More time is spent in swapping pages than doing actual work
- If more time is spent in swapping, CPU will be idle.
- Global page replacement -> Kernel sees that CPU are idle. It brings more process. Leads to more thrashing and a cycle.
- Local page replacement -> thrashing will not happen but local page replacement has some disadvantages.
- Equal frame allocation vs proportional frame allocation





FILE SYSTEM MANAGEMENT