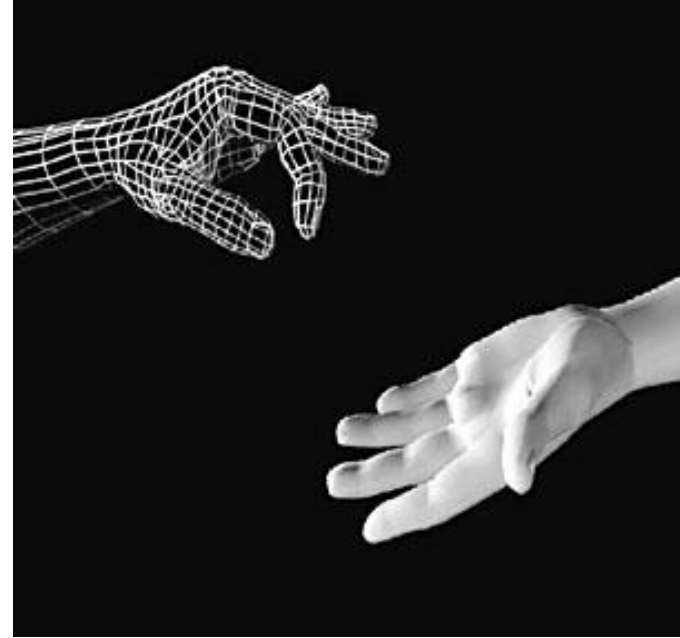


ENCE360

Operating Systems



Memory Management

Virtual Memory for More Memory

MOS Ch 4

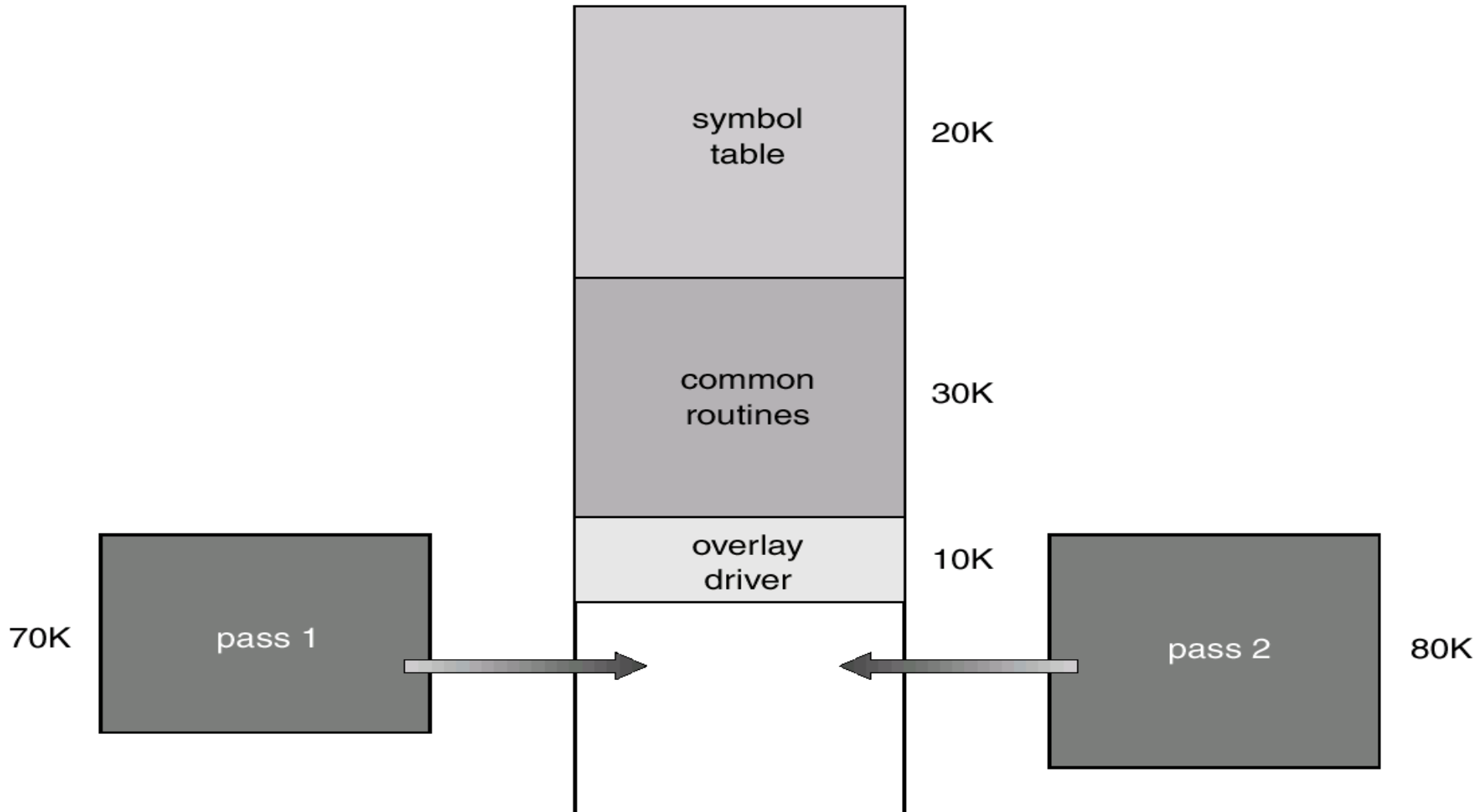


In the beginning...

- All addressing physical
- Single process
- Spatial separation between the operating system and user program
- Very little safeguards



Single user, single program





Problem 1: multiple programs

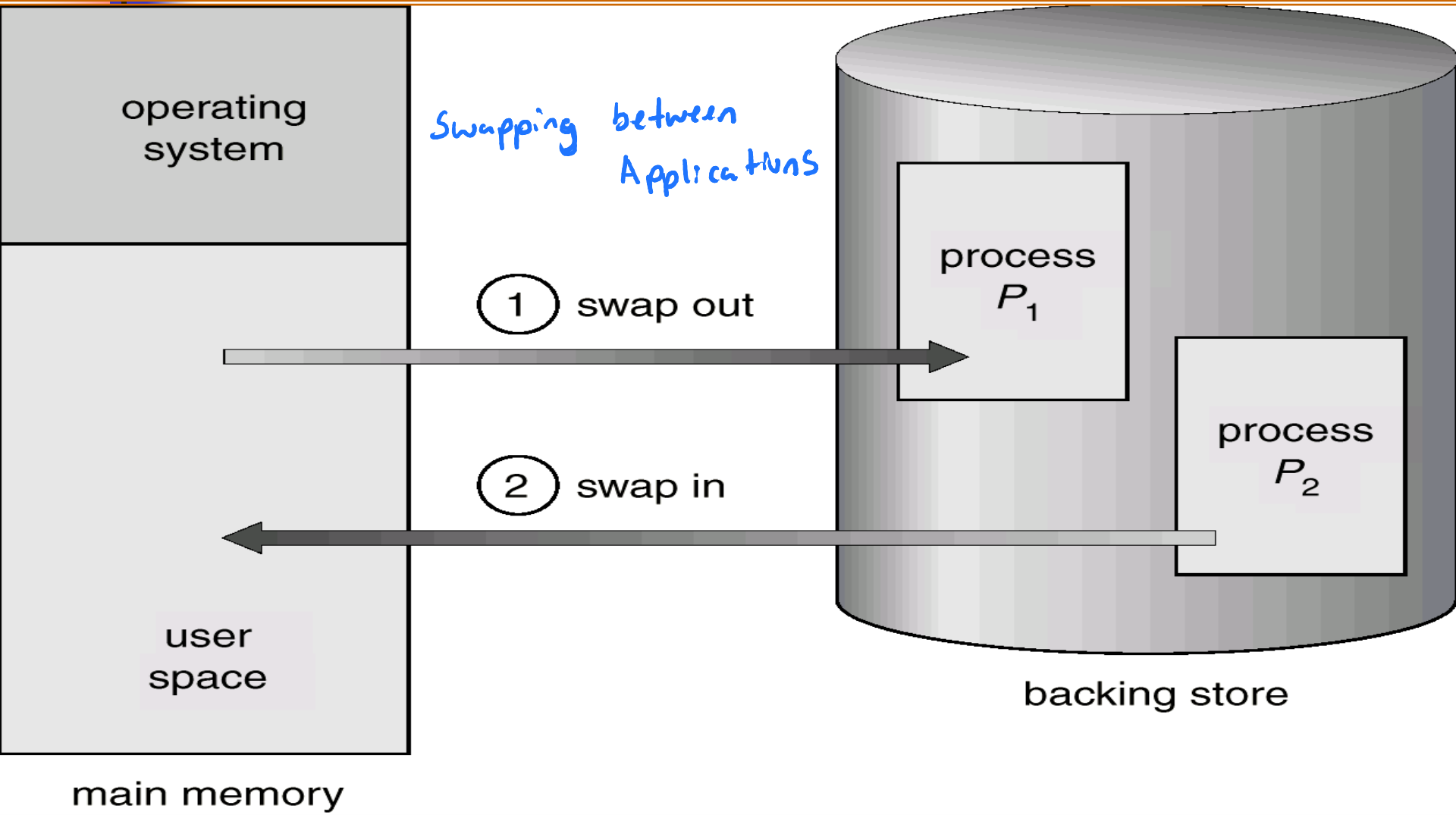
- More than one process running at a time
- Protection needed between address spaces
- Protection from operating system
- Address space small



Solution 1: Monoprogramming

- OS fixed into low addresses
- Programs compiled with *fixed* address space
- Poor solution:
 - Only certain combinations of programs can be run
 - External fragmentation
 - Operating system *not protected*

Multi-user, single program





Solution 2: base-limit address

- *Virtual* address space of 0 to $P_{\text{limit}}-1$
- Processor given P_{base} and P_{limit} when this process switched into context
- Hardware (CPU) converts:
 - Address $A_p = A_v + P_{\text{base}}$
 - Trap if $A_p \geq P_{\text{limit}}$ (memory violation fault)



Base-limit problems

- Wasteful:
 - entire process loaded (locality)
 - External fragmentation
- P_{limit} dictates total address space size
 - Write small programs, OR
 - Manage memory yourself (write out intermediate results)

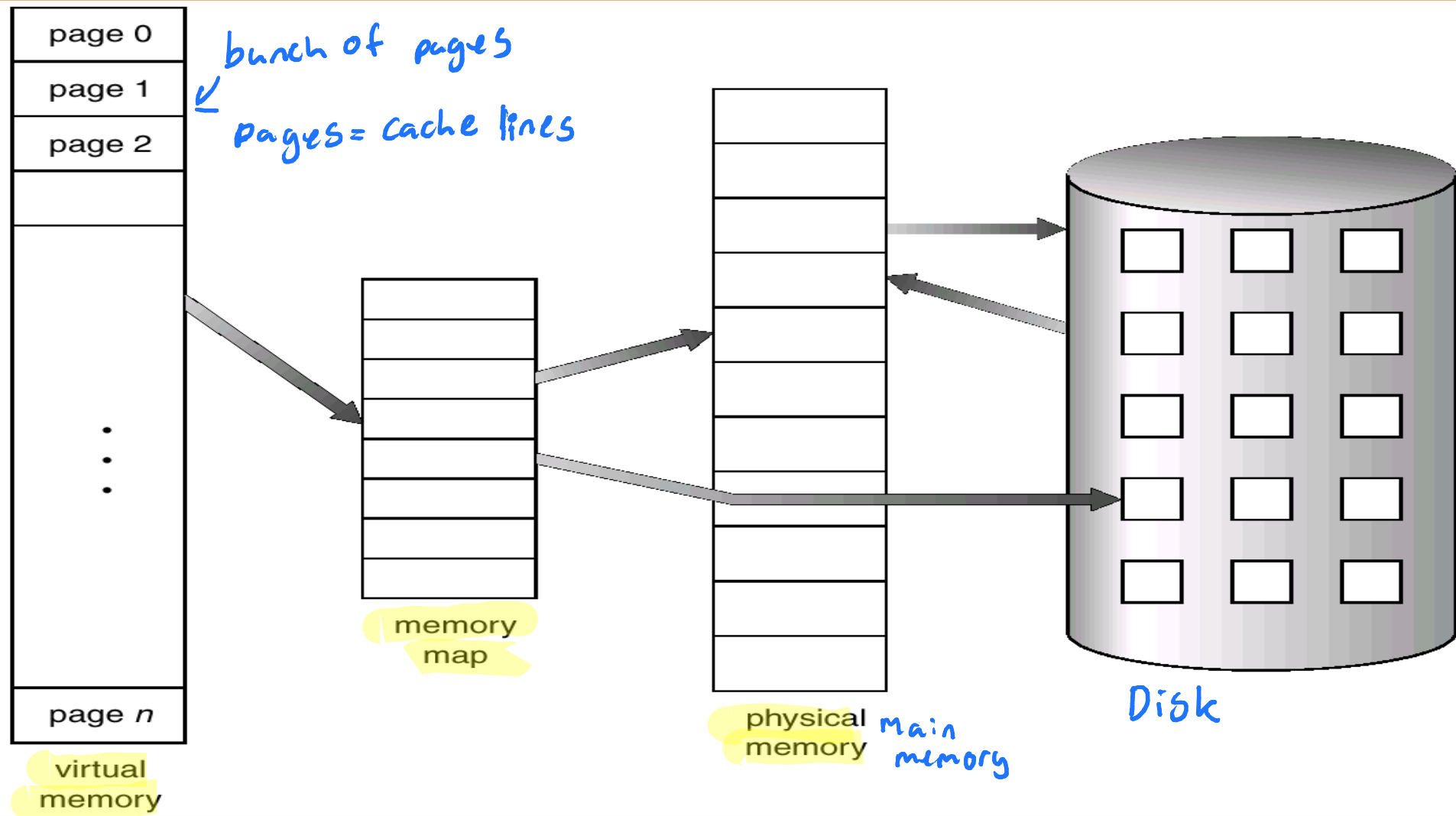
- Break memory in a bunch of Blocks
- non-continuous
- Blocks = pages in memory

The "Manchester" Solution

Hardware Support

- Divorce virtual address from physical memory
- Allow arbitrary address contiguous space
(e.g. 32bits - 4GB)
(e.g. 64bits - 18 million Terabytes (18 Exabytes))
↑
virtual memory
- Virtual address space *does not really exist anywhere*
 - Physical memory mapped to currently used portion of address space

Virtual Memory That is Larger Than Physical Memory

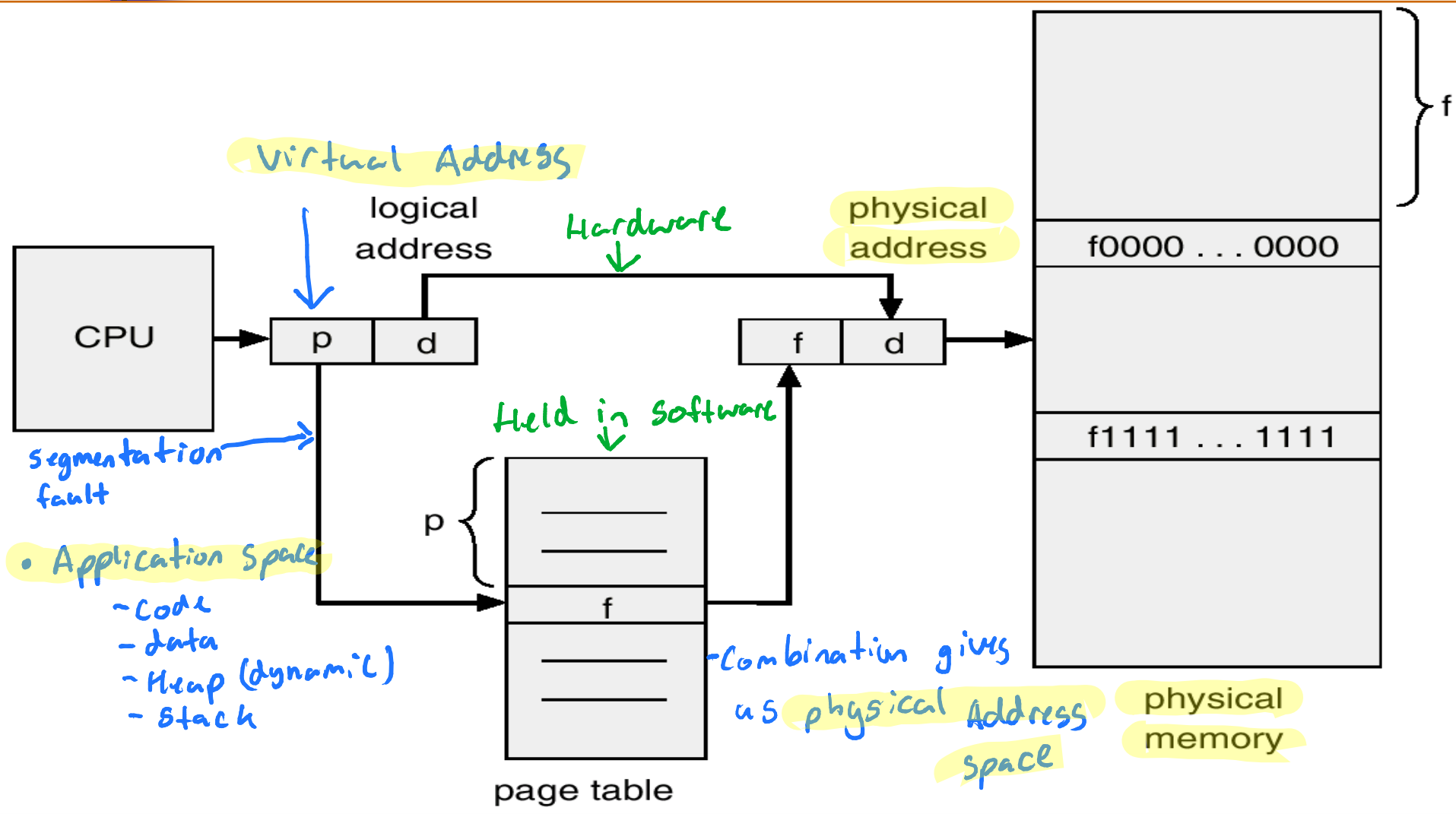




Virtual Memory Paging

- Memory (V and P) divided up into pages of 512 to 8192 bytes
- MMU: V page mapped to physical using *page table*
 - E.g. specialised direct-mapped cache (no tag)
 - Data in cache is *physical page number*
 - Add page number to low bytes of V address (byte number) to get physical address
- Treats physical memory as a fully-associative cache of the virtual address space (page table instead of tags)

Example: Linear Page Table





Page Table Entry (PTE) Contents

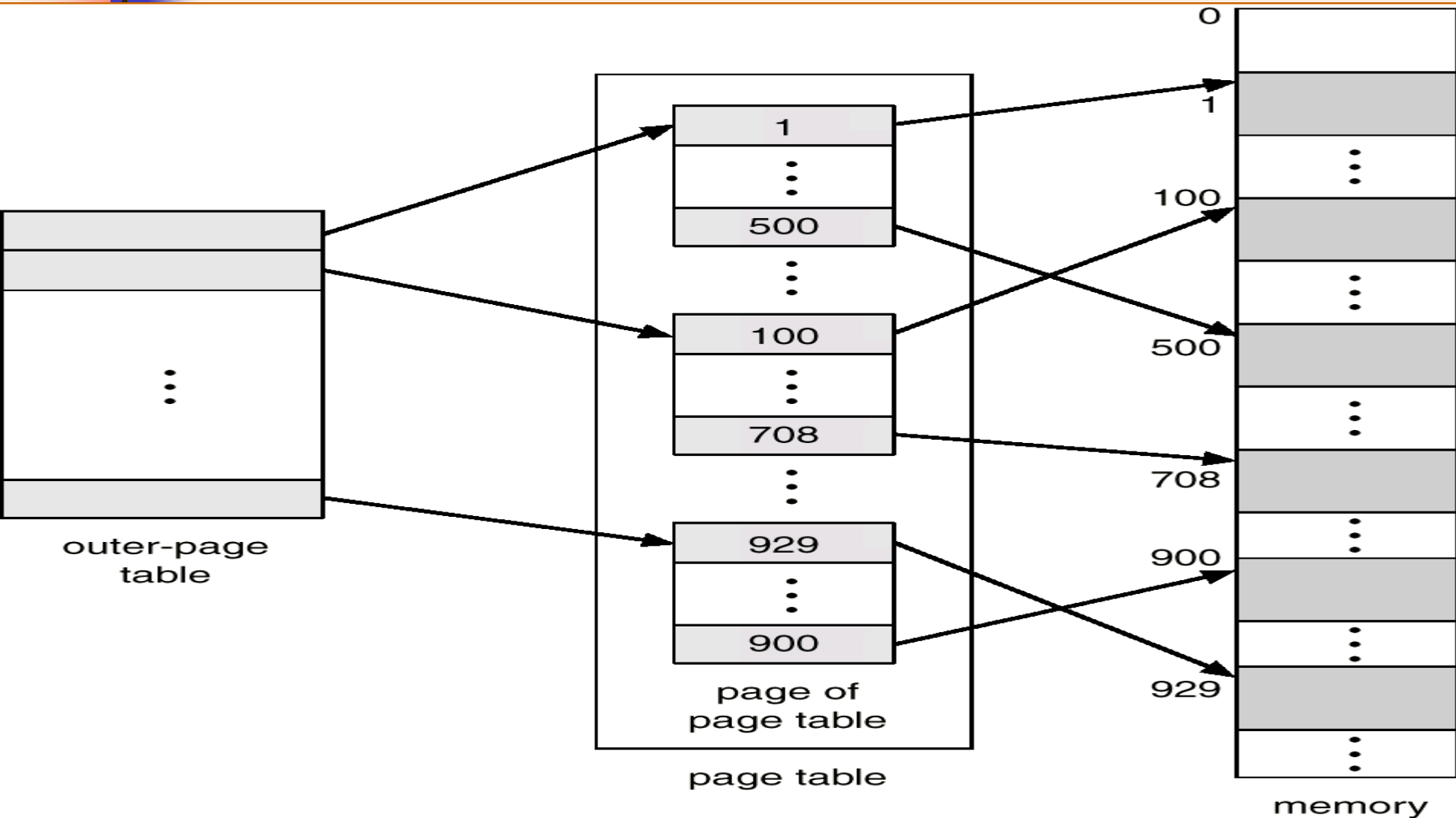
- Usually 32 bits:
 - Page frame number
 - Present/absent? bit *is the page frame in memory?*
 - Modified? bit
 - Referenced? bit
 - Caching Disabled? bit
- OS records where other pages can be found



More problems

- Linear page table not feasible for big address spaces
- Use a *tree*: “hierarchical page table”
 - Most virtual pages are empty
 - Index the V address space
 - “Page table descriptor” nodes
 - Include nodes for resident pages only
 - Locality of reference assumed

Hierarchical page table

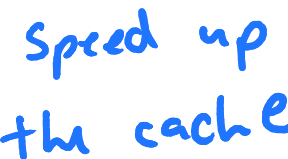




Translation Lookaside Buffer

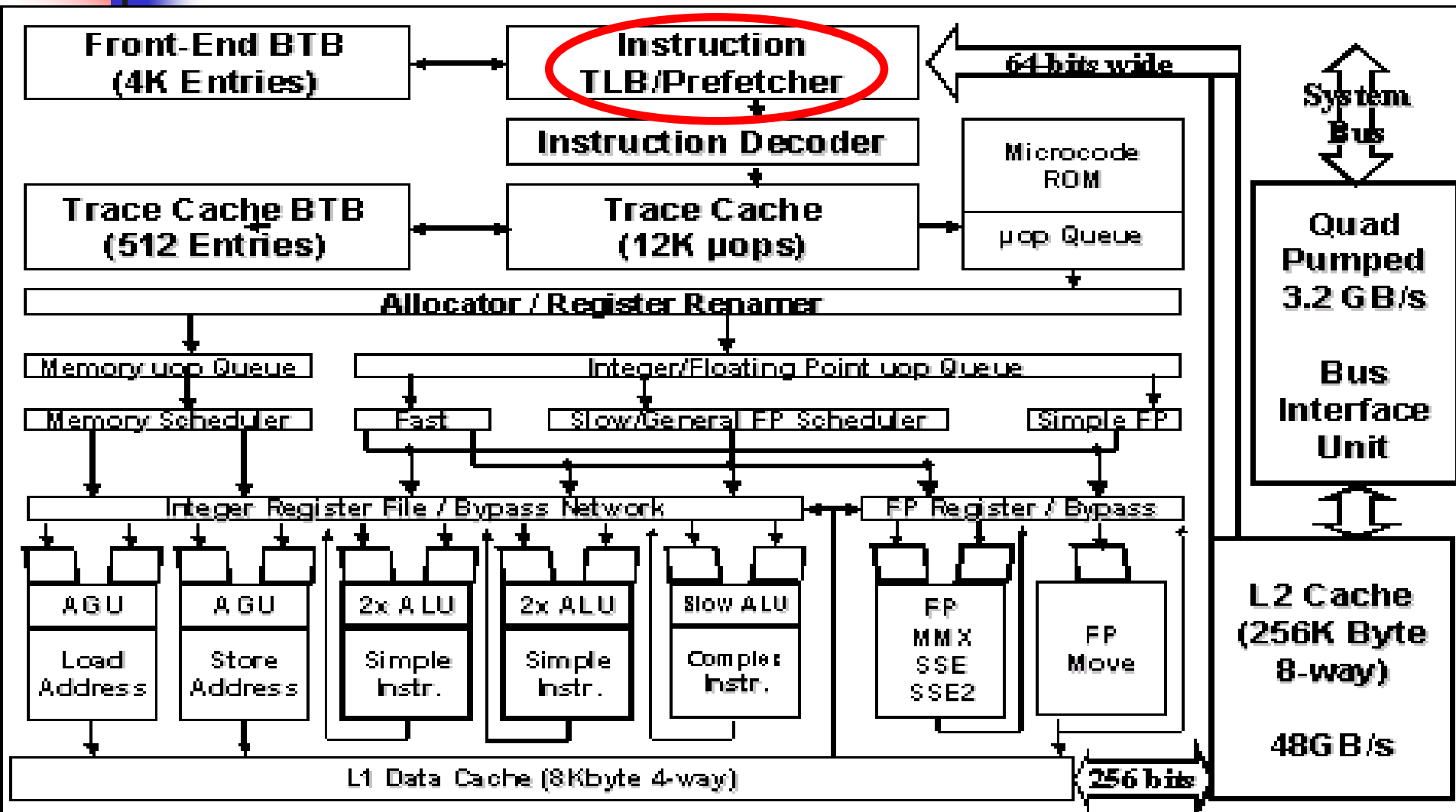
- Cache of translations, stored in CPU/MMU (often the same)
- Look in TLB first. If failed, go to page table
- Needs to know context:
 - Invalidate on context switch OR
 - Store context too (just like cache)
 - Usually split Instruction/Data
- Small, fully associative (parallel search), hardwired (99% hit)

A diagram illustrating a step function. It shows a horizontal axis with a vertical line at the point of discontinuity. To the left of the line, the function is represented by a red rectangle. To the right of the line, the function is represented by a blue rectangle. The function is zero for $x < 0$ and $x > 1$. The function value is 1 for $0 < x < 1$ and 0 for $x < 0$ and $x > 1$. The function is not defined at $x = 0$ and $x = 1$.



- page table can be swapped out to disk

Pentium





And if that's not enough...

- UltraSPARC: translation storage buffer (TSB)
 - If TLB (on CPU) miss, go to TSB
 - TSB controlled by OS, helped by hardware
 - If TSB lookup fails, go to page table (no hardware help)
- OS has to help!
 - Manage TSB according to exact format
 - Manage page table however it sees fit



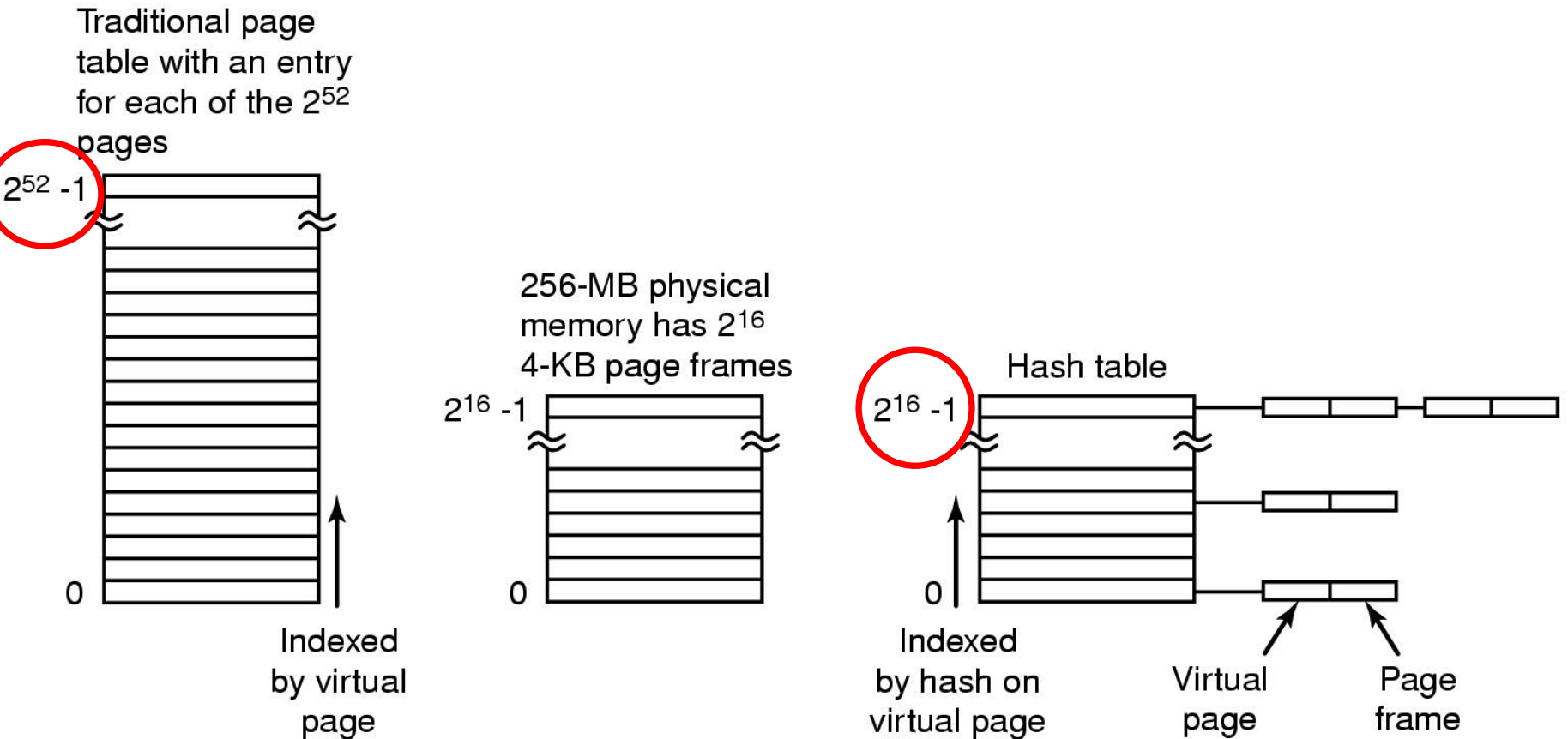
Still not fast enough?

Virtual addresses getting *huge* (physical memory relatively smaller - e.g. 32bit 2GB RAM with 4B page table entries in 4K page size => **4GB page table per process**)

Invert the page table: **one page table for all processes**

- One entry per *physical* page
- If TLB misses, look up inverted page table
- If not there (so not in memory) page fault occurs
- Hashing on virtual page number
- Decreases memory needed to store each page table, but increases time needed to search the table when a page reference occurs.

Inverted Page Table

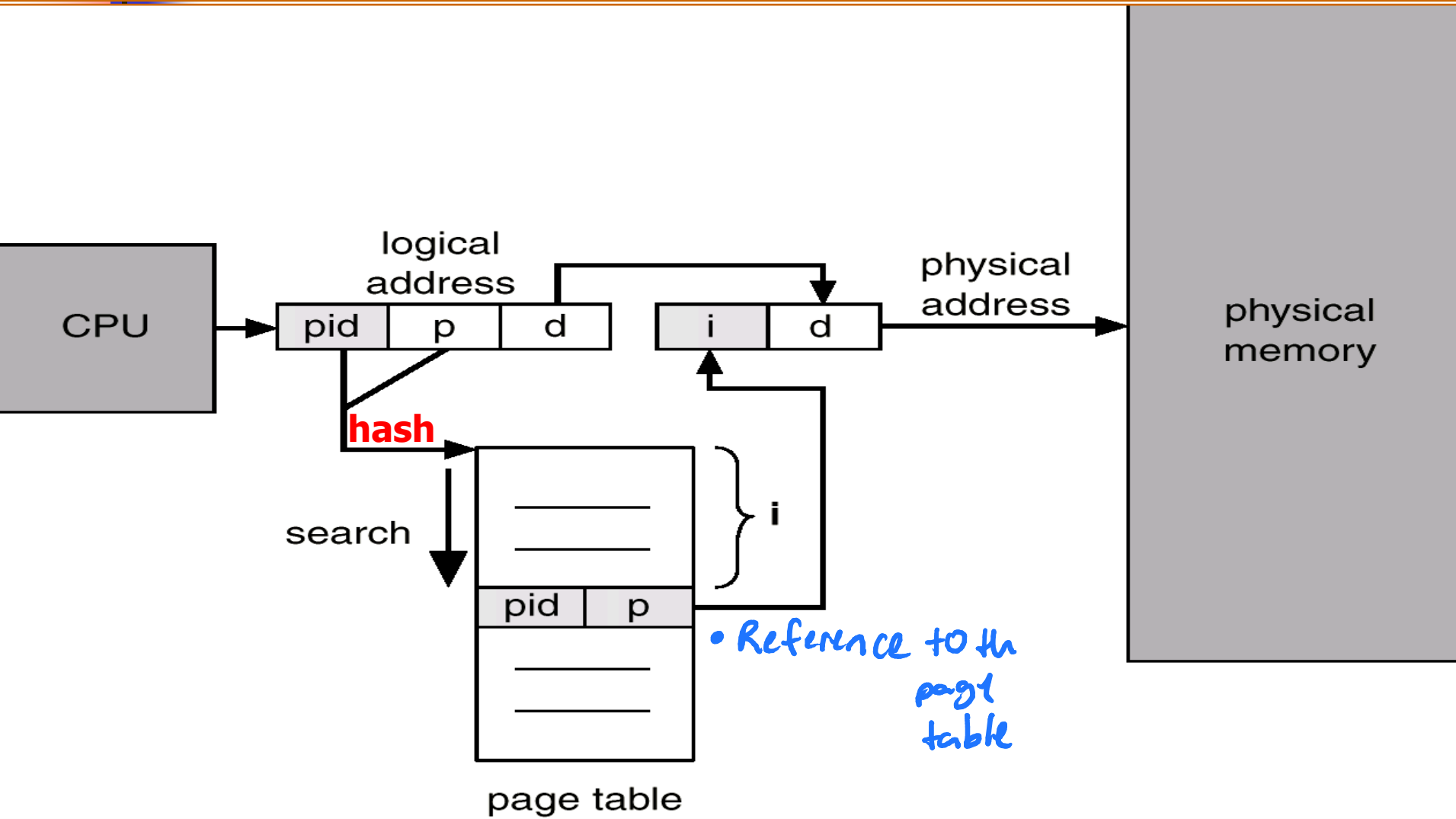


Comparison of a traditional **page table per process** (above left) with one single **inverted page table for all processes** (right)

Inverted Page Table (hash table)

one single global inverted page table for all processes

(pid=process ID, p=virtual page number, i=physical frame address, d=offset)

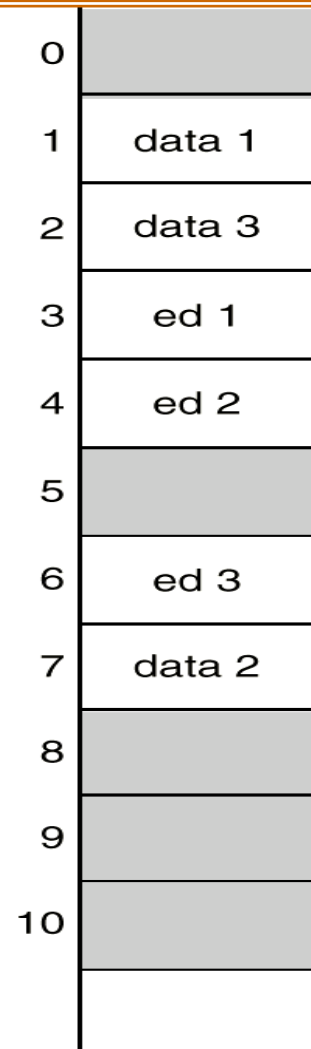
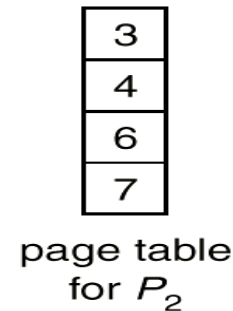
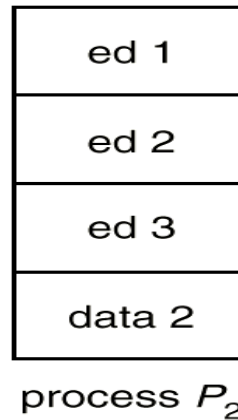
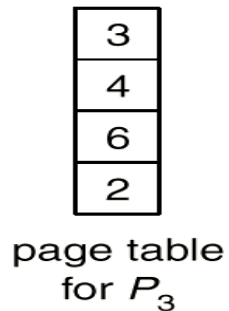
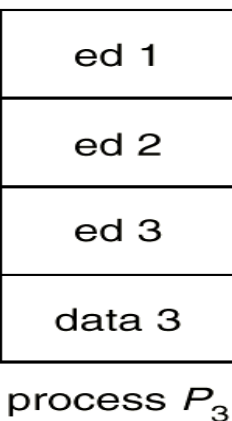
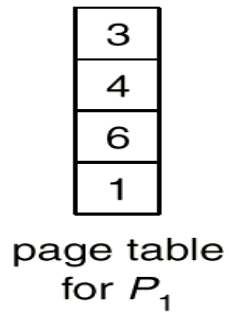




Sharing memory pages

- If multiple processes need same memory pages, map to *same* physical memory
- Problem if one process wants to modify its copy
- Windows was: “copy-on-write”
 - Share same copy initially
 - On write, copy to new location, update one copy only
 - Also called “lazy evaluation”

Shared pages example





Address space division

- Some memory should not be cached (e.g. device registers)
- Some memory should not be mapped to virtual addresses (e.g. kernel)
- E.g. MIPS R2000
 - 2GB cached, mapped user segment
 - 512MB cached, unmapped kernel segment
 - 512MB uncached, unmapped kernel segment
 - 1GB cached, mapped kernel segment



Segmentation

- Virtual address needs managing by OS
- Solution: “segmentation”
 - Multiple virtual address spaces
 - Handled by the hardware (MMU)
 - Separate into logical units:
 - Program
 - Variables
 - Stack
 - Heap, etc.



Page Faults

- Only a subset of the V address space can fit in memory (working set)
- Access to non-resident address (page) signals a “page fault” exception = *reference to something on disk*
- Instruction “rolled back”
- Page fetched from disk, instruction executed again
- Other work (processes) done during the wait



Page Replacement Algorithms

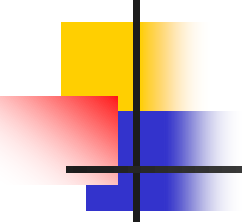
- Page fault forces choice
 - which page must be removed
 - make room for incoming page
- Modified page must first be saved
 - unmodified just overwritten
- Better not to choose an often used page
 - will probably need to be brought back in soon

Optimal Page Replacement Algorithm

- page least likely to be used
- Analysis before the process runs
- Operating system code

- Replace page needed at the farthest point in future
 - Optimal but unrealizable
- Estimate by ...
 - logging page use on previous runs of process
 - although this is impractical

Not Recently Used Page Replacement Algorithm

- 
- Each page has Reference bit, Modified bit
 - bits are set when page is referenced, modified
 - Pages are classified
 1. not referenced, not modified
 2. not referenced, modified
 3. referenced, not modified
 4. referenced, modified
 - NRU removes page at random
 - from lowest numbered non empty class

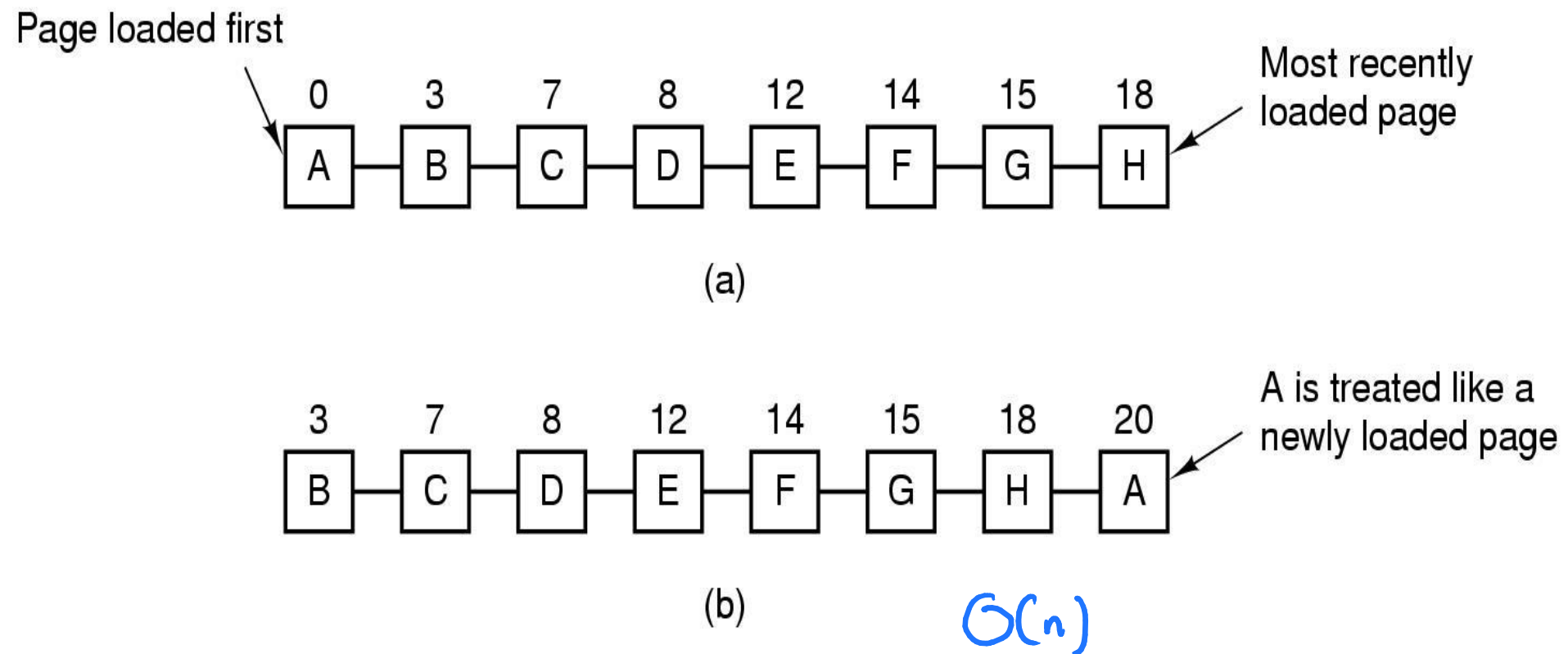
FIFO Page Replacement Algorithm

First page = oldest page (might be used / might not be)

Poor Technique

- Maintain a linked list of all pages
 - in order they came into memory
- Page at beginning of list replaced
- Disadvantage
 - page in memory the longest may be often used

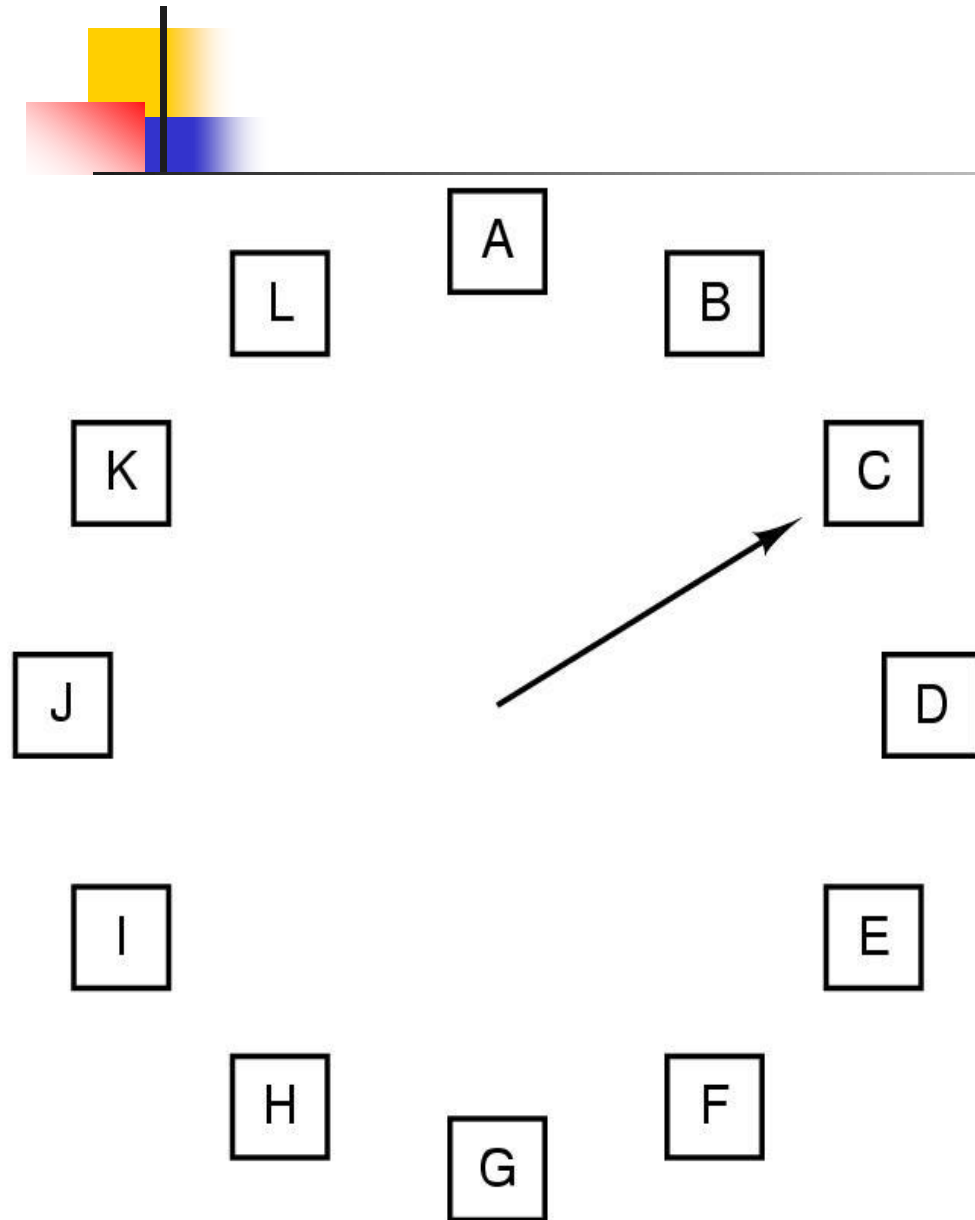
Second Chance Page Replacement Algorithm



■ Operation of a second chance

- pages sorted in FIFO order
- Page list if fault occurs at time 20, A has *R* bit set (numbers above pages are loading times)

The Clock Page Replacement Algorithm

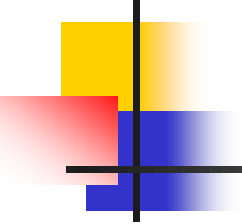


When a page fault occurs, the page the hand is pointing to is inspected. The action taken depends on the R bit:

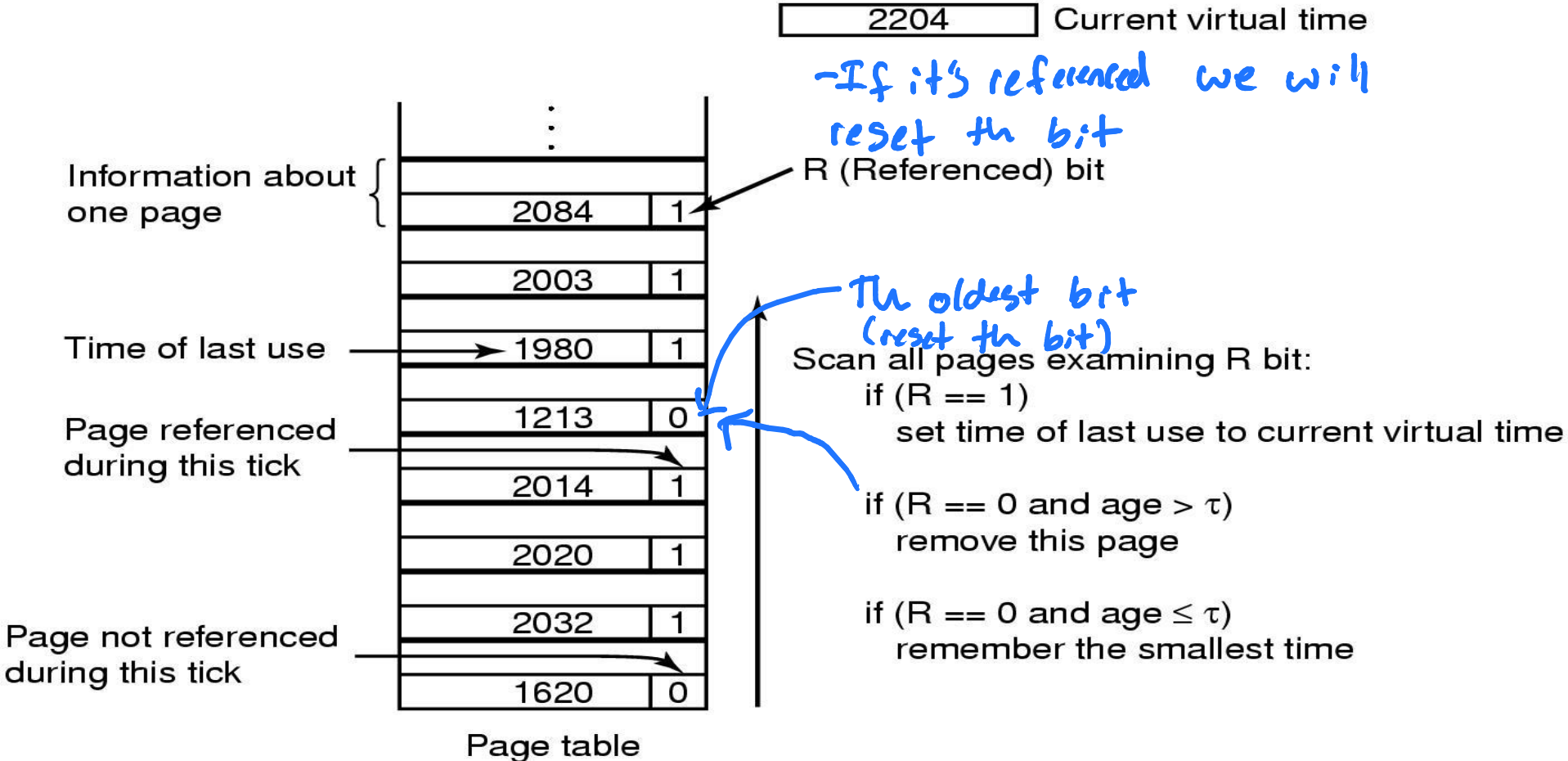
R = 0: Evict the page

R = 1: Clear R and advance hand

Least Recently Used (LRU)

- 
- Assume pages used recently will be used again soon
 - throw out page that has been unused for longest time
 - Must keep a linked list of pages
 - most recently used at front, least at rear
 - update this list every memory reference !!
 - Alternatively keep counter in each page table entry
 - choose page with lowest value counter
 - periodically zero the counter

The Working Set Page Replacement Algorithm

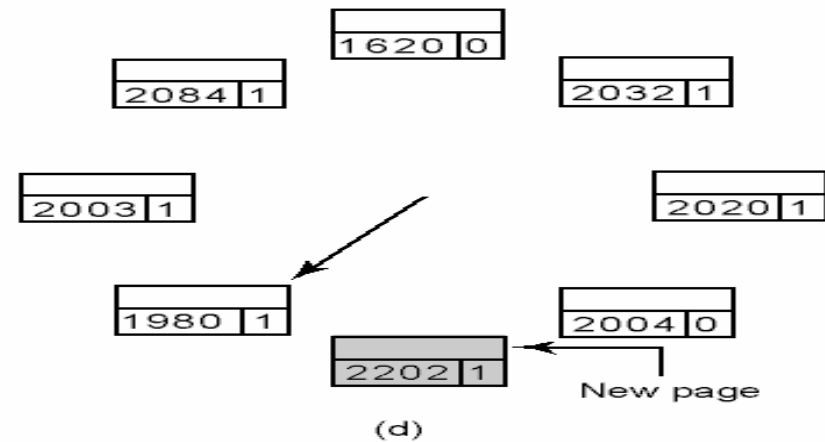
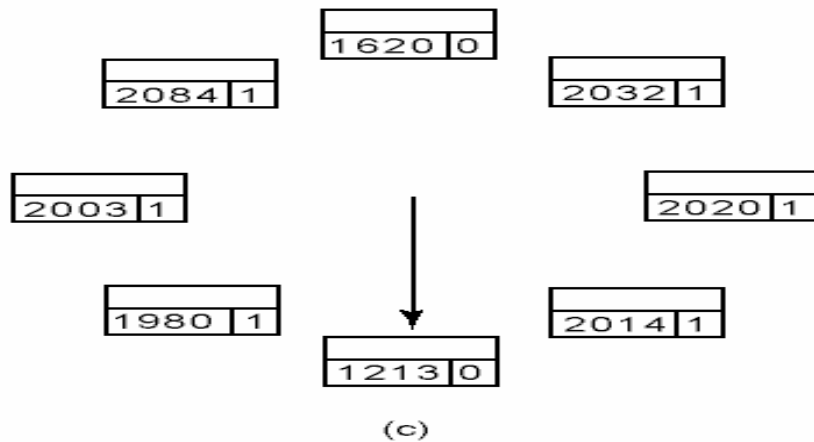
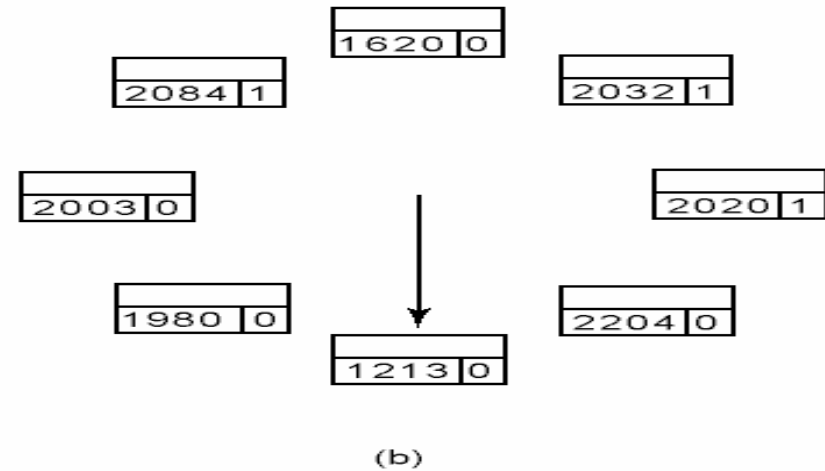
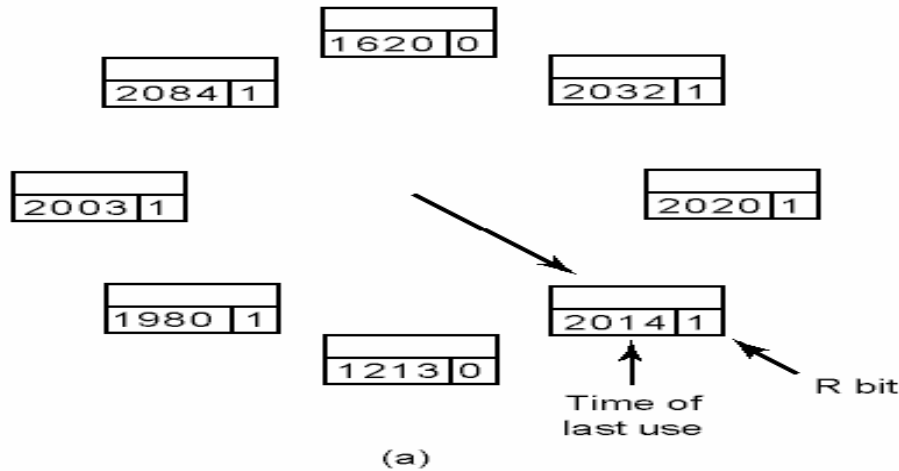


- The working set is the set of pages used by the k most recent memory references
- $w(k,t)$ is the size of the working set at time, t

The WSClock Page Replacement Algorithm

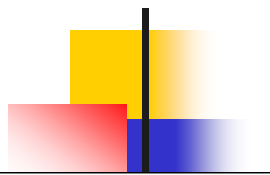
2204

 Current virtual time



Operation of the WSClock algorithm

Review of Page Replacement Algorithms



Algorithm	Comment
Optimal	Not implementable, but useful as a benchmark
NRU (Not Recently Used)	Very crude
FIFO (First-In, First-Out)	Might throw out important pages
Second chance	Big improvement over FIFO
Clock	Realistic
LRU (Least Recently Used)	Excellent, but difficult to implement exactly
NFU (Not Frequently Used)	Fairly crude approximation to LRU
Aging	Efficient algorithm that approximates LRU well
Working set (FIFO + WSClock)	Somewhat expensive to implement
WSClock	Good efficient algorithm

Modeling Page Replacement Algorithms

Belady's Anomaly

All pages frames initially empty

	0	1	2	3	0	1	4	0	1	2	3	4	
Youngest page		0	1	2	3	0	1	4	4	4	2	3	3
			0	1	2	3	0	1	1	1	4	2	2
Oldest page				0	1	2	3	0	0	0	1	4	4
	P	P	P	P	P	P	P				P	P	

9 Page faults

(a)

	0	1	2	3	0	1	4	0	1	2	3	4	<u>memory</u>
Youngest page		0	1	2	3	3	3	4	0	1	2	3	4
			0	1	2	2	2	3	4	0	1	2	3
Oldest page				0	1	1	1	2	3	4	0	1	2
					0	0	0	1	2	3	4	0	1
	P	P	P	P				P	P	P	P	P	P

10 Page faults

↙ # increased physical
memory space

(b)

- FIFO with 3 page frames
- FIFO with 4 page frames
- *P*s show which page references show page faults

Page Fault Handling

needs to happen very very fast (due to lots of possible misses)



1. Hardware traps to kernel

2. General registers saved

3. OS determines which virtual page needed

4. OS checks validity of address, seeks page frame

5. If selected frame is dirty, write it to disk

6. OS brings schedules new page in from disk

7. Page tables updated

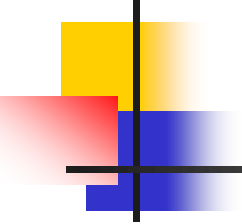
- Faulting instruction backed up to when it began

6. Faulting process scheduled

7. Registers restored

- Program continues

Cleaning Policy

- 
- Need for a background process, paging daemon
 - periodically inspects state of memory
 - When too few frames are free
 - selects pages to evict using a replacement algorithm
 - It can use same circular list (clock)
 - as regular page replacement algorithm, but with different pointer

Implementation Issues

Operating System Involvement with Paging

Four times when OS involved with paging:

1. Process creation
 - determine program size
 - create page table
2. Process execution
 - MMU reset for new process
 - TLB flushed
3. Page fault time
 - determine virtual address causing fault
 - swap target page out, needed page in
4. Process termination time
 - release page table, pages



Swapping

- Separate to paging
- Entire context (process) unloaded from memory, stored on disk
- Totally OS controlled
- Reduces page thrashing but *huge* performance hit for offending process



Page size

Small page size:

- Advantages

- less internal fragmentation
- better fit for various data structures, code sections
- less unused program in memory

- Disadvantages

- programs need many pages, larger page tables



Page size

- Tradeoff between speed and efficiency:
 - Larger page size = more read per disk access
 - Smaller page size means less wastage
- Depends on *use*:
 - Scattered memory addresses: smaller page size is better
 - Contiguous memory addresses: larger page



OS designer issues

- Page replacement
- Memory division
 - sharing
 - caching
- Page/Swap file management
- Controlling thrashing (careful memory access)
- TLB replacement (e.g. MIPS R2000)

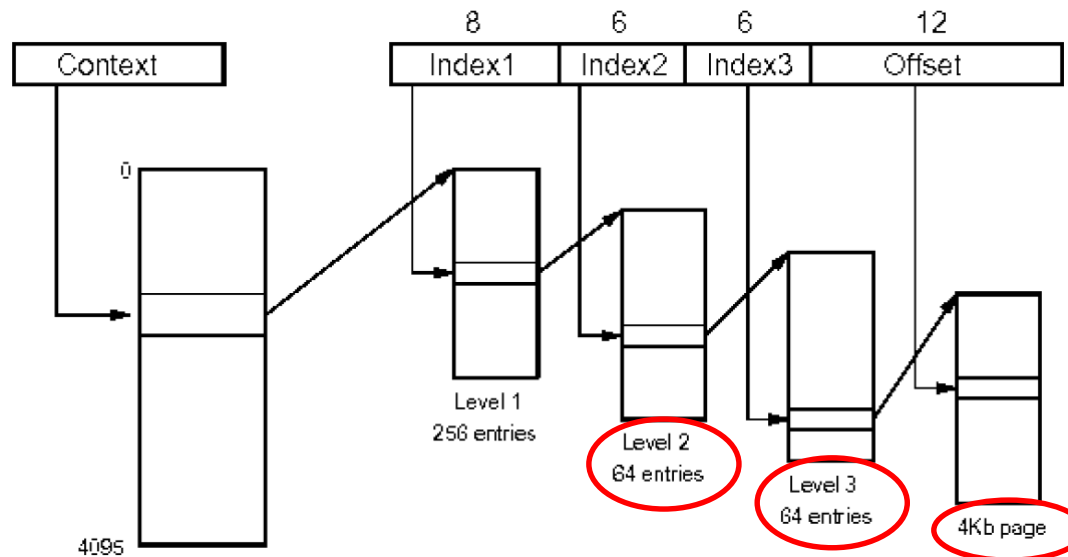


Summary

- Paging allows memory to do more:
 - Exceed physical memory size
 - Run multiple processes efficiently
- Complex set of caches etc to keep speed up
- Hardware and OS both heavily involved
- Like caching, OS can be optimised to minimise page faulting

Example Exam Question

The **32-bit** SPARC architecture uses a **3-level page table** as illustrated below.



- (a) List *one* advantage and *one* disadvantage of using this type of page table.
- (b) Calculate **how many bytes are required for the page table** of a process on a SPARC system with a **5MByte** text segment, a **20MByte** data segment, and a **16MByte** stack. Assume that the text segment starts at 0x0, that the data segment follows the text segment, and that the stack grows down from 0xFFFFFFFF. **Level 1 nodes are 1024 bytes** each, **level 2 and 3 nodes are 256 bytes** each. *Show your working.*



Solution: (a)

Advantages:

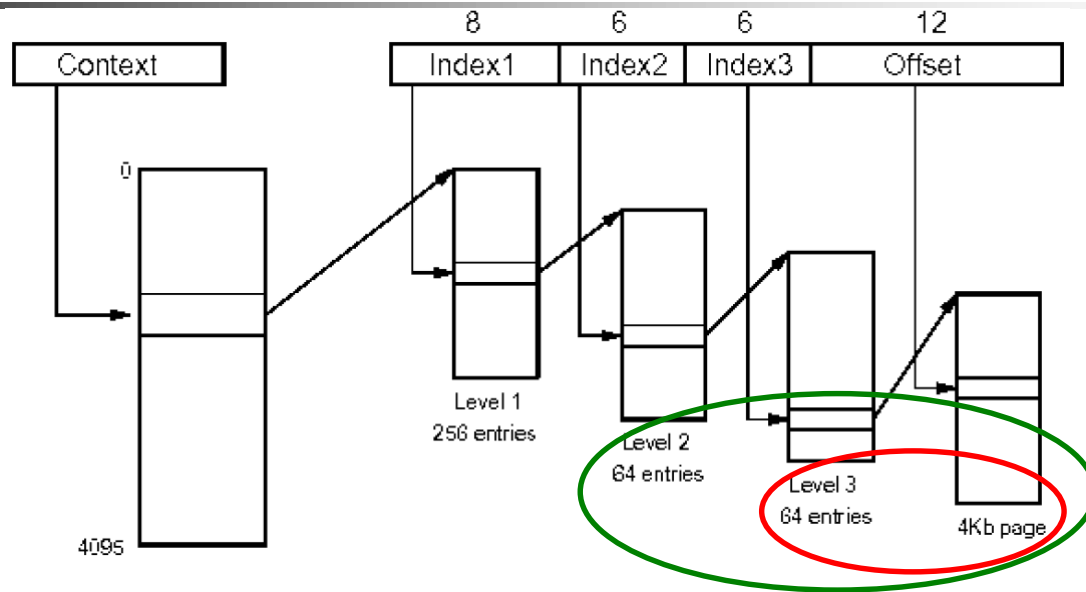
- Takes up less space than a 2 or 1 level page table
- Not all of the address space needs to be in memory at once, only the pages that are required.

Disadvantages:

- Slower to look up than a 2 or 1 page table.
- Can become large and complex. (A better option may be to invert the page table so there is one entry for each physical frame.)

Solution: (b)

How much memory can a L1, L2, L3 page/node address?



- A level 3 page addresses $64 \times 4\text{KB}$ pages
= **256KB** (in the red ellipse above)
- A level 2 page addresses $64 \times 64 \times 4\text{KB}$ pages
= **16MB** (in the green ellipse)
- A level 1 page addresses all 32bit virtual memory (**4GB**)

Solution: (b)

How many L1, L2, L3 pages/nodes are needed?

Level 1 pages

Need only *one* level 1 page/node

Level 2 pages

For 5MB text + 20MB data (from 0x0 to 25MB)

Need 25MB/**16MB**(L2 page) rounded up

= two level 2 pages (which can access **16MB** each)

For 16MB stack

Need only one level 2 (**16MB**) page

Need total of *three* level 2 pages/nodes

Solution: (b)

How many L1, L2, L3 pages/nodes are needed?



Level 3 pages

For 5MB text + 20MB data

Need 25MB/**256KB**(L3 page)

= 100 level 3 pages (which can access **256KB** each)

For 16MB stack


Need 16MB/**256KB**(L3 page)

= 64 level 3 pages

Need total of 164 level 3 pages/nodes

Solution: (b)

What is the total memory used?



The question stated that “Level 1 nodes are **1024** bytes each, level 2 and 3 nodes are **256** bytes each”.

Page table size (total memory used):

1 L1 node + **3** L2 nodes + **164** L3 nodes

1 x **1024** + **3** x **256** + **164** x **256** bytes

= 1024 + 768 + 41984 bytes

= **43,776 bytes**

Virtualisation



Virtualisation: feature of multi-core chips
- designed to run multiple operating systems simultaneously.

Teloportation: Virtual Machine is temporarily stopped and then resumed on a different computer

Products: Parallels, **VirtualBox**, Virtual Iron, Virtual PC, Hyper-V, **Vmware** KVM, QEMU, Adeos, Mac-on-Linux, Win4BSD, Win4Lin Pro, vBlade