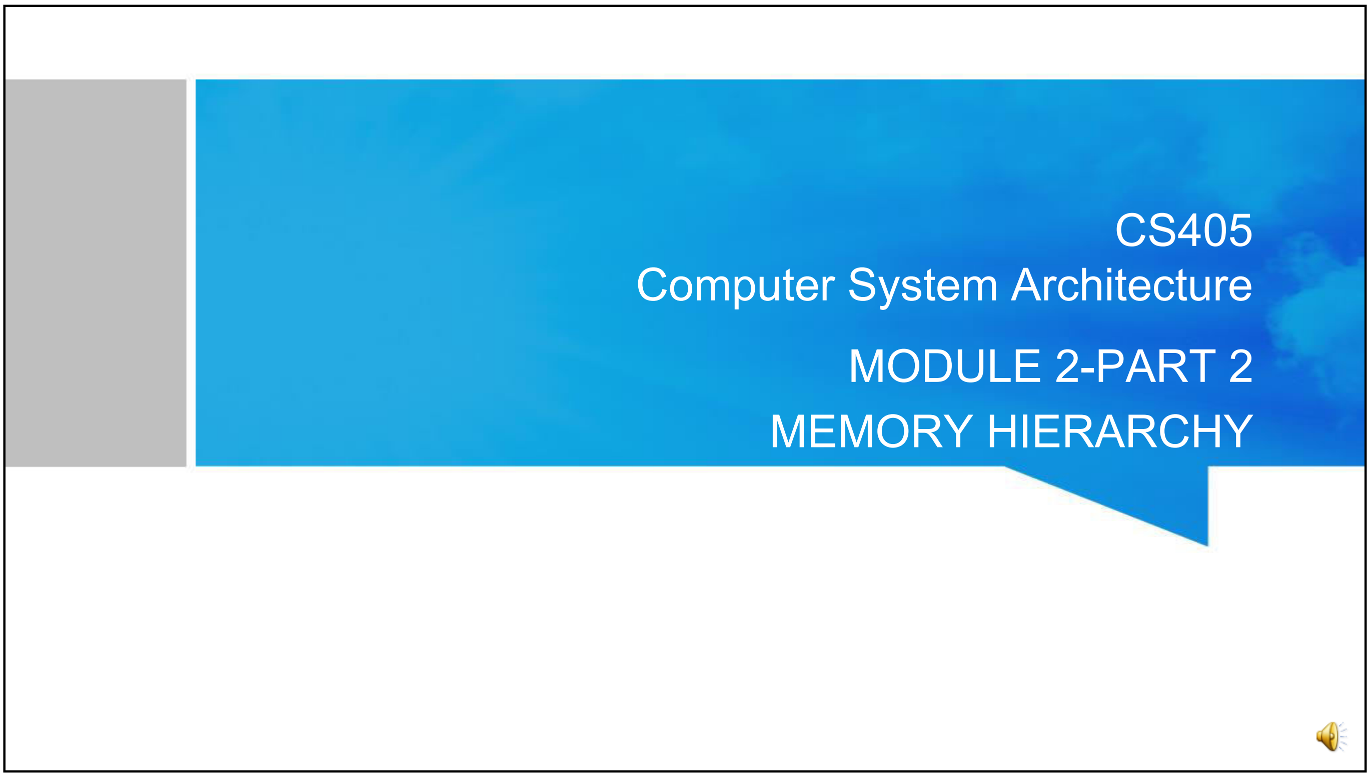
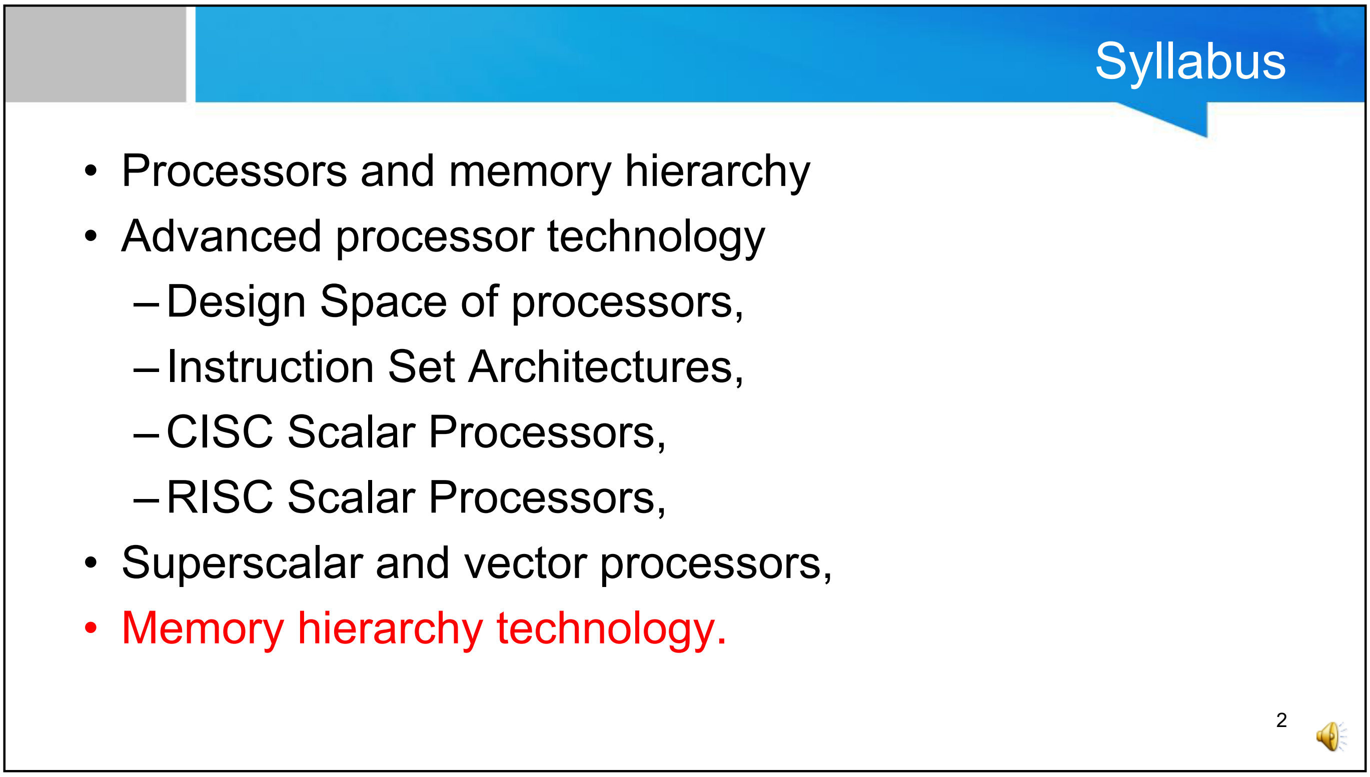


CS405
Computer System Architecture

MODULE 2-PART 2
MEMORY HIERARCHY

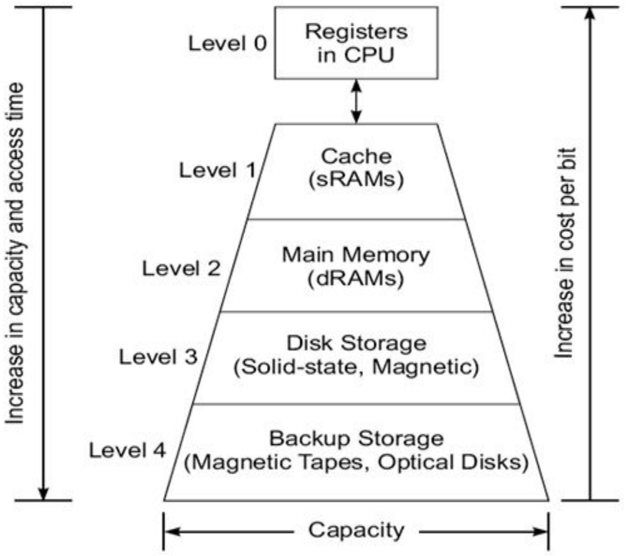


Syllabus

- Processors and memory hierarchy
- Advanced processor technology
  - Design Space of processors,
  - Instruction Set Architectures,
  - CISC Scalar Processors,
  - RISC Scalar Processors,
- Superscalar and vector processors,
- Memory hierarchy technology.

Memory Hierarchy Technology

- Memory Hierarchy
  - The goal of memory hierarchy is to keep the contents that are needed now at or near the top of the hierarchy
  - **Parameters**
    - Access time (t_i)
    - Memory size (s_i)
    - Cost per byte (c_i)
    - Transfer bandwidth (b_i)
    - Unit of transfer (x_i)
  - **Properties**
    - Inclusion
    - Coherence
    - Locality


Increase in capacity and access time
Level 0
Registers in CPU
Level 1
Cache (sRAMs)
Level 2
Main Memory (dRAMs)
Level 3
Disk Storage (Solid-state, Magnetic)
Level 4
Backup Storage (Magnetic Tapes, Optical Disks)
Increase in cost per bit
Capacity

A four-level memory hierarchy with increasing capacity and decreasing speed and cost from low to high levels

Memory Hierarchy Technology

❑Memory Hierarchy Parameters

- **T(i): Access time** *(round-trip time from CPU to i-th level memory)*
  - T(i-1) < T(i) <T(i+1)
- **S(i): Memory size** *(number of bytes or words in i-th level memory)*
  - S(i-1) < S(i) < S(i+1)
- **C(i): Cost per byte** *(per byte cost of i-th level memory; total cost estimated by C(i)*S(i))*
  - C(i-1) > C(i) > C(i+1)
- **B(i): Transfer bandwidth** *(rate at which information is transferred between adjacent levels)*
  - B(i-1) > B(i) > B(i+1)
- **X(i): Unit of transfer** *(grain size for data transfer between levels i and i+1)*
  - X(i-1) < X(i) < X(i+1)

For i=1,2,3,4 where
i=1 Register level
i=2,Main memory
i=3 Disk
i=4, Backup and storage level

Memory Hierarchy Properties

- Information stored in Memory Hierarchy(M1,M2,..Mn) satisfies following properties:
  - Inclusion property
  - Coherence property
  - Locality Property
- Cache is considered innermost level M1 this communicates with CPU directly
- Outermost Level Mn contains all information word stored

Inclusion

- Inclusion Property

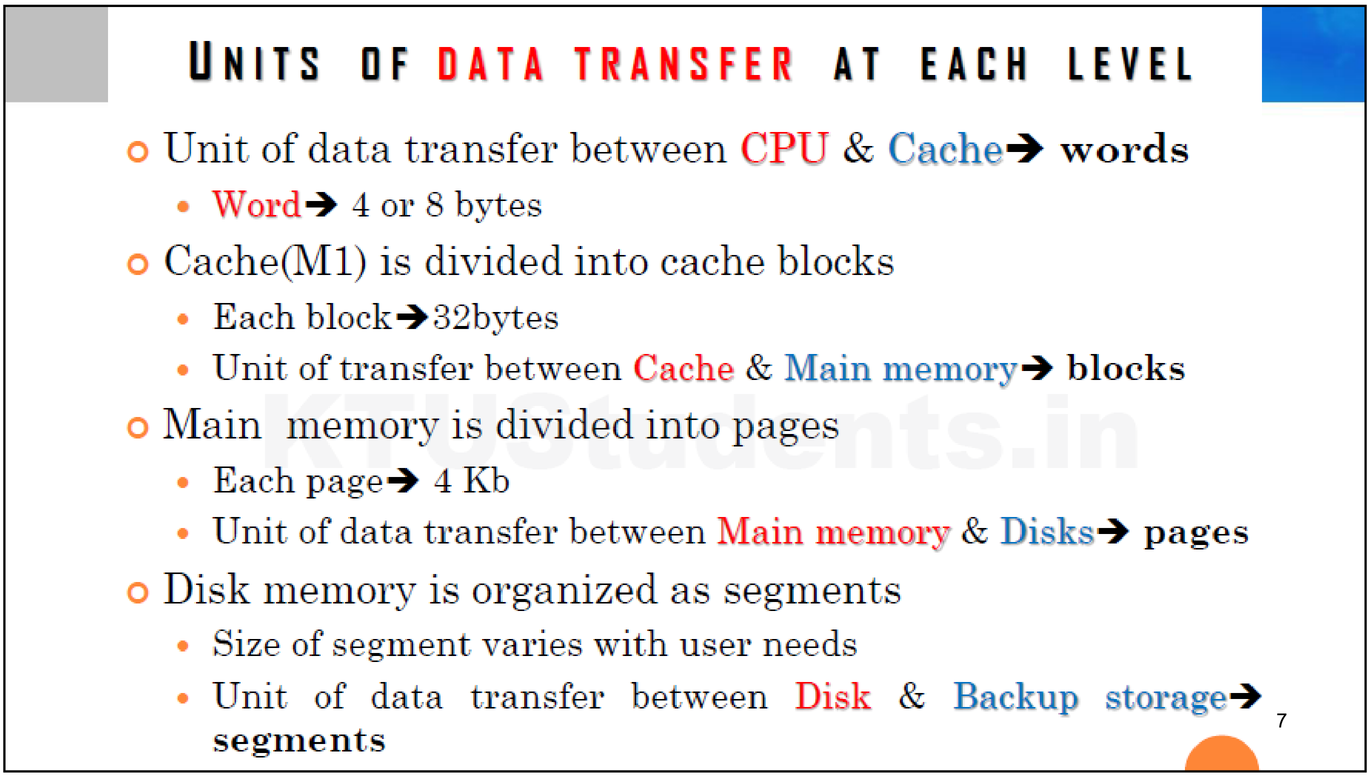
❑implies that all information items are originally stored in level Mn. During the processing, subsets of Mn are copied into Mn-1 similarly, subsets of Mn-1 are copied into Mn-2, etc.

❑If a value is found at one level, it should be present at all of the levels inner to it.

❑ M(i-2) is a subset of M(i-1) subset of M(i)

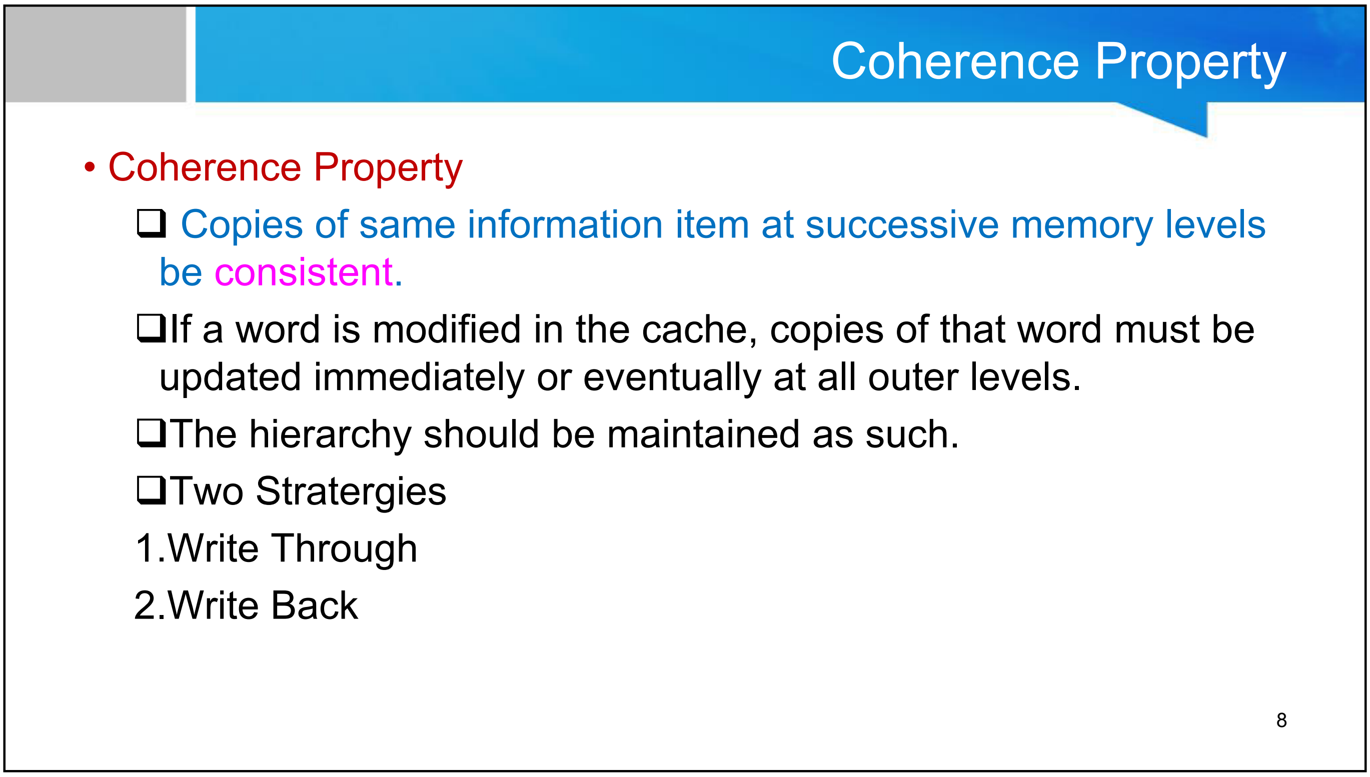
❑A word missing in M(i) implies that it is also missing from lower levels M(i-1), M(i-2),..M(1)

❑The highest level is backup storage where everything can be found



UNITS OF DATA TRANSFER AT EACH LEVEL

- Unit of data transfer between CPU & Cache→ **words**
  - Word→ 4 or 8 bytes
- Cache(M1) is divided into cache blocks
  - Each block→32bytes
  - Unit of transfer between Cache & Main memory→ **blocks**
- Main memory is divided into pages
  - Each page→ 4 Kb
  - Unit of data transfer between Main memory & Disks→ **pages**
- Disk memory is organized as segments
  - Size of segment varies with user needs
  - Unit of data transfer between Disk & Backup storage→ **segments**



Coherence Property

- Coherence Property
  - ❑ Copies of same information item at successive memory levels be consistent.
  - ❑If a word is modified in the cache, copies of that word must be updated immediately or eventually at all outer levels.
  - ❑The hierarchy should be maintained as such.
  - ❑Two Stratergies

1.Write Through

2.Write Back

Coherence strategies

- Write through
  - As soon as a data item in Mi is modified, immediate update of the corresponding data item(s) in Mi+1, Mi+2, … Mn is required. This is the most aggressive (and expensive) strategy.
- Write-back
  - The update of the data item in Mi+1 corresponding to a modified item in Mi is not updated until the block or page in Mi that contains it is replaced or removed. This is the most efficient approach, but cannot be used (without modification) when multiple processors share Mi+1, Mi+2, … Mn.

Locality of Reference

- Locality Example:
- Data
  - Reference array elements in succession (stride-1 reference pattern)
  - Reference `sum` each iteration
- Instructions
  - Reference instructions in sequence
  - Cycle through loop repeatedly

```
sum = 0;
for (i = 0; i < n; i++)
    sum += a[i];
return sum;
```

❖ Famous 40/10 rule that comes from empirical observation: "A program spends 40% of its time in 10% of its code"

Locality of Reference

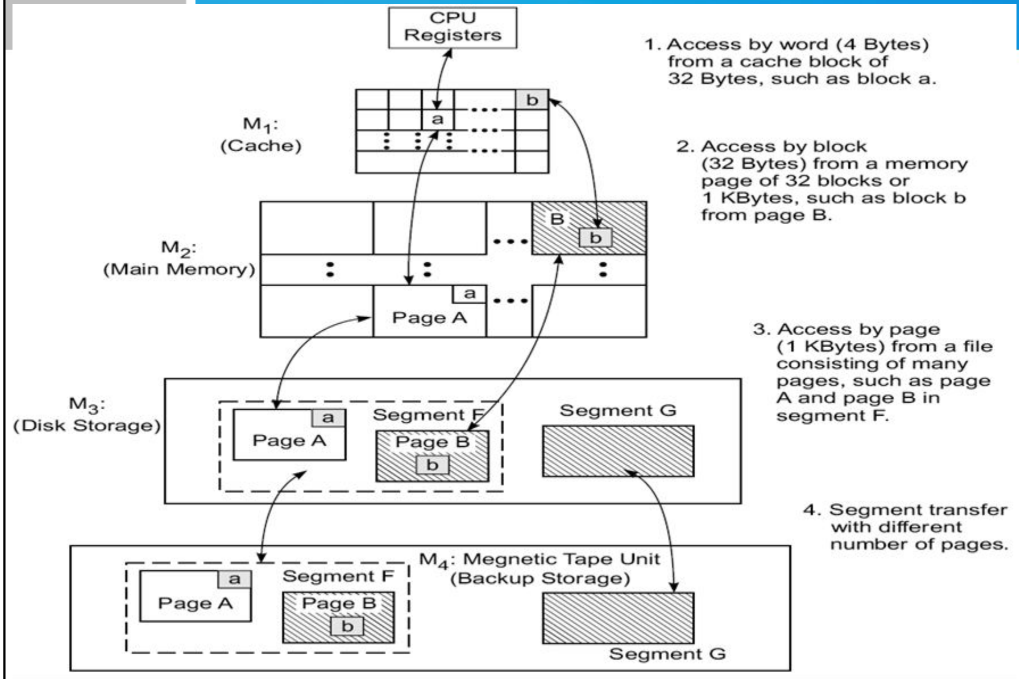
- **Temporal**
  - recently referenced items are likely to be referenced again in near future (loop iterations, process stacks, temporary variables or subroutines)
  - Thus if location M is referenced at time t, then (location M) will be referenced again at some time t+Dt.
- **Spatial**
  - tendency of a process to access items whose addresses are near one another (elements of an array, subroutines or macros stored in near by locations)
  - Thus if location M is referenced at time t, then another location M±Dm will be referenced again at some time t+Dt.

- **Sequential**
  - execution of instructions follows a sequential program order unless branch instructions create out-of-order execution.
  - Thus if location M is referenced at time t, then locations M+1,M+2,… will be referenced again at some time t+Dt, t+Dt', etc.

Memory Design Implications (Each type of locality affects design of memory hierarchy)

One of the implications of the locality is data and instructions should have separate data and instruction caches. The main advantage of separate caches is that one can fetch instructions and operands simultaneously. (Design basis of Harvard architecture)

Memory Hierarchy



1. Access by word (4 Bytes) from a cache block of 32 Bytes, such as block a.

2. Access by block (32 Bytes) from a memory page of 32 blocks or 1 KBytes, such as block b from page B.

3. Access by page (1 KBytes) from a file consisting of many pages, such as page A and page B in segment F.

4. Segment transfer with different number of pages.

The inclusion property and data transfers between adjacent levels of a memory hierarchy

Memory Capacity Planning

- The performance of a memory hierarchy is determined by the effective access time Teff to any level in the hierarchy. It depends on the hit ratios and access frequencies at successive levels.

Memory Capacity Planning-Terms

- Hit Ratio
  - when an information item is found in Mi, we call it a hit, otherwise a miss.
  - Considering memory level Mi to Mi-1 in a hierarchy , i=1,2,3…n-1. The hit ratio ,hi at Mi is the probability that an item required will be found in Mi. The miss ratio at Mi is 1-hi.
  - The hit ratios at successive levels are a function of memory capacities, management policies and program behavior.
  - Every time a miss occurs a penalty has to be paid to access next higher level of memory. Cache miss is 2 to 4 times costlier than a cache hit. Page faults are 1000 to 10000 times costly than page hit
  - We assume h0 =0 and hn = 1, which means CPU always accesses M1 first, and the access to the outermost memory Mn is always a hit.

- Access frequency
  - Access frequency to Mi is defined as

    fi =(1-h1)(1-h2)….(1-hi-1)hi.
  - This is the probability of successfully accessing Mi when there are i-1 misses at the lower levels and a hit at Mi. Note that $\sum_{i=1}^{n} fi = 1$ and $f_1 = h_1$
  - Due to the locality property, the access frequencies decrease very rapidly from low to high levels; that is, f1 >>f2>> f3>> …>> fn.
  - This implies that the inner levels of memory are accessed more often than the outer levels.

- Average memory-access time = effective access time = Hit time + Miss rate x Miss penalty (ns or clocks)
- that is, our memory access, on average, is the time it takes to access the cache, plus for a miss, how much time it takes to access memory

- Effective access time
  - In practice, we wish to achieve as high a hit ratio as possible at M1. Every time a miss occurs, a penalty must be paid to access the next higher level of memory.
  - The misses have been called block misses in the cache and page faults in the main memory because blocks and pages are the units of transfer between these levels.
  - Using the access frequencies fi=1,2,….n, we can formally define the effective access time of a memory hierarchy as follows

$$T_{eff} = \sum_{i=1}^{n} f_i \cdot t_i$$

$$= h_1 t_1 + (1 - h_1) h_2 t_2 + (1 - h_1)(1 - h_2) h_3 t_3 + \ldots +$$

$$(1 - h_1)(1 - h_2) \ldots (1 - h_{n-1}) t_n$$

- Hierarchy Optimization
  - The total cost of a memory hierarchy is estimated as follows

$$C_{total}=\sum_{i=1}^{n} \quad Ci.Si$$

  - Ci=cost per byte Si=memory size ti=access time
  - This implies cost is distributed across n levels.
  - Since c1>c2>c3>…cn we have to choose s1<s2<s3<..sn.
  - The optimal design of a memory hierarchy should result in a T_{eff} close to the t1 of M1 and a total cost close to the cost of Mn.
  - In reality, this is difficult to achieve due to the tradeoffs among n levels.

- The optimization process can be formulated as a linear programming problem, given a ceiling C0 on the total cost— that is, a problem to minimize

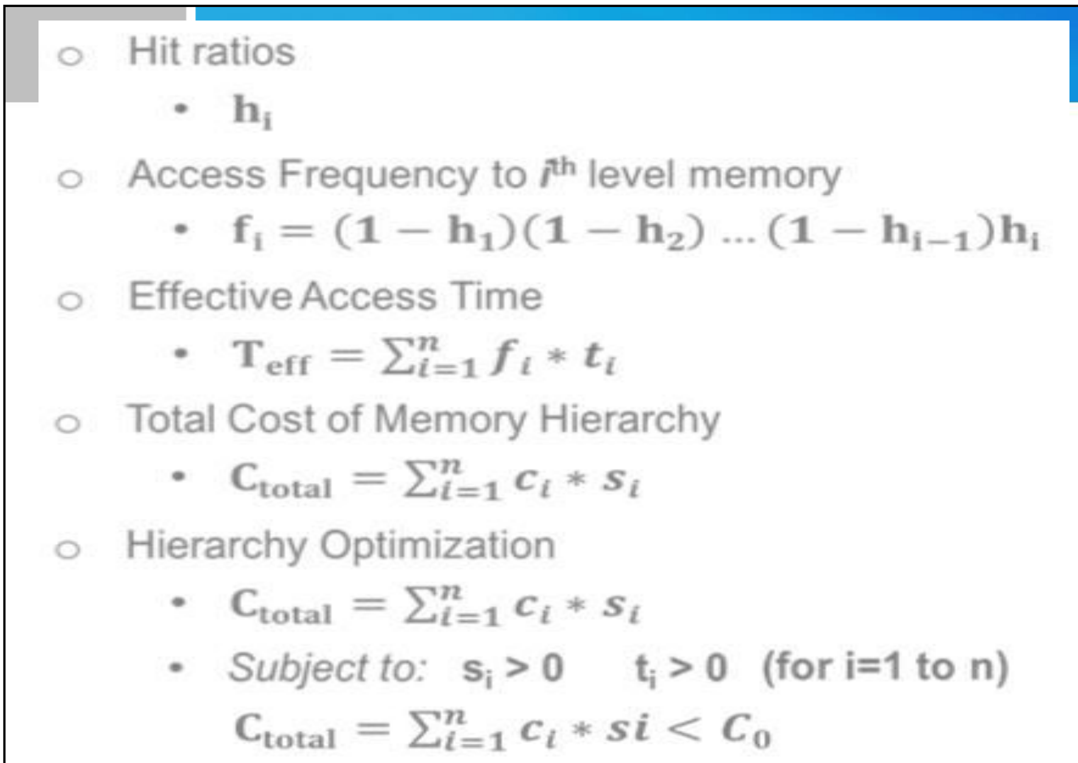
$$T_{eff} = \sum_{i=1}^{n} f_i \cdot t_i$$

- subject to the following constraints:

$$s_i > 0, t_i > 0 \quad \text{for } i = 1, 2,\ldots, n$$

$$C_{total} = \sum_{i=1}^{n} c_i \cdot s_i < C_0$$

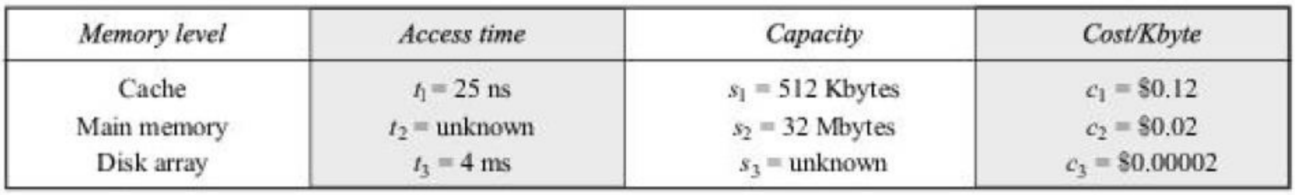
- the above optimization involves tradeoffs among ti,ci,si and fior hi at all levels = 1, 2, . . ., n.



- Hit ratios
  - $\mathbf{h_i}$
- Access Frequency to i^{th} level memory
  - $\mathbf{f_i = (1 - h_1)(1 - h_2) \ldots (1 - h_{i-1})h_i}$
- Effective Access Time
  - $\mathbf{T_{eff}} = \sum_{i=1}^{n} f_i * t_i$
- Total Cost of Memory Hierarchy
  - $\mathbf{C_{total}} = \sum_{i=1}^{n} c_i * s_i$
- Hierarchy Optimization
  - $\mathbf{C_{total}} = \sum_{i=1}^{n} c_i * s_i$
  - *Subject to:* $s_i > 0$ $t_i > 0$ (for i=1 to n)

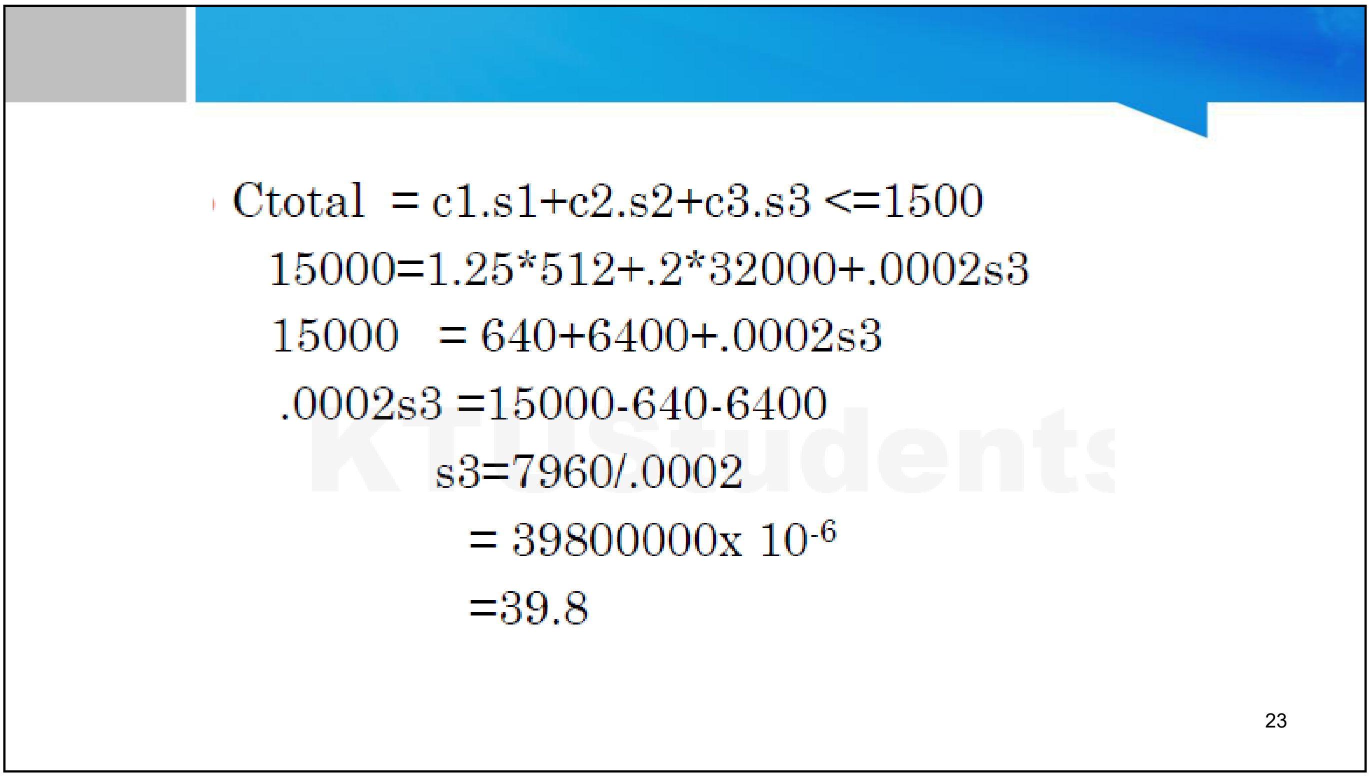
    $\mathbf{C_{total}} = \sum_{i=1}^{n} c_i * si < C_0$

- Consider the design of a three-level memory hierarchy with the following specifications for memory characteristics:



Memory level	*Access time*	*Capacity*	*Cost/Kbyte*
Cache	t_1 = 25 ns	s_1 = 512 Kbytes	c_1 = \$0.12
Main memory	t_2 = unknown	s_2 = 32 Mbytes	c_2 = \$0.02
Disk array	t_3 = 4 ms	s_3 = unknown	c_3 = \$0.00002

- Aim is to achieve an effective memory-access time t=850ns
- cache hit ratio h1 = 0.98 and
- a main memory hit ratio by = 0.99
- Also, the total cost is upper-bounded by $1,500.
- Calculate unknown specifications based on these conditions



Ctotal = c1.s1+c2.s2+c3.s3 <=1500

15000=1.25*512+.2*32000+.0002s3

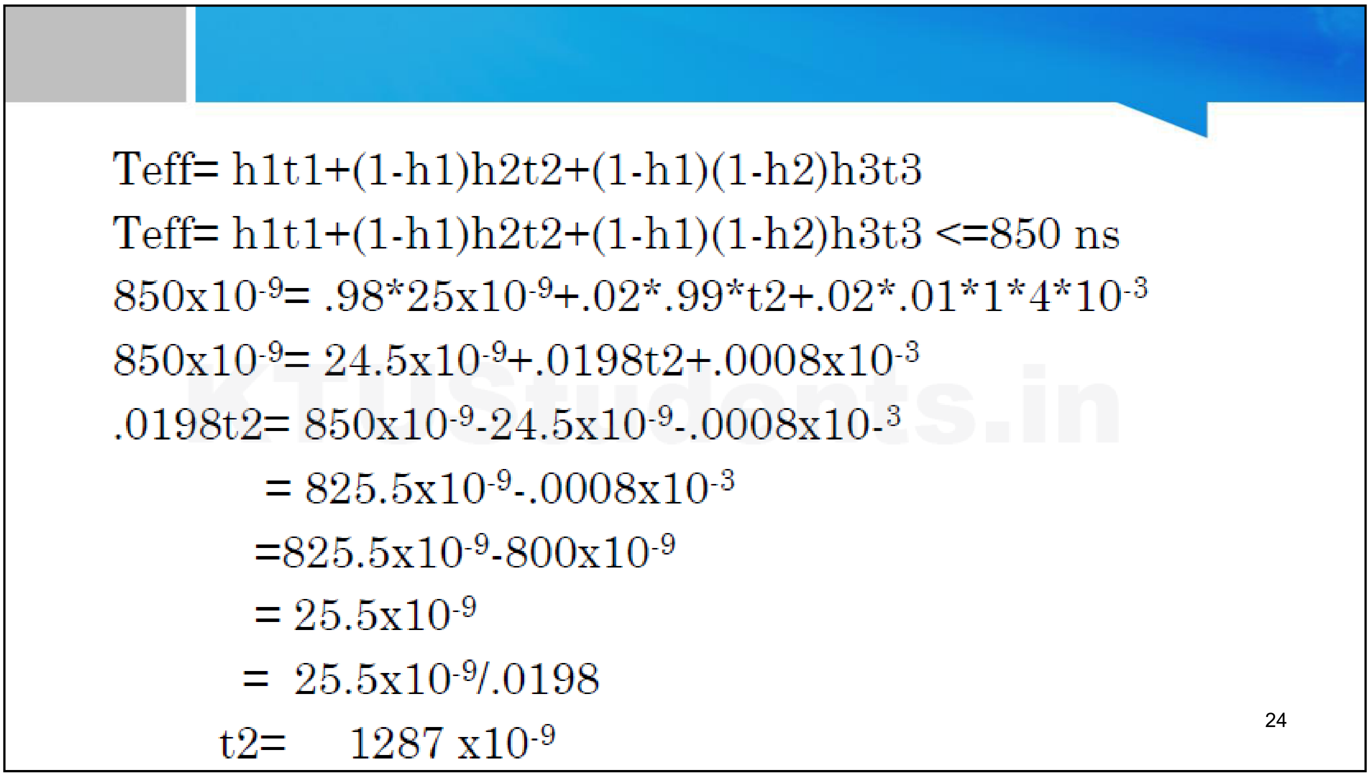
15000 = 640+6400+.0002s3

.0002s3 =15000-640-6400

s3=7960/.0002

$= 39800000\text{x } 10^{-6}$

=39.8



Teff= h1t1+(1-h1)h2t2+(1-h1)(1-h2)h3t3

Teff= h1t1+(1-h1)h2t2+(1-h1)(1-h2)h3t3 <=850 ns

$850\text{x}10^{-9}$= $.98*25\text{x}10^{-9}$+.02*.99*t2+.02*.01*1*4*10^{-3}

$850\text{x}10^{-9}$= $24.5\text{x}10^{-9}$+.0198t2+$.0008\text{x}10^{-3}$

.0198t2= $850\text{x}10^{-9}$-$24.5\text{x}10^{-9}$-$.0008\text{x}10^{-3}$

= $825.5\text{x}10^{-9}$-$.0008\text{x}10^{-3}$

=$825.5\text{x}10^{-9}$-$800\text{x}10^{-9}$

= $25.5\text{x}10^{-9}$

= $25.5\text{x}10^{-9}$/.0198

t2= $1287\text{ x}10^{-9}$

2-level Cache Effective Access Time :

- Average memory access time =

  hit time_0 + miss rate_0 * (hit time_1 + miss rate_1 * miss penalty_1)

- including the impact of paging:
- effective access time = hit time_0 + miss rate_0 * (hit time_1 + miss rate_1 * (hit time_2 + miss rate_2 * miss penalty_2))
  - Level 0 is on-chip cache
  - Level 1 is off-chip cache
  - Level 2 is main memory
  - Level 3 is disk (miss penalty_2 is disk access time, which is lengthy)
- Access time: time to access outer level = f(latency to outer level)
- Transfer time: time to transfer the block =f(BW between upper & lower levels)

❖ Example:

Hierarchy = primary cache, secondary cache, main memory

t1 = Access time to primary cache = 5 ns

t2 = Access time to secondary cache = 20 ns

t3 = Access time to main memory = 70 ns

h1 = hit ratio to primary cache = 0.9

h2 = hit ratio to secondary cache = 0.95

h3 = hit ratio to main memory = 1

What is the average access time for this memory hierarchy?

➢ Teff = h1.t1 + (1-h1)h2.t2 + (1-h1)(1-h2)h3.t3 + ...
  + (1-h1)(1-h2)(1-h3) ... (1-hn-1)hn.tn

❖Example:

- On chip cache hit rate is 90%, hit time is 5 ns, off chip cache hit rate is 96%, hit time is 10 ns, main memory hit rate is 99.8%, hit time is 60 ns, memory miss penalty is 10 ms = 10,000 ns.

(memory miss penalty is the same as the disk hit time, or disk access time)

- Access time

$= \text{hit time}_0 + \text{miss rate}_0 * (\text{hit time}_1 + \text{miss rate}_1 * (\text{hit time}_2 + \text{miss rate}_2 * \text{miss penalty}_2))$

$= 5 \text{ ns} + .10 * (10 \text{ ns} + .04 * (60 \text{ ns} + .002 * 10{,}000 \text{ ns}))$

$= 6.32 \text{ ns}$

–So our memory hierarchy adds over 20% to our memory access

Virtual Memory

- Just as DRAM(main memory) acts as a backup for cache, hard disk (known as the swap space) acts as a backup for DRAM
- This is known as virtual memory

–Virtual memory is necessary because most programs are too large to store entirely in memory

- Also, there are parts of a program that are not used very often, so why waste the time loading those parts into memory if they won’t be used?

–Page – a fixed sized unit of memory – all programs and data are broken into pages

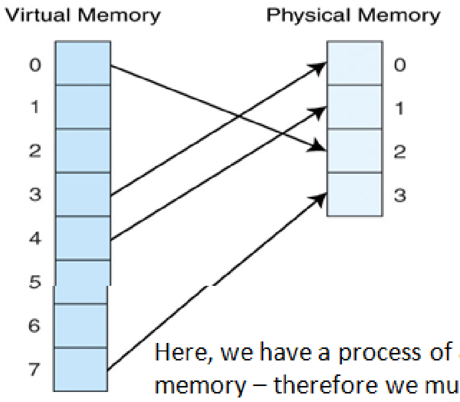
–Paging – the process of bringing in a page when it is needed (this might require throwing a page out of memory, moving it back to the swap disk)

- The operating system is in charge of Virtual Memory and it moves needed pages into memory from disk and keeps track of where a specific page is placed

The Paging Process

- When the CPU generates a memory address, it is a logical (or virtual) address
  - The first address of a program is 0, so the logical address is merely an offset into the program or into the data segment
- For instance, address 25 is located 25 from the beginning of the program
- But 25 is not the physical address in memory, so the logical address must be translated (or *mapped*) into a physical address
  - Assume memory is broken into fixed size units known as frames (1 page fits into 1 frame)
- We know the logical address as its page # and the offset into the page
  - We have to translate the page # into the frame # (that is, where is that particular page currently be stored in memory – or is it even in memory?)
- Thus, the mapping process for paging means finding the frame # and replacing the page # with it

Example of Paging



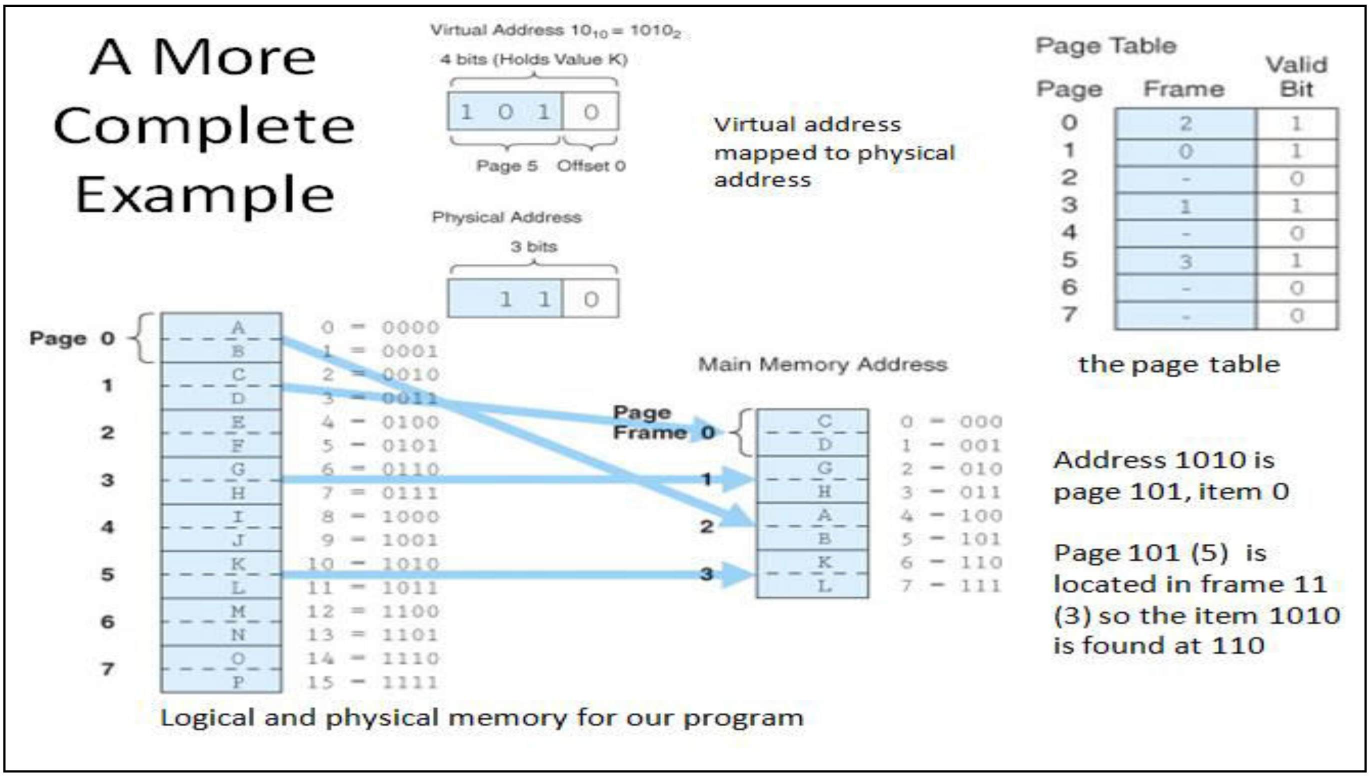
Page Table

Page	Frame #	Valid Bit
0	2	1
1	-	0
2	-	0
3	0	1
4	1	1
5	-	0
6	-	0
7	3	1

Here, we have a process of 8 pages but only 4 physical frames in memory – therefore we must place a page into one of the available frames in memory whenever a page is needed

At this point in time, pages 0, 3, 4 and 7 have been moved into memory at frames 2, 0, 1 and 3 respectively

This information (of which page is stored in which frame) is stored in memory in a location known as the Page Table. The page table also stores whether the given page has been modified (the valid bit – much like our cache)



A More Complete Example

Virtual address mapped to physical address

Page Table

Page	Frame	Valid Bit
0	2	1
1	0	1
2	-	0
3	1	1
4	-	0
5	3	1
6	-	0
7	-	0

the page table

Address 1010 is page 101, item 0

Page 101 (5) is located in frame 11 (3) so the item 1010 is found at 110

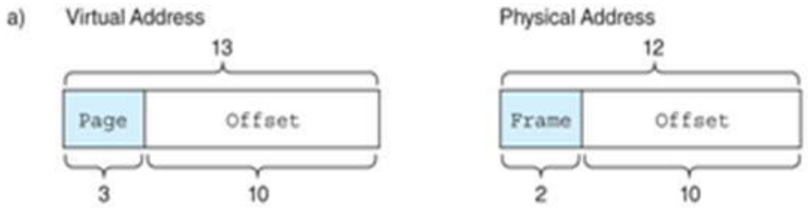
Logical and physical memory for our program

Page Faults

- Just as cache is limited in size, so is main memory – a process is usually given a limited number of frames
- What if a referenced page is not currently in memory?
  - The memory reference causes a *page fault*
    - The page fault requires that the OS handle the problem
  - The process' status is saved and the CPU switches to the OS
  - The OS determines if there is an empty frame for the referenced page, if not, then the OS uses a replacement strategy to select a page to discard
    - if that page is dirty, then the page must be written to disk instead of discarded
  - The OS locates the requested page on disk and loads it into the appropriate frame in memory
  - The page table is modified to reflect the change
    - Page faults are time consuming because of the disk access – this causes our effective memory access time to deteriorate badly!

Another Paging Example

Virtual Address Space: 8K = 2^{13}
Physical Memory: 4K = 2^{12}
Page Size: 1K = 2^{10}



b) Page Table

Page	Frame	Valid Bit
0	-	0
1	3	1
2	0	1
3	-	0
4	-	0
5	1	1
6	2	1
7	-	0

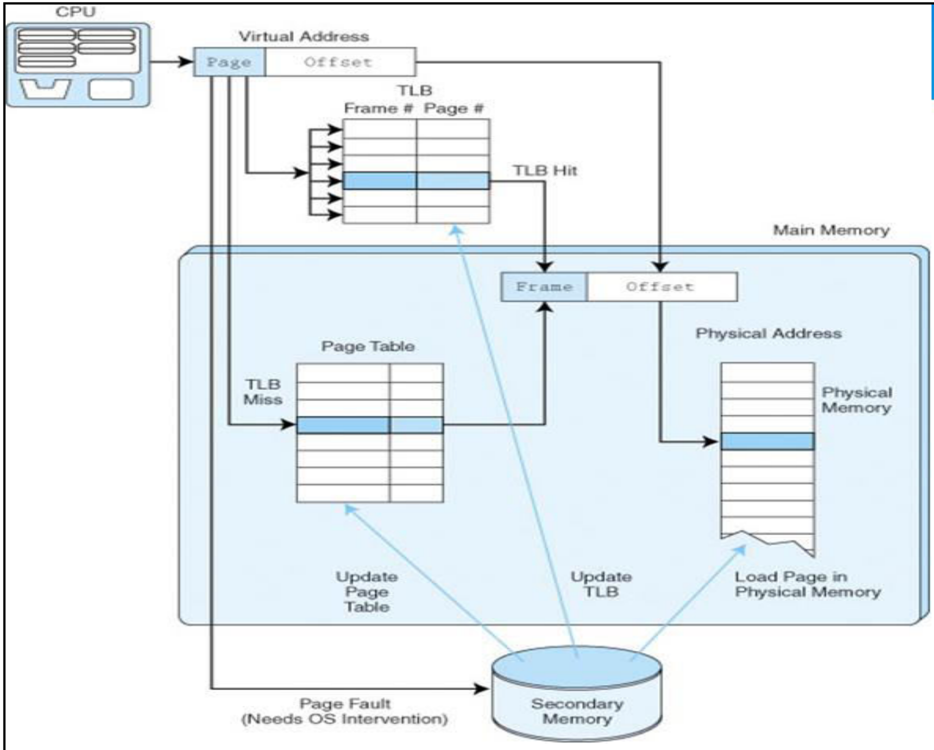
c) Addresses

Page	Base 10	Base 16
0 :	0 - 1023	0 - 3FF
1 :	1024 - 2047	400 - 7FF
2 :	2048 - 3071	800 - BFF
3 :	3072 - 4095	C00 - FFF
4 :	4096 - 5119	1000 - 13FF
5 :	5120 - 6143	1400 - 17FF
6 :	6144 - 7167	1800 - 1BFF
7 :	7168 - 8191	1C00 - 1FFF

Here, we have 13 bits for our addresses even though main memory is only 4K = 2^{12}

The Full Paging Process

- if every memory access now requires first accessing the page table, which is in memory, it slows down our computer
- So we move the most used portion of the page table into a special cache known as Translation Lookaside Buffer, TLB



CPU

Virtual Address

Page | Offset

TLB

Frame # Page #

TLB Hit

Main Memory

Frame | Offset

Physical Address

Page Table

TLB Miss

Physical Memory

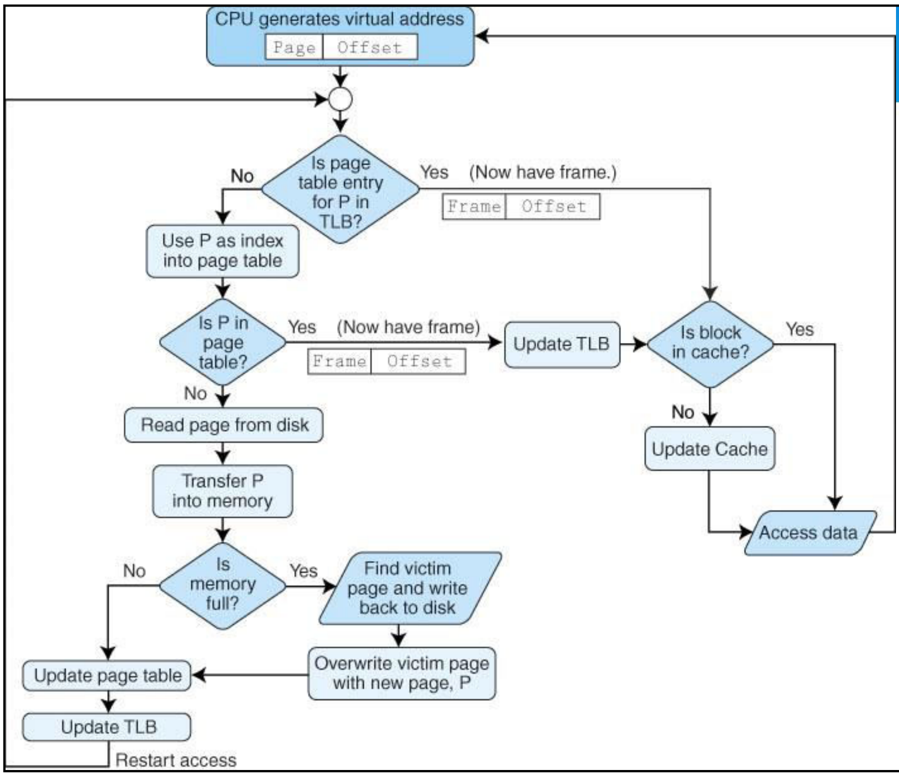
Update Page Table

Update TLB

Load Page in Physical Memory

Page Fault (Needs OS Intervention)

Secondary Memory



CPU generates virtual address

Page | Offset

Is page table entry for P in TLB?

No

Yes (Now have frame.)

Frame | Offset

Use P as index into page table

Is P in page table?

Yes (Now have frame)

Frame | Offset

Update TLB

Is block in cache?

Yes

No

Update Cache

Access data

No

Read page from disk

Transfer P into memory

Is memory full?

No

Yes

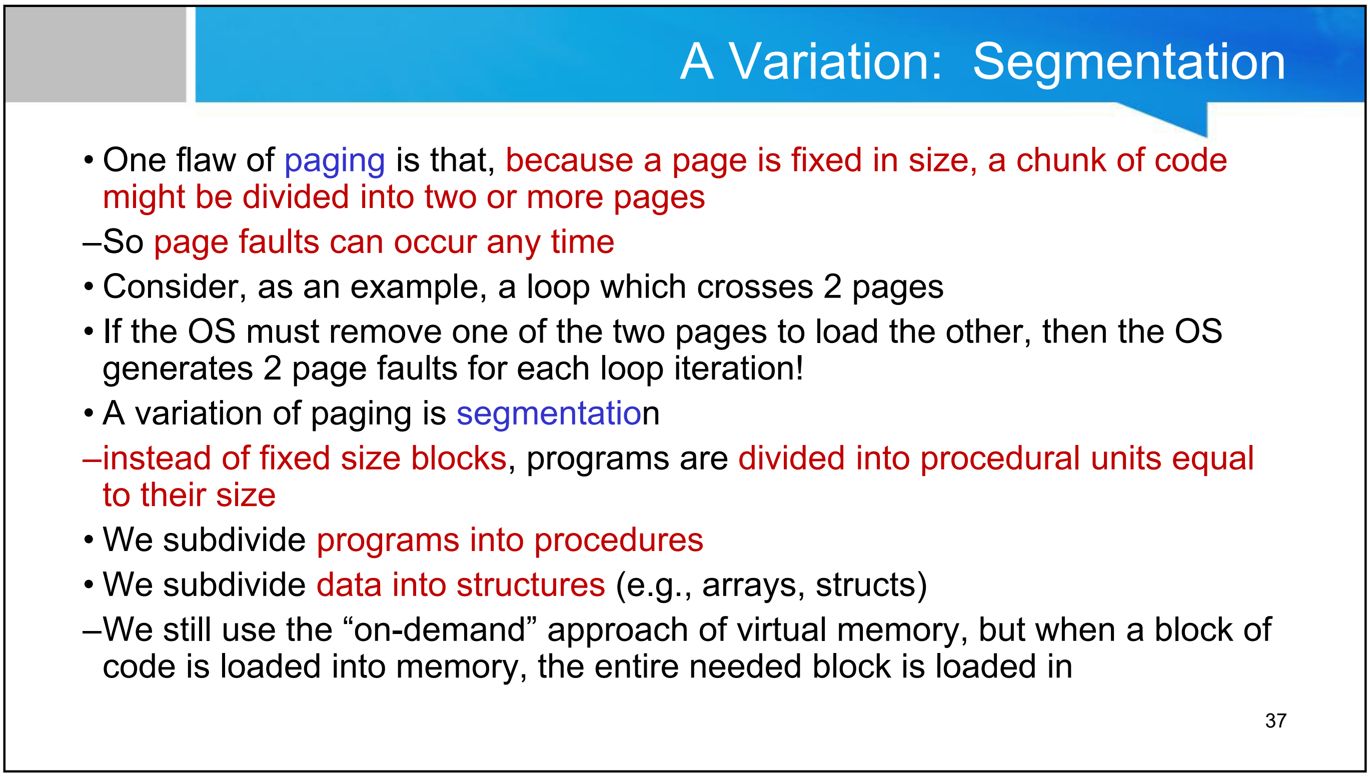
Find victim page and write back to disk

Overwrite victim page with new page, P

Update page table

Update TLB

Restart access



A Variation: Segmentation

- One flaw of paging is that, because a page is fixed in size, a chunk of code might be divided into two or more pages

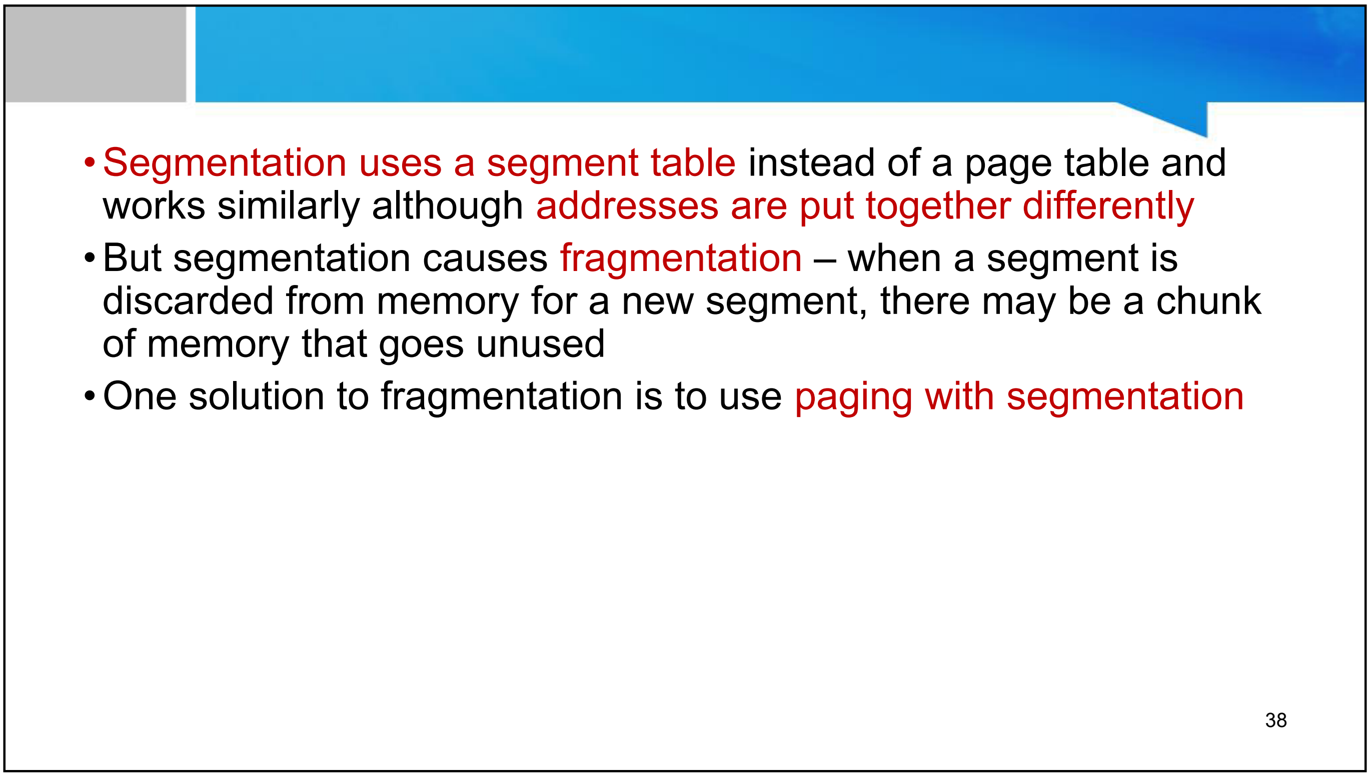
–So page faults can occur any time

- Consider, as an example, a loop which crosses 2 pages
- If the OS must remove one of the two pages to load the other, then the OS generates 2 page faults for each loop iteration!
- A variation of paging is segmentation

–instead of fixed size blocks, programs are divided into procedural units equal to their size

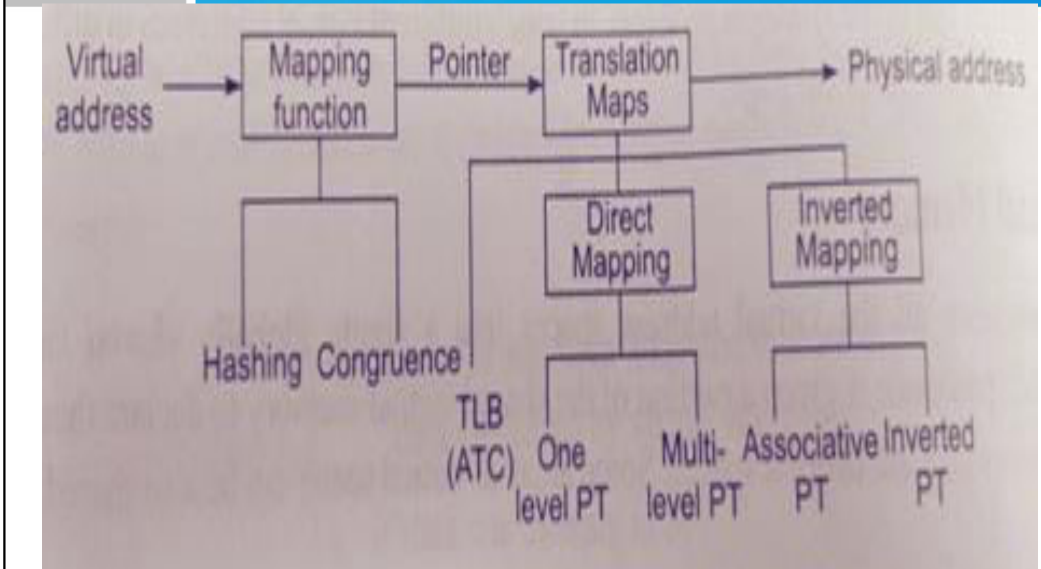
- We subdivide programs into procedures
- We subdivide data into structures (e.g., arrays, structs)

–We still use the “on-demand” approach of virtual memory, but when a block of code is loaded into memory, the entire needed block is loaded in



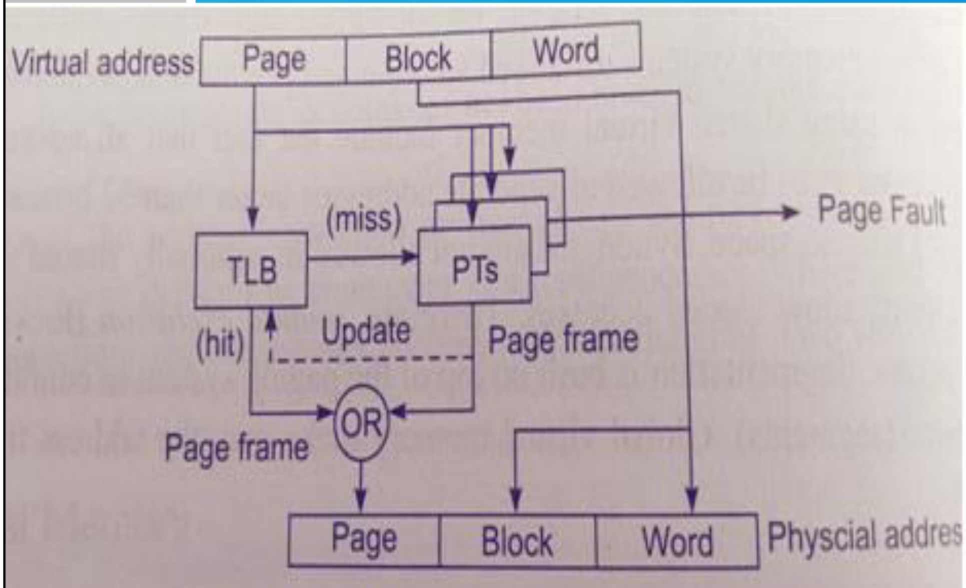
- Segmentation uses a segment table instead of a page table and works similarly although addresses are put together differently
- But segmentation causes fragmentation – when a segment is discarded from memory for a new segment, there may be a chunk of memory that goes unused
- One solution to fragmentation is to use paging with segmentation

Virtual Address Translation schemes



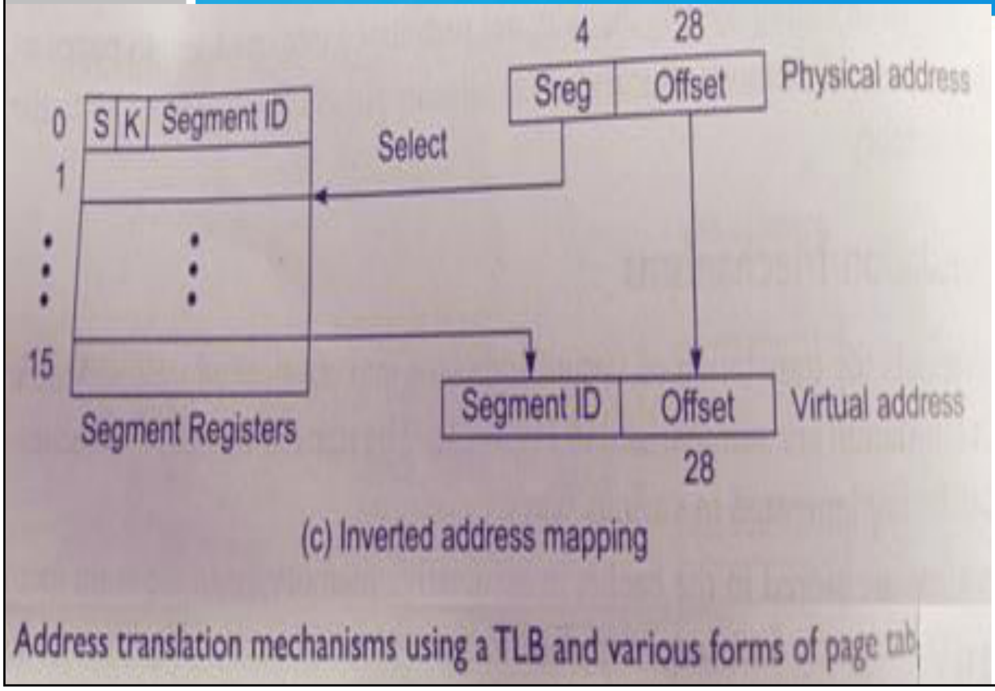
(a) Virtual addrress translation schemes (PT = page table)

Address Translation using TLB and PTs



(b) Use of a TLB and PTs for address translation

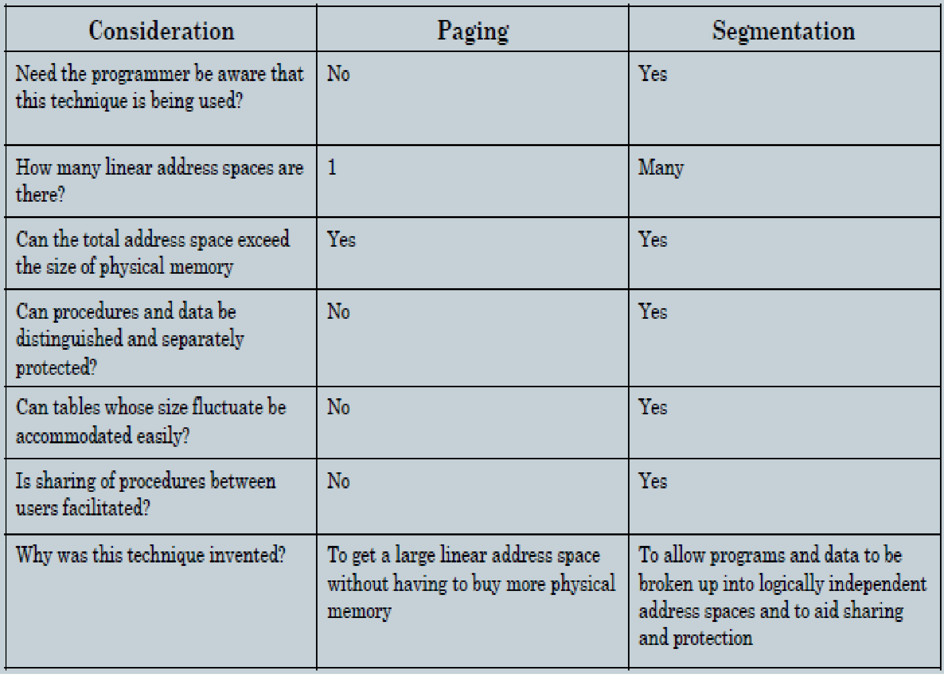
Inverted Address Mapping



(c) Inverted address mapping

Address translation mechanisms using a TLB and various forms of page tab

Paging vs Segmentation

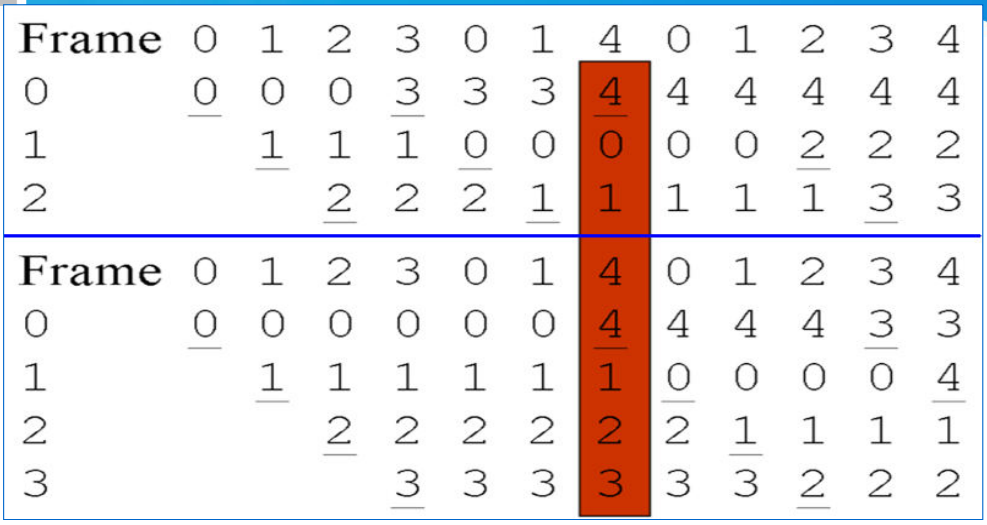


Consideration	Paging	Segmentation
Need the programmer be aware that this technique is being used?	No	Yes
How many linear address spaces are there?	1	Many
Can the total address space exceed the size of physical memory	Yes	Yes
Can procedures and data be distinguished and separately protected?	No	Yes
Can tables whose size fluctuate be accommodated easily?	No	Yes
Is sharing of procedures between users facilitated?	No	Yes
Why was this technique invented?	To get a large linear address space without having to buy more physical memory	To allow programs and data to be broken up into logically independent address spaces and to aid sharing and protection

Page Replacement Algorithms

- When there is a page fault, the referenced page must be loaded.
  - If there is no available frame in memory, then one page (victim) is selected for replacement.
  - If the selected page has been modified, it must be copied back to disk (swapped out)
- Various algorithms are used:
  FIFO
  LRU
  LFU
  Circular FIFO
  Optimal
  Random

Belady’s Anomaly



Frame	0	1	2	3	0	1	4	0	1	2	3	4
0	0	0	0	3	3	3	4	4	4	4	4	4
1		1	1	1	0	0	0	0	0	2	2	2
2			2	2	2	1	1	1	1	1	3	3

Frame	0	1	2	3	0	1	4	0	1	2	3	4
0	0	0	0	0	0	0	4	4	4	4	3	3
1		1	1	1	1	1	1	0	0	0	0	4
2			2	2	2	2	2	2	1	1	1	1
3				3	3	3	3	3	3	2	2	2

FIFO is not a stack algorithm. In certain cases, the number of page faults can actually increase when more frames are allocated to the process. In the example, there are 9 page faults for 3 frames and 10 page faults for 4 frames.

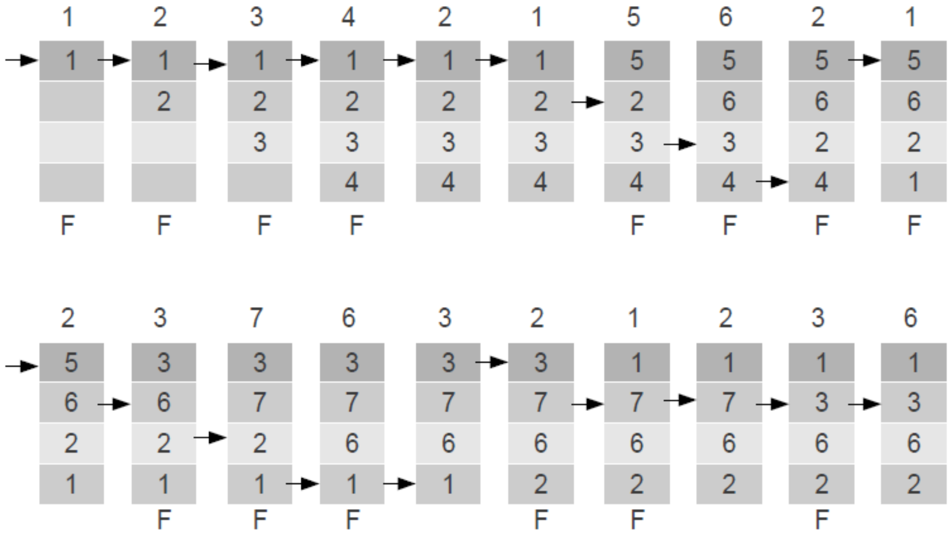
Examples - Page Replacement algorithms

- Given page reference string:
- **1,2,3,4,2,1,5,6,2,1,2,3,7,6,3,2,1,2,3,6**

- Compare the number of page faults and page references for LRU, FIFO and Optimal page replacement algorithms

FIFO (First In First Out)

- FIFO- On a page fault, the frame that has been in memory the longest (oldest) is replaced.

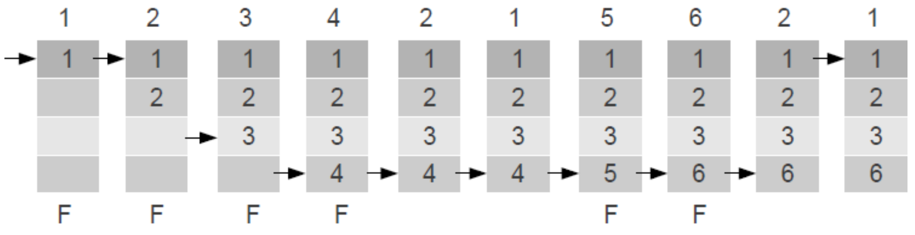


1	2	3	4	2	1	5	6	2	1
1	1	1	1	1	1	5	5	5	5
	2	2	2	2	2	2	6	6	6
		3	3	3	3	3	3	2	2
			4	4	4	4	4	4	1
F	F	F	F			F	F	F	F

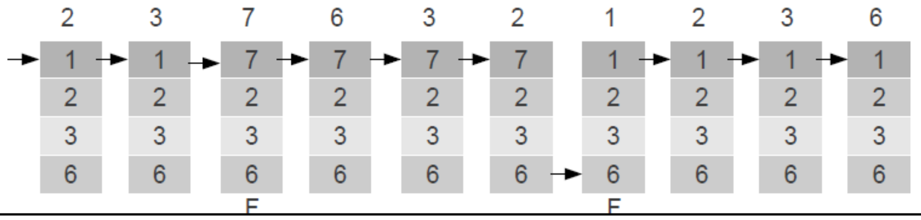
2	3	7	6	3	2	1	2	3	6
5	3	3	3	3	3	1	1	1	1
6	6	7	7	7	7	7	7	3	3
2	2	2	6	6	6	6	6	6	6
1	1	1	1	1	2	2	2	2	2
	F	F	F		F	F		F	

Optimal

- The Belady’s optimal algorithm is not a real replacement algorithm. It looks forward in time to see which frame to replace on a page fault. It gives us a frame of reference for a given static frame access sequence.



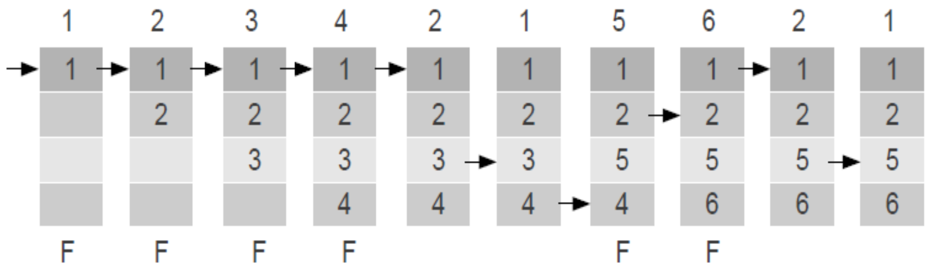
1	2	3	4	2	1	5	6	2	1
1	1	1	1	1	1	1	1	1	1
	2	2	2	2	2	2	2	2	2
		3	3	3	3	3	3	3	3
			4	4	4	5	6	6	6
F	F	F	F			F	F		



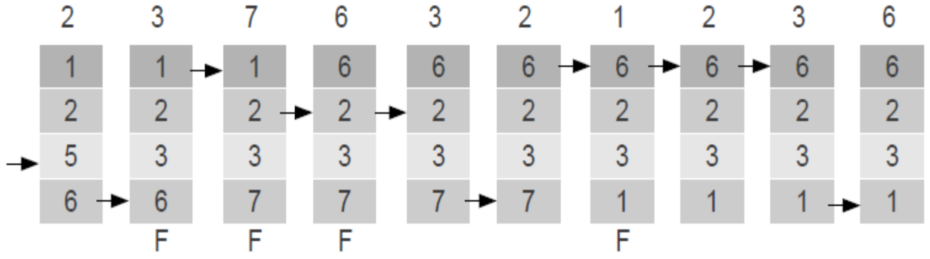
2	3	7	6	3	2	1	2	3	6
1	1	7	7	7	7	1	1	1	1
2	2	2	2	2	2	2	2	2	2
3	3	3	3	3	3	3	3	3	3
6	6	6	6	6	6	6	6	6	6
		F				F			

LRU (Least Recently Used)

- On a page fault, the frame that was least recently used page is replaced.



1	2	3	4	2	1	5	6	2	1
1	1	1	1	1	1	1	1	1	1
	2	2	2	2	2	2	2	2	2
		3	3	3	3	5	5	5	5
			4	4	4	4	6	6	6
F	F	F	F			F	F		

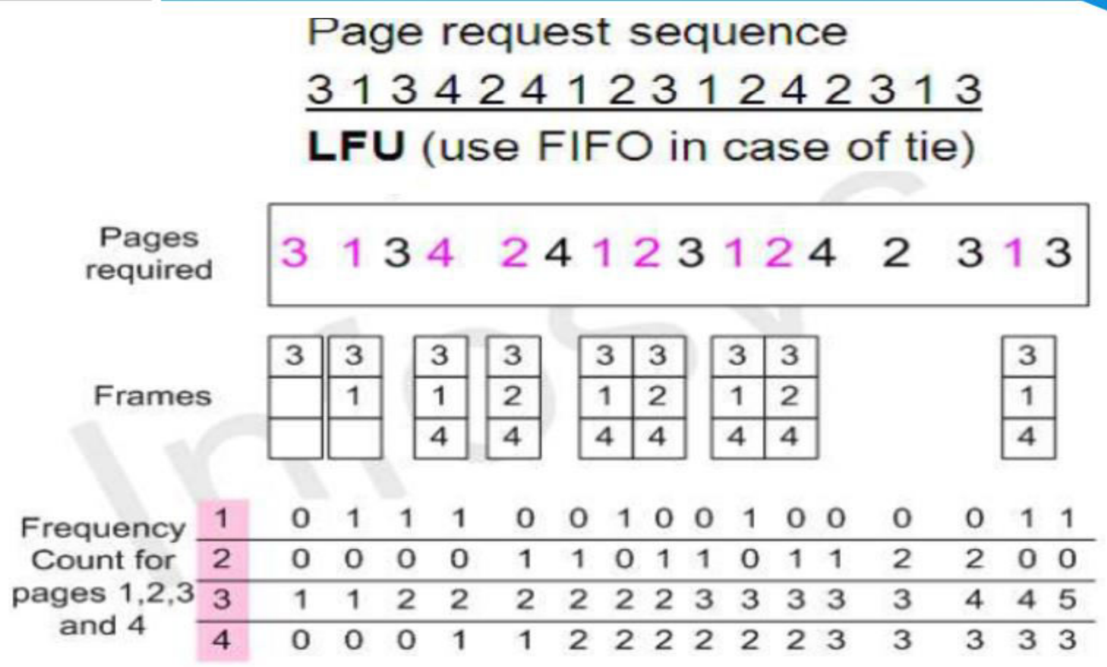


2	3	7	6	3	2	1	2	3	6
1	1	1	6	6	6	6	6	6	6
2	2	2	2	2	2	2	2	2	2
5	3	3	3	3	3	3	3	3	3
6	6	7	7	7	7	1	1	1	1
	F	F	F			F			

LRU

- Get a counter, maybe a 64-bit counter
- The counter is incremented after each instruction access and stored into the page entry at each reference
- Store the value of the counter in each entry of the page table (last access time to the page)
- When is time to remove a page, find the lowest counter value (this is the LRU page)
- Nice & good but expensive: it requires dedicated hardware

LFU (Least Frequently Used)



Page request sequence

3 1 3 4 2 4 1 2 3 1 2 4 2 3 1 3

LFU (use FIFO in case of tie)

Pages required: 3 1 3 4 2 4 1 2 3 1 2 4 2 3 1 3

Frequency Count for pages 1,2,3 and 4																
1	0	1	1	1	0	0	1	0	0	1	0	0	0	0	1	1
2	0	0	0	0	1	1	0	1	1	0	1	1	2	2	0	0
3	1	1	2	2	2	2	2	2	3	3	3	3	3	4	4	5
4	0	0	0	1	1	2	2	2	2	2	2	3	3	3	3	3

Circular FIFO(Clock)

- The set of frames candidate for replacement is considered as a circular buffer.
- When a page is replaced, a pointer is set to point to the next frame in buffer.
- A reference bit or use bit for each frame is set to 1 whenever:
  - a page is first loaded into the frame.
  - the corresponding page is referenced.
- When it is time to replace a page, the first frame encountered with the reference bit set to 0 is replaced:
  - During the search for replacement, each reference bit set to 1 is changed to 0.

❖ **Algorithm of Circular FIFO(Clock)**

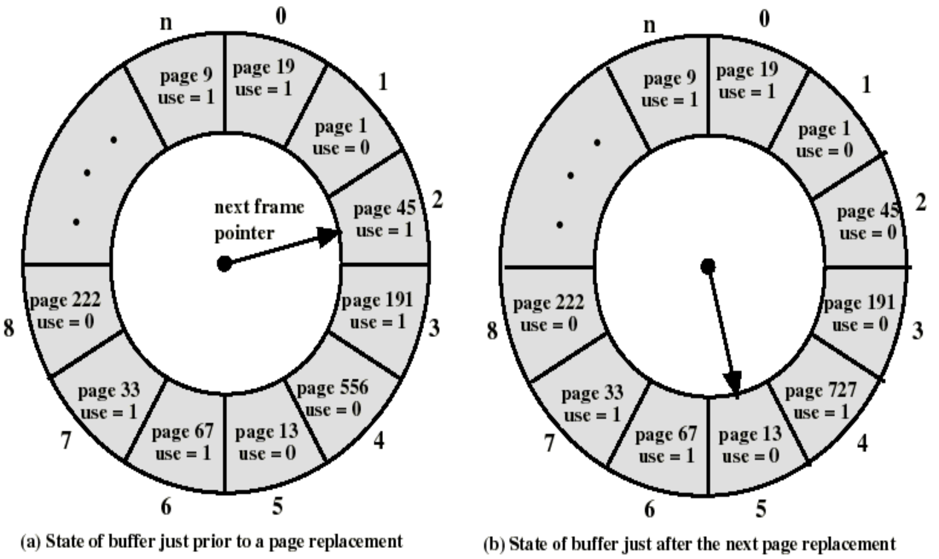
- Like FIFO but…
- Before throwing out a page checks the R bit / use bit:

✓If 0 remove it.

✓If 1 clear it.

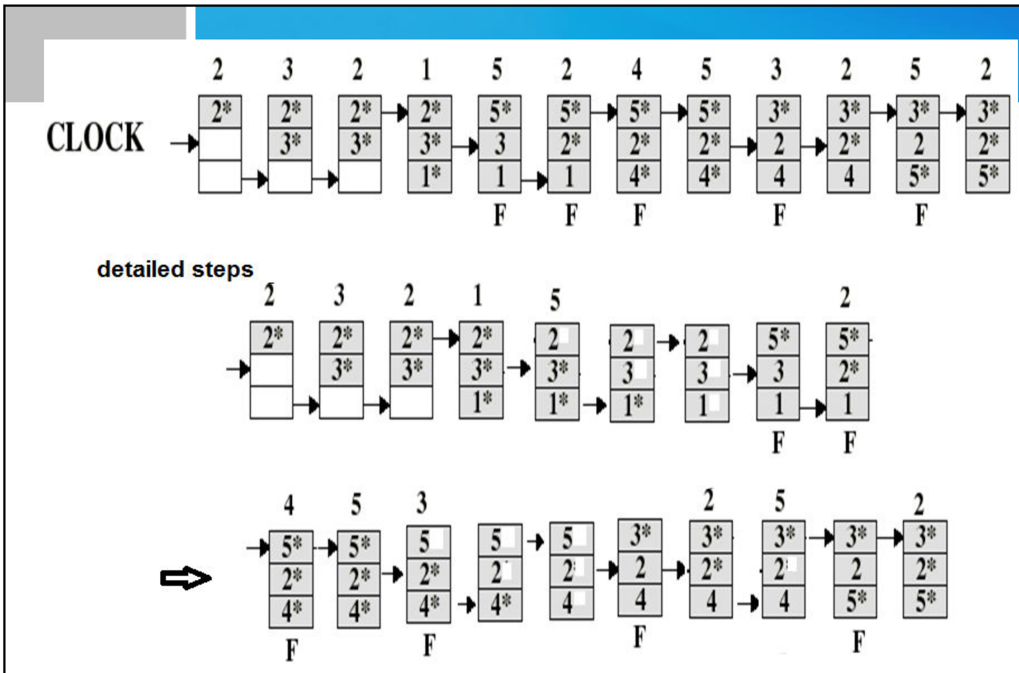
✓If all pages have R=1, eventually degenerates to FIFO

Clock Page-Replacement Algorithm



(a) State of buffer just prior to a page replacement

(b) State of buffer just after the next page replacement



CLOCK

2	3	2	1	5	2	4	5	3	2	5	2
2*	2*	2*	2*	5*	5*	5*	5*	3*	3*	3*	3*
	3*	3*	3*	3	2*	2*	2*	2	2*	2	2*
			1*	1	1	4*	4*	4	4	5*	5*
				F	F	F		F		F	

detailed steps

2	3	2	1	5				2
2*	2*	2*	2*	2	2	2	5*	5*
	3*	3*	3*	3*	3	3	3	2*
			1*	1*	1*	1	1	1
							F	F

4	5	3				2	5		2
5*	5*	5	5	5	3*	3*	3*	3*	3*
2*	2*	2*	2	2	2	2*	2	2	2*
4*	4*	4*	4*	4	4	4	4	5*	5*
F		F			F			F	

Asterisk indicates that the corresponding reference/use bit is set to 1.
The arrow indicates the current position of the pointer.