

Memory Management

Chapter 8

Outline

- Memory management hw & sw architectures
- Page replacement algorithms

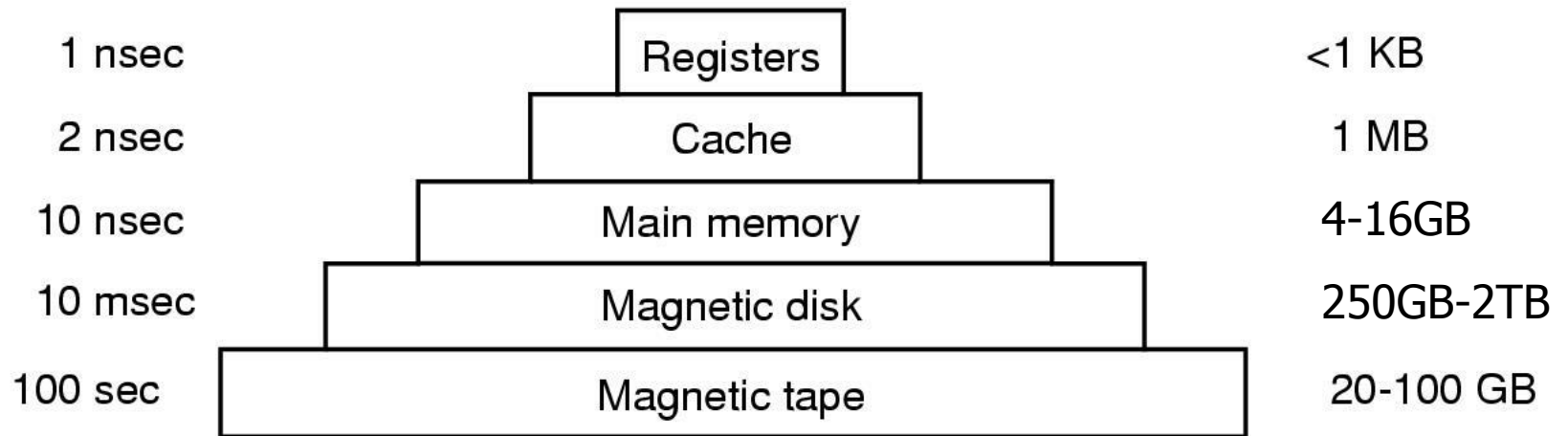
Memory Management

- Ideally programmers want memory that is
 - large
 - Fast
 - cheap
 - non volatile
- Memory hierarchy
 - small amount of fast, expensive memory – cache
 - some medium-speed, medium price main memory
 - gigabytes of slow, cheap disk storage
- Memory manager handles the memory hierarchy

Computer Hardware Review (memory architecture)

Typical access time

Typical capacity

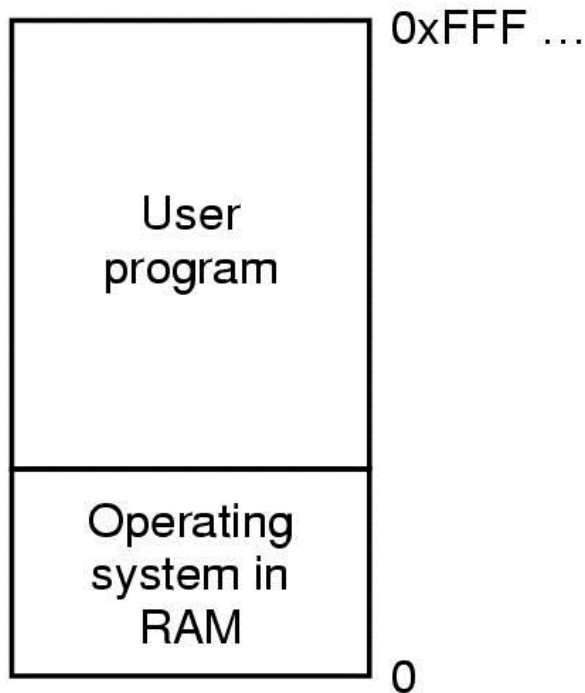


- Typical memory hierarchy
 - numbers shown are rough approximations and can get outdated quickly as technology progresses

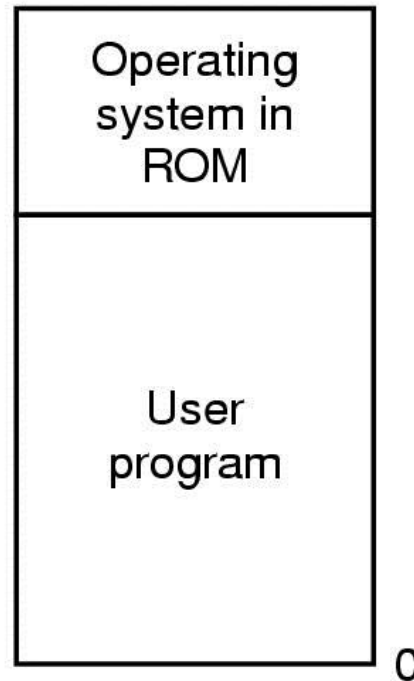
Basic Memory Management

Monoprogramming without Swapping or Paging

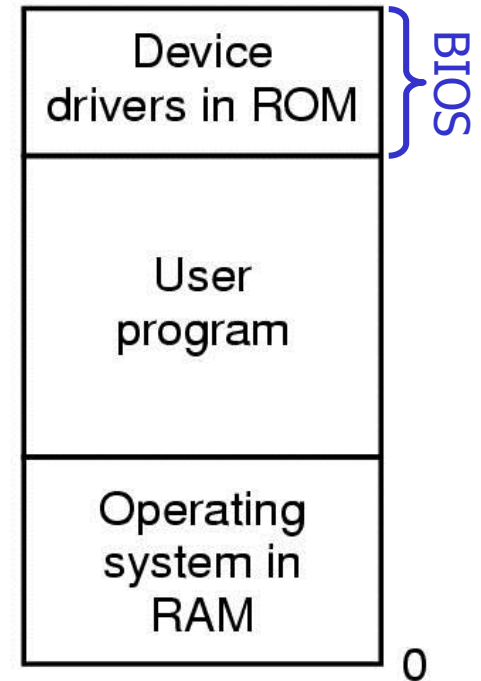
Old mainframes



Palmtop computers



Early PC's

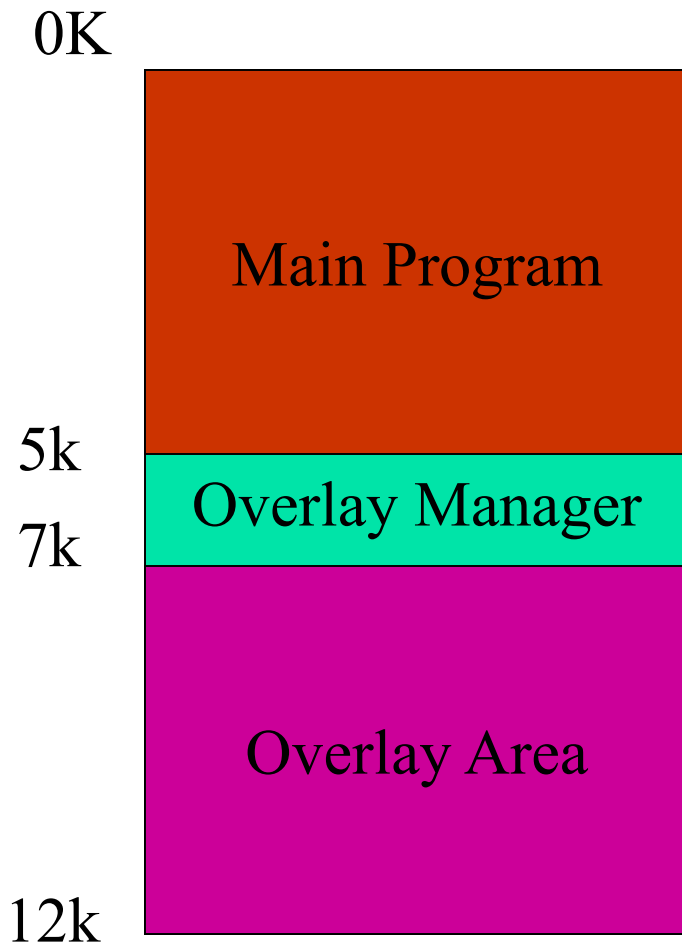


Three simple ways of organizing memory
- an operating system with one user process

Overlaying

- Done in old times (before virtual memory systems)
- User specifies what procedures are in which overlay (often support in assembly language)
- Overlays are program segments that do not have to be in memory at the same time.
- Overlay manager:
 - Piece of the OS
 - if there is a procedure in overlay_i , it loads overlay_i into memory, runs it, and returns control to main.
- **Goal:** so that a program can be larger than physical memory.
- Too cumbersome for the programmer.

Overlaying



Overlays

- Modern systems use virtual memory, so do not need overlays
- But some embedded systems (particularly real-time embedded systems) may still use overlays in order to have more deterministic response times than paging.
 - Example: the Space Shuttle *Primary Avionics System Software (PASS)* uses programmed overlays.

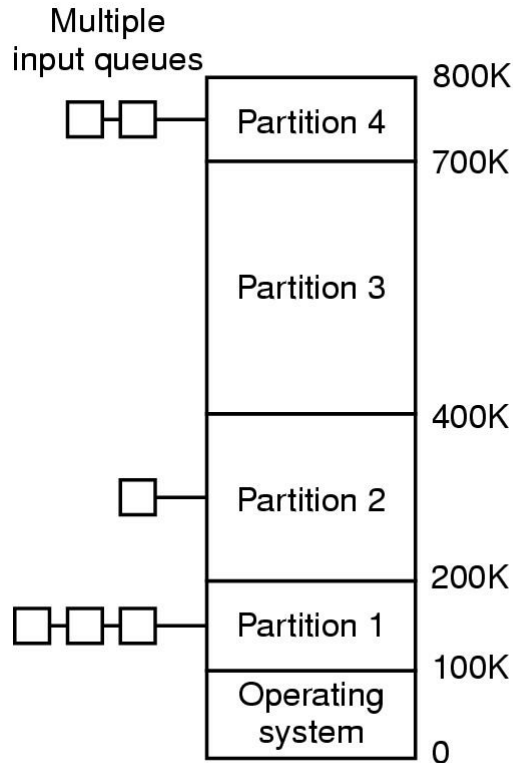
Multiprogramming with fixed memory partitions

- Typically applies to batch systems.
- I/O devices idle while CPU works.
- When job written/read from disk, CPU idle.
- Fixed partitions: attempt to pipeline jobs.
 - Fixed partitions assigned to different jobs.
- Cycle goes:
 - Read next job
 - CPU runs current job (CPU utilized)
 - At the same time the previous job is written (IO utilized)
- Therefore, CPU and IO could be utilized simultaneously.

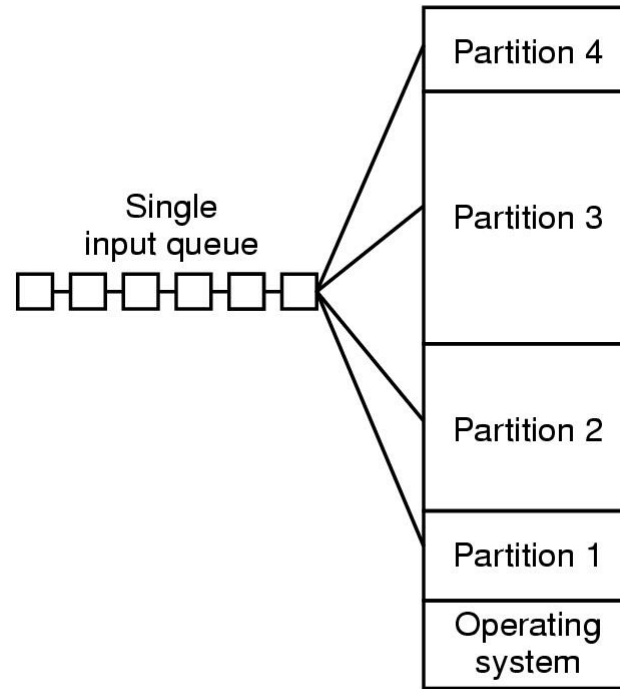
Multiprogramming with fixed partitions

- Used in old mainframe batch computers. IBM OS/360
- Fixed memory partitions created by the operator in the morning and computer ran all day with these partitions.
- Called Multiprogramming with Fixed number of Tasks (OS/MFT)

Multiprogramming with Fixed Partitions



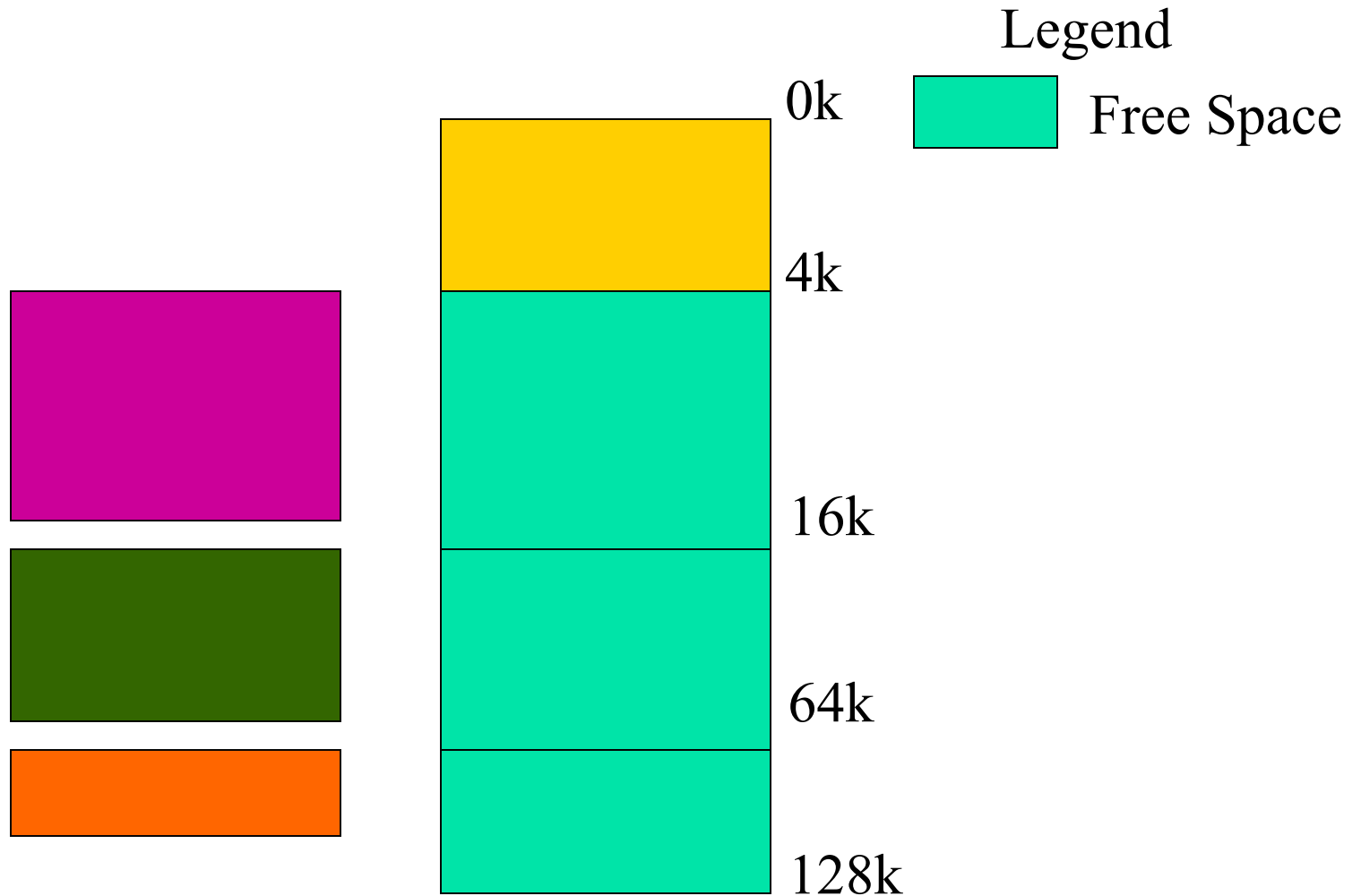
(a)



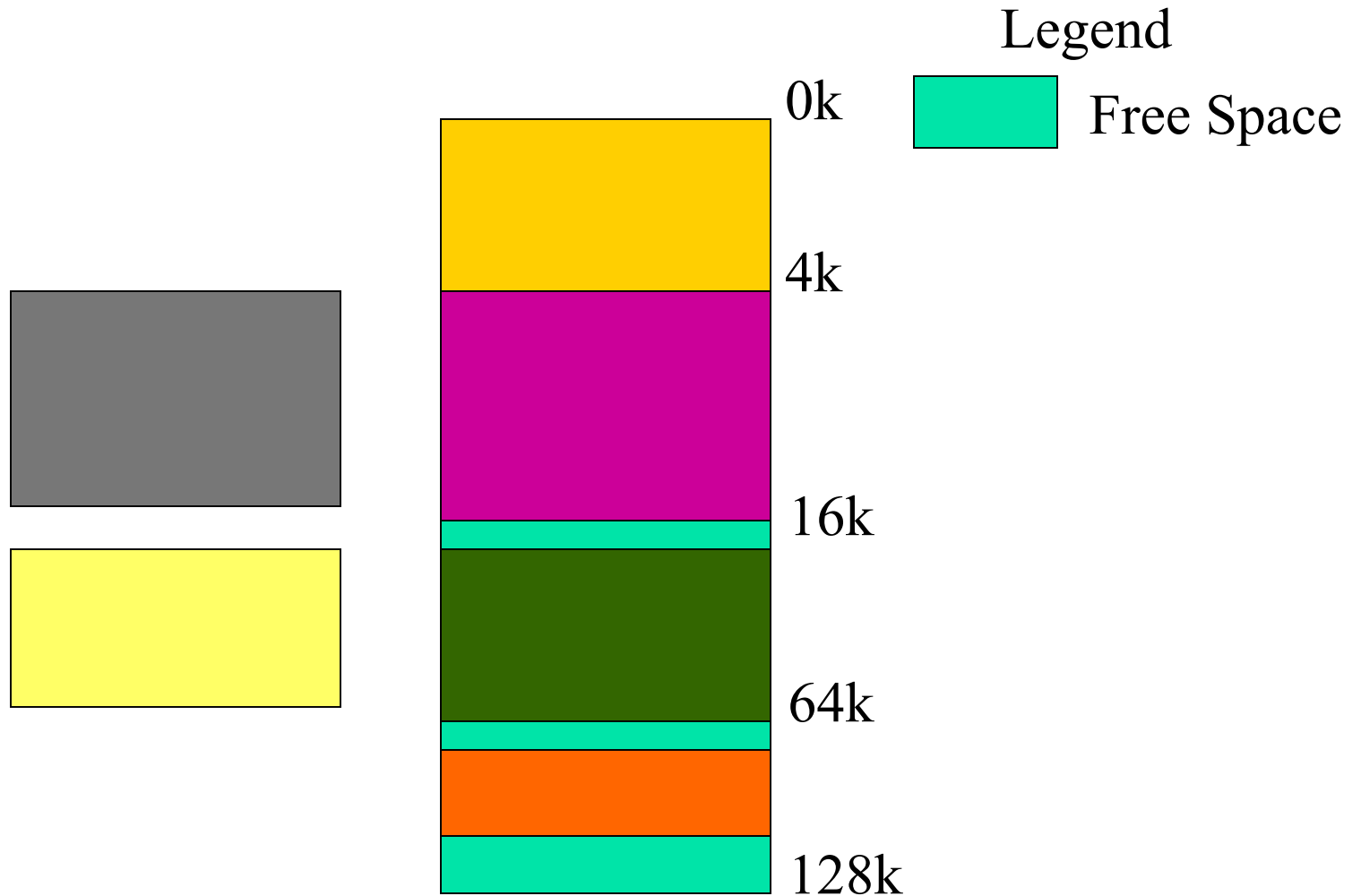
(b)

- Fixed memory partitions
 - (a) separate input queues for each partition
 - (b) single input queue (utilizes partitions a little better)

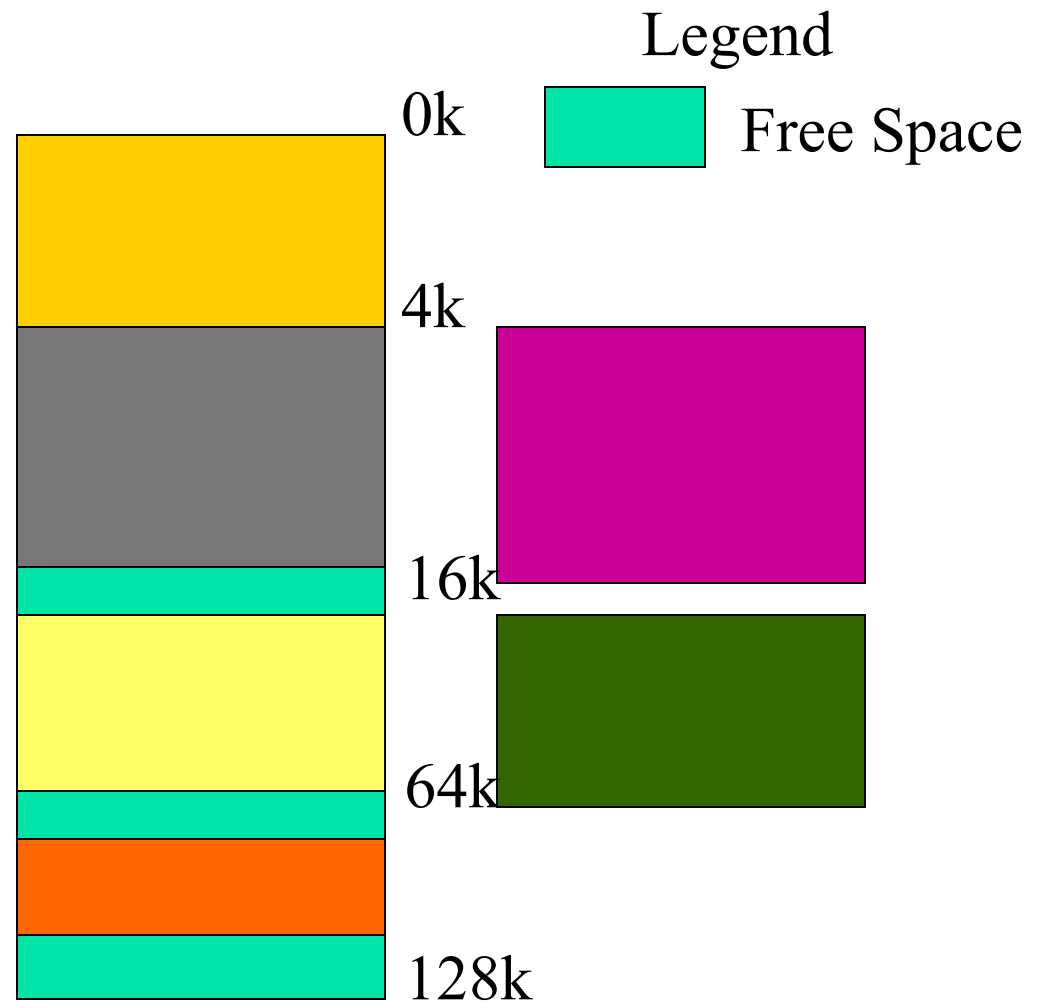
Fixed Partitions



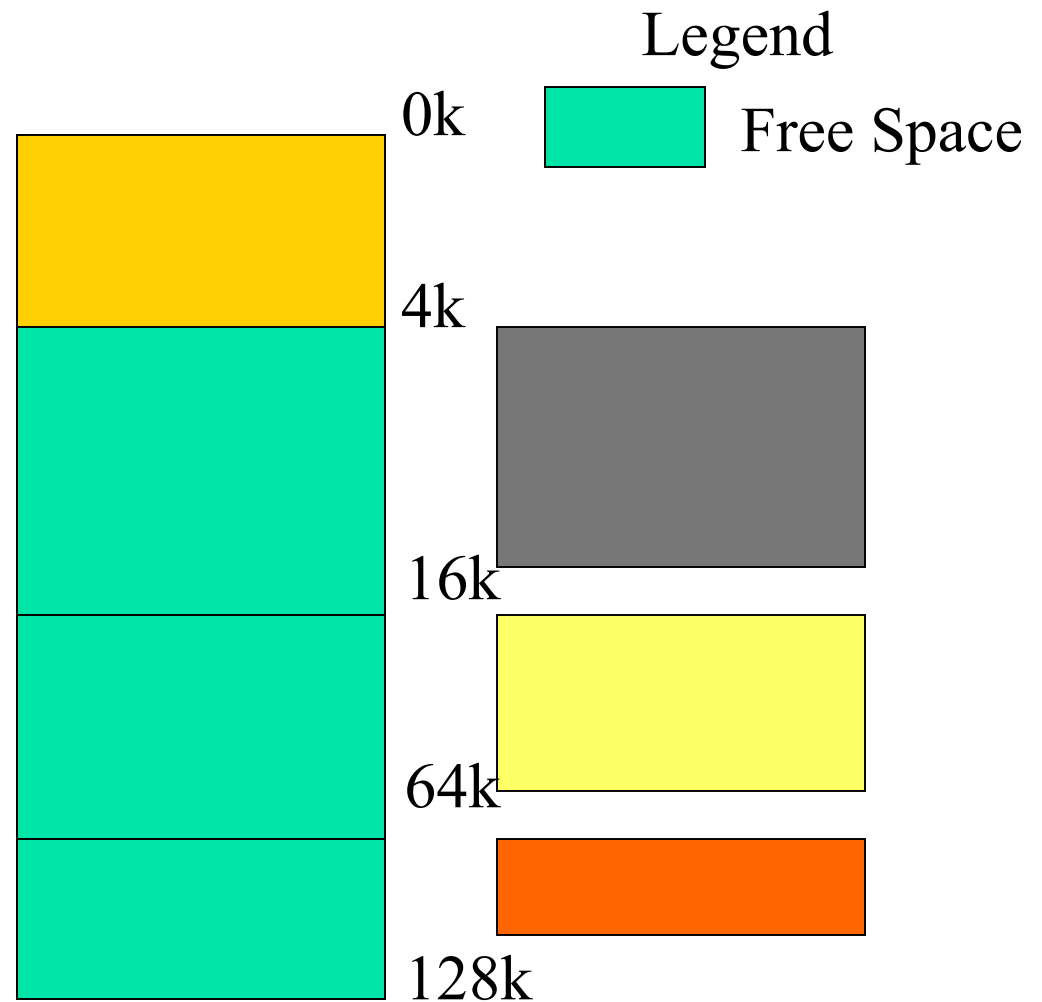
Fixed Partitions



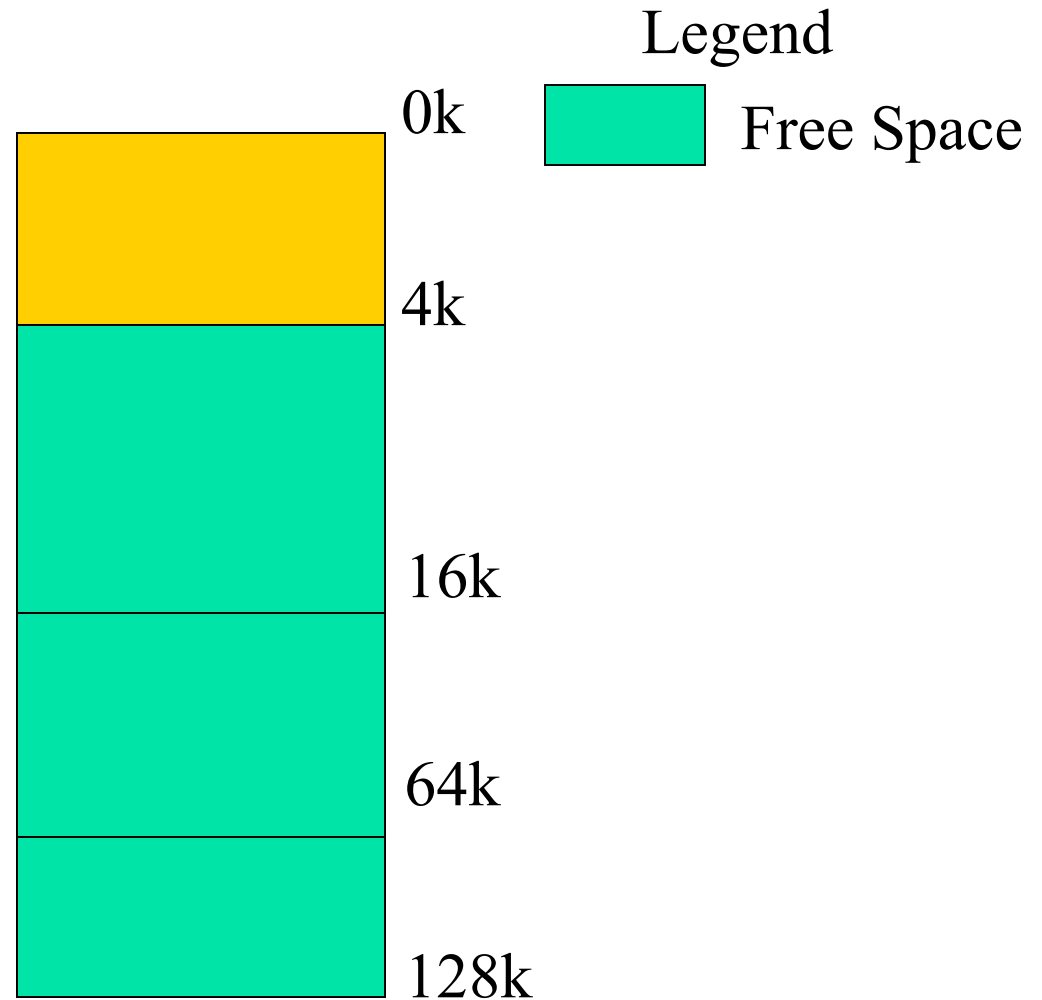
Fixed Partitions



Fixed Partitions



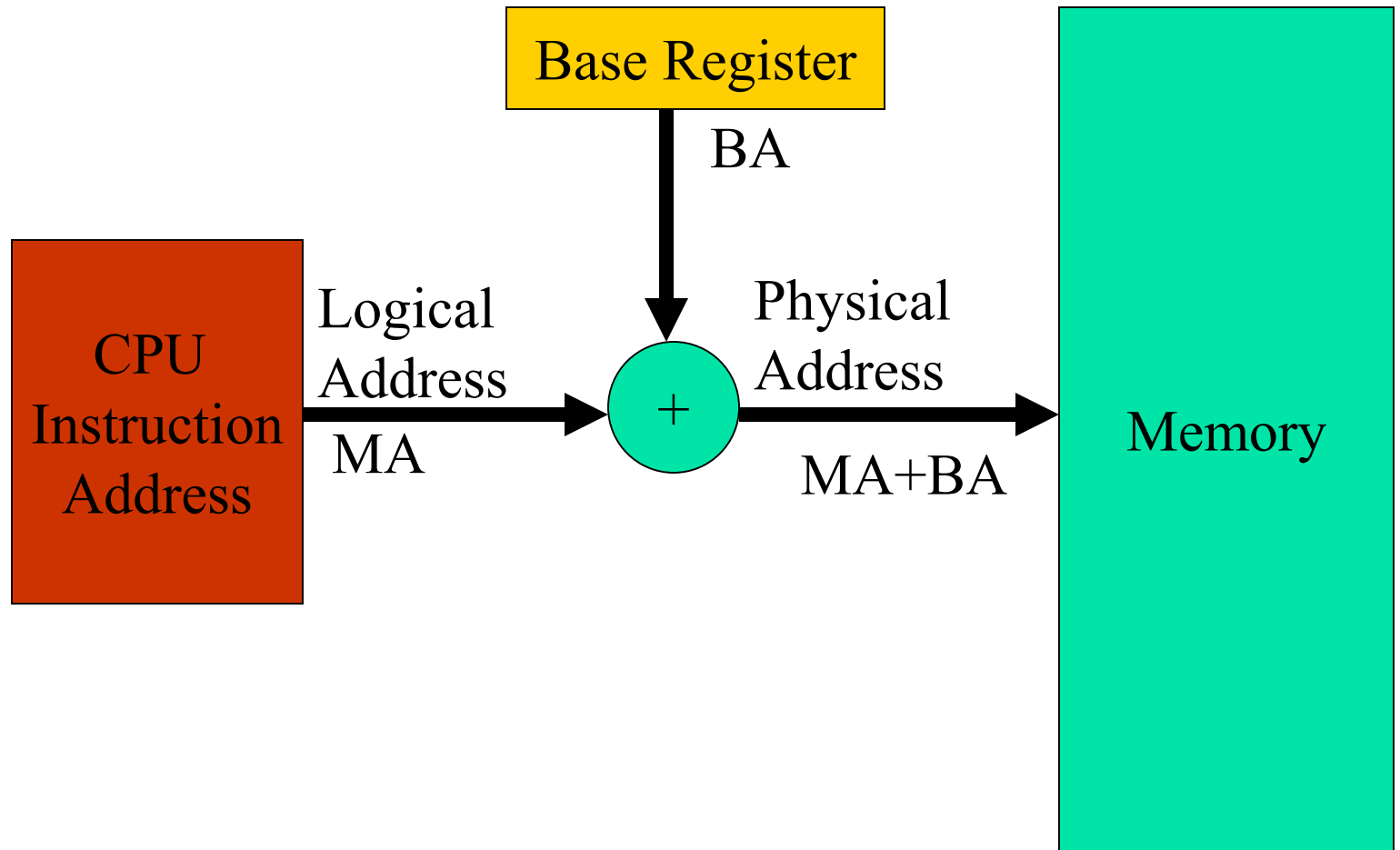
Fixed Partitions



Relocation of programs

- Program instructions refer to memory locations. For example:
 - `lw $10, 50($5)`
- The RTL is given by
 - `R10 \leftarrow Mem[R5+50]`
 - The address computed by R5+50: is it absolute? Virtual?
- If it is virtual, then it will have to change depending on where the program and its data is loaded in memory.
- Two ways to approach it:
 1. At load time, modify the memory addresses in each instruction in the program.
 2. Use a relocation (base) register and don't modify the instructions at all.

Relocation Register



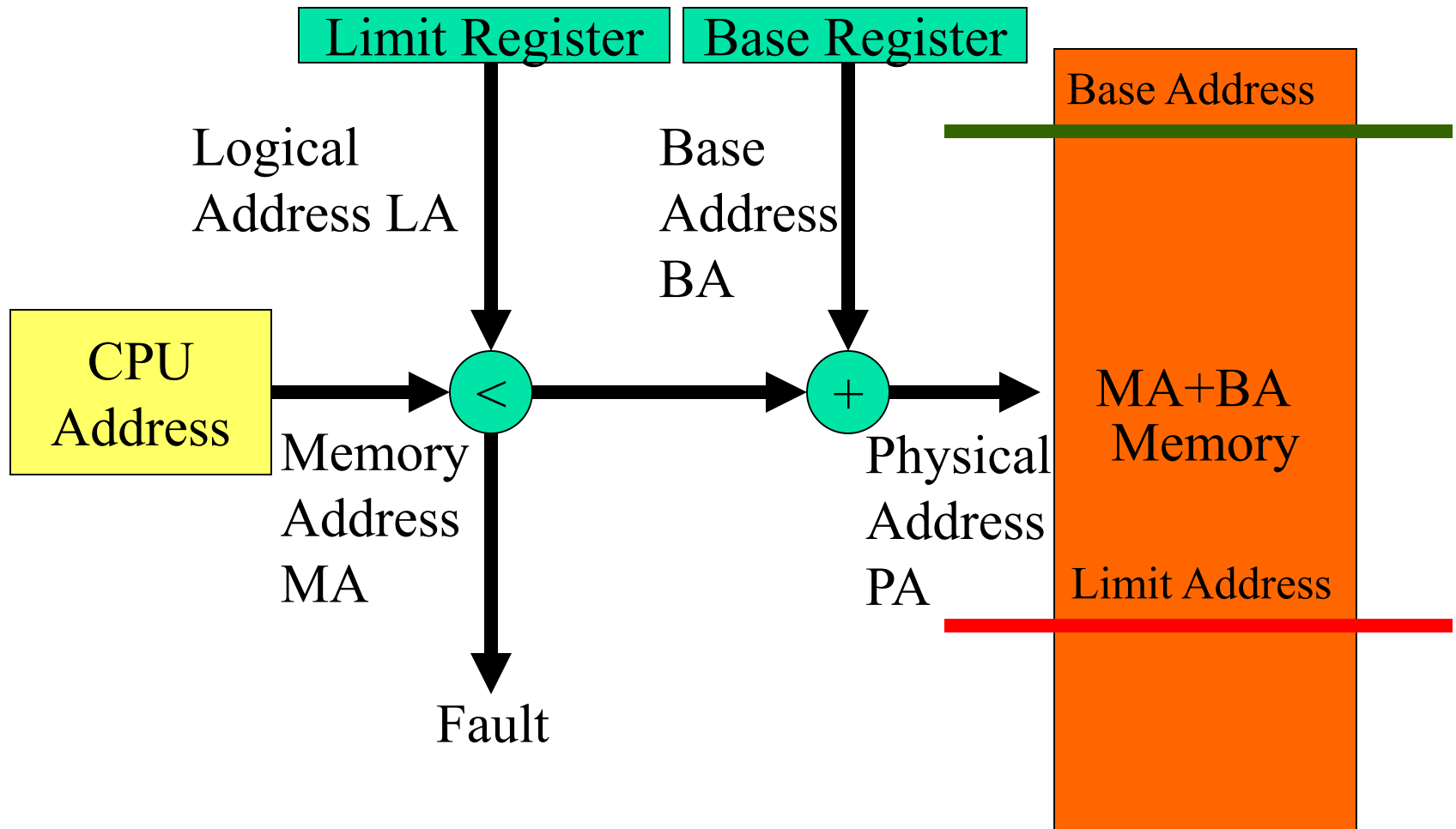
Relocation register

- Addresses in monitor < 0
 - Protection of user programs getting into monitor address space.
- User programs do not change
- Easier to debug
- All user program addresses start with 0.

Protection

- Memory protection can be done by adding a limit register.
 - Typically specifies the size of the valid memory address space.
- Use base and limit values
 - address locations added to base value to map to physical address
 - address locations larger than limit value is an error

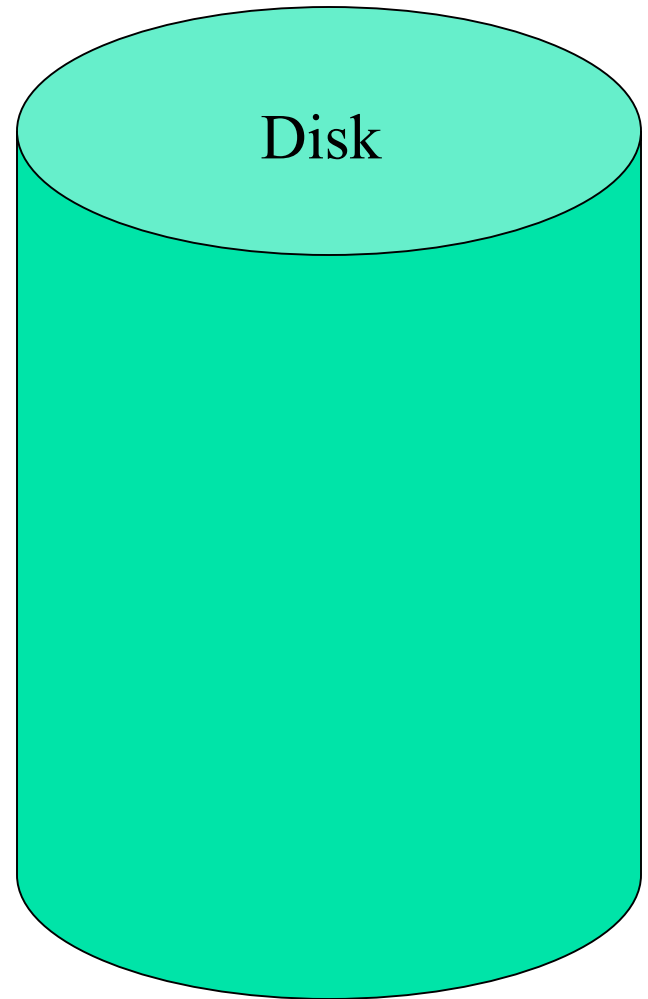
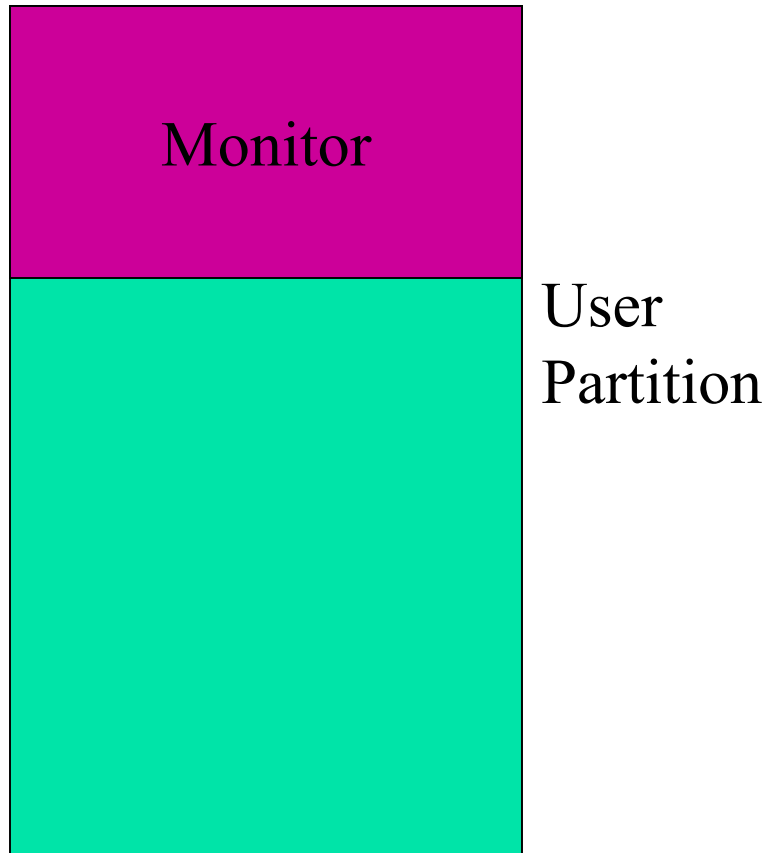
Base & Limit Registers



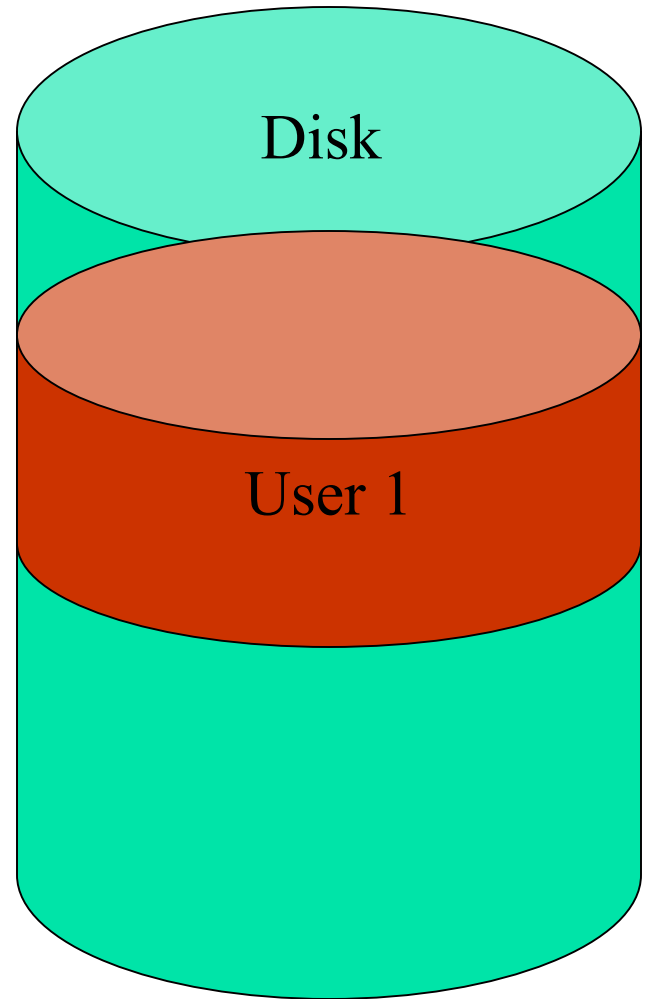
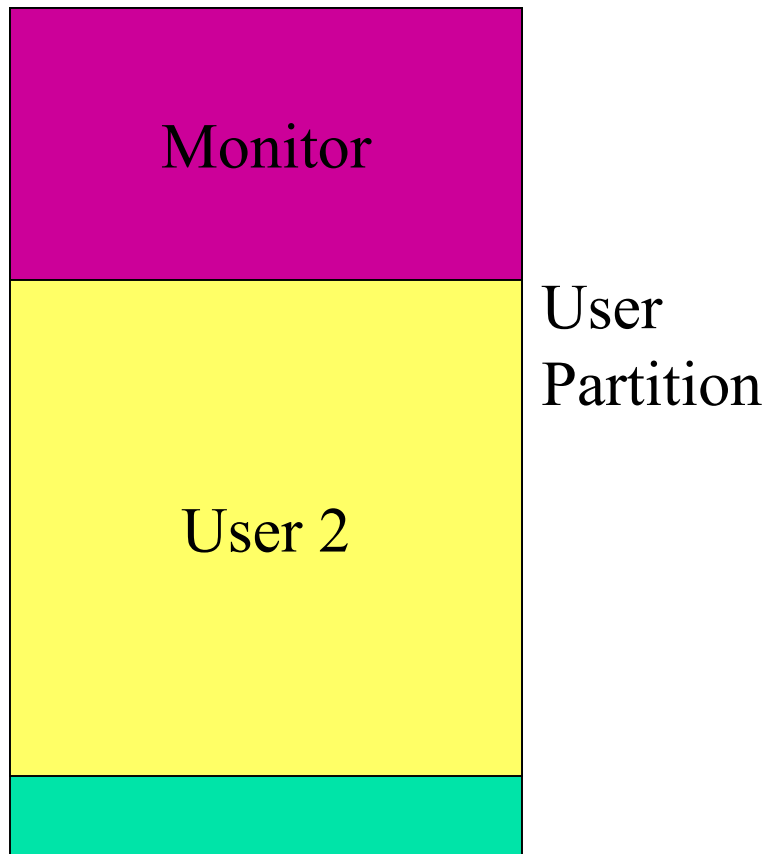
Swapping

- Fixed partitions work with batch systems.
- With interactive/timesharing systems, this is harder.
- Sometimes there is not enough memory to hold all the jobs. Two approaches:
 - Swapping (some jobs are swapped to disk)
 - Virtual memory (parts of programs may be loaded into memory).

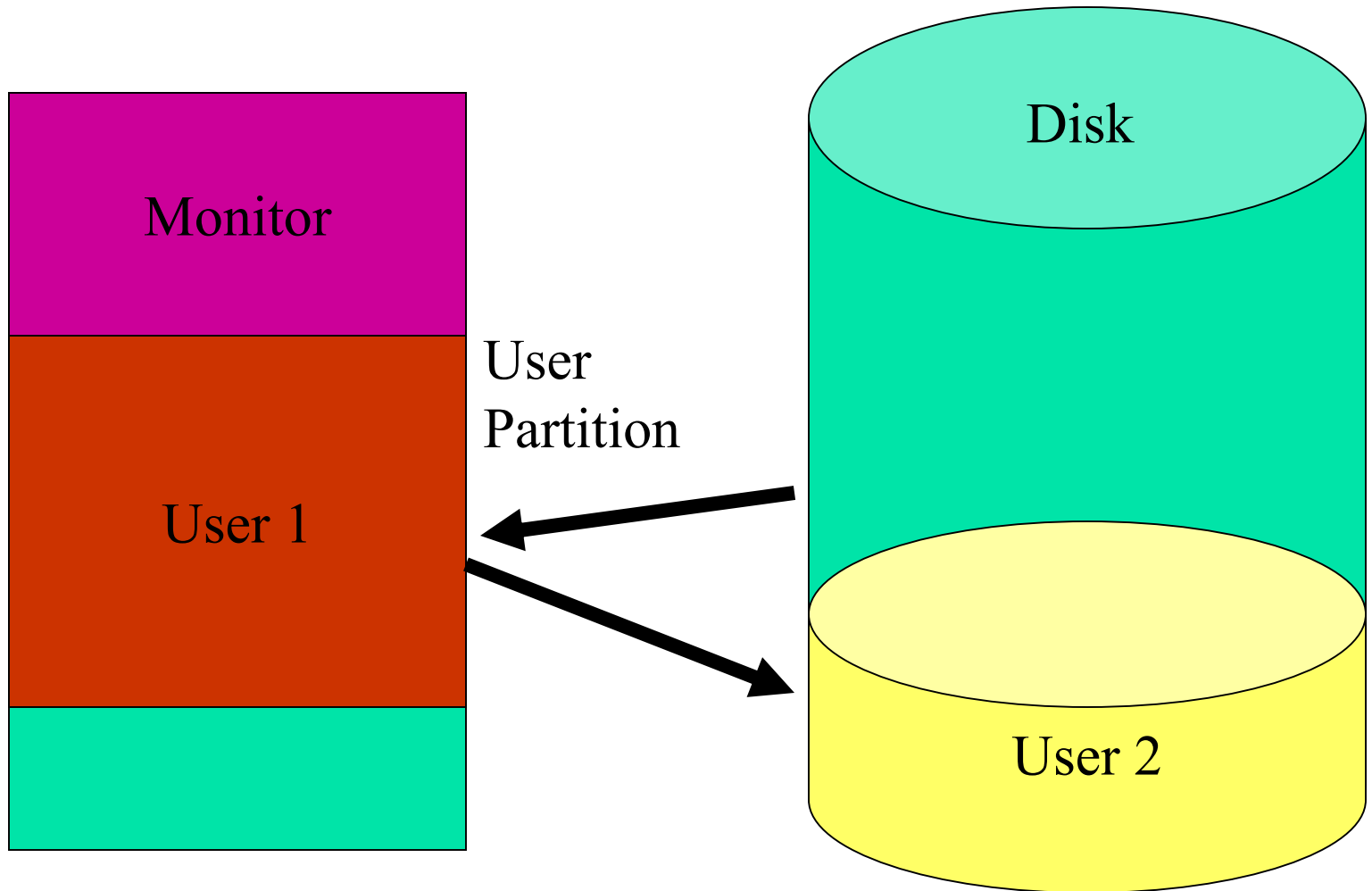
Swapping



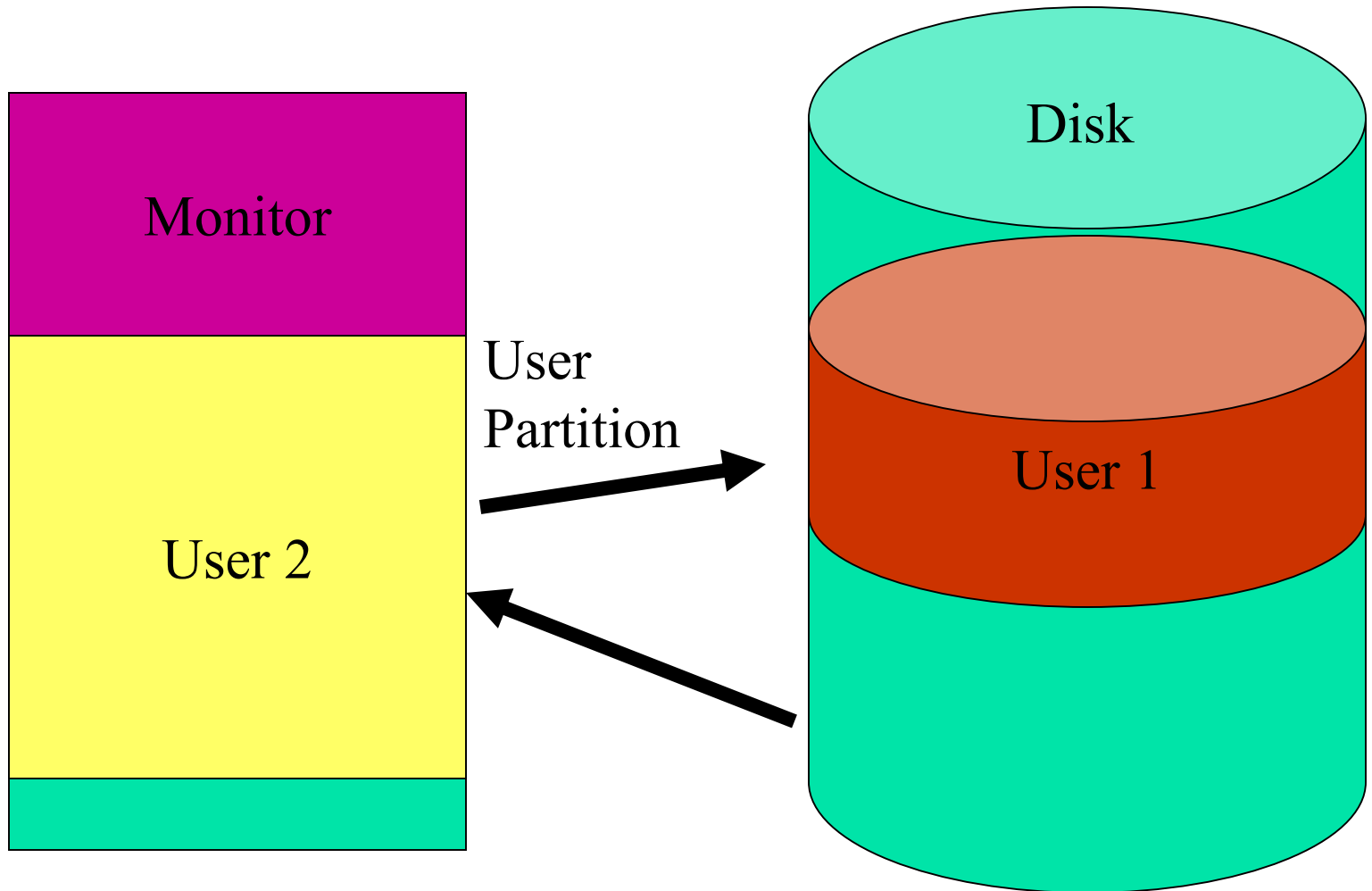
Swapping



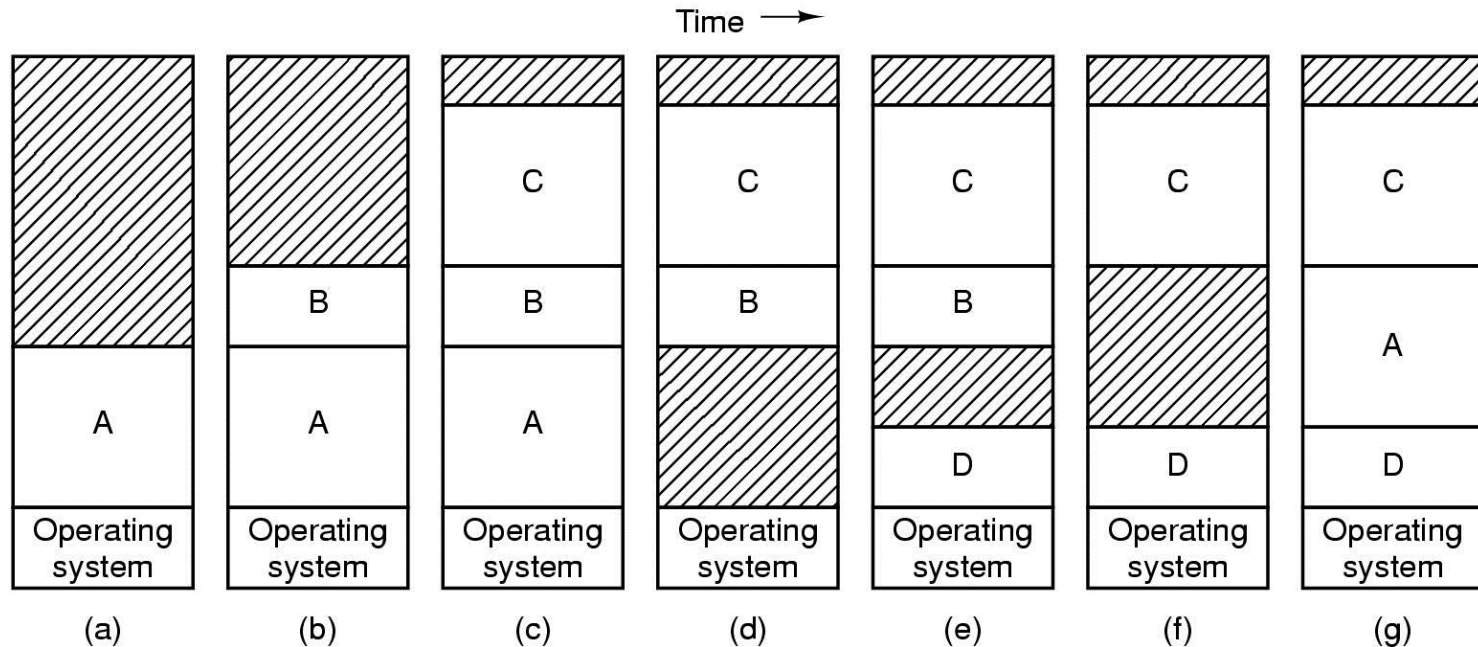
Swapping



Swapping



Swapping



- Memory allocation changes as
 - processes come into memory
 - leave memory
- Partitions are not fixed: they vary depending on job size.
- Shaded regions are unused memory

Fixed Partitions

- Partitions fixed, jobs variable sized
 - memory space fragmented: that is, parts of the fixed partitions that are not used, are wasted.
 - Called **Internal fragmentation**. The wasted space is internal to the fixed sized partition.

Variable Partitions and Fragmentation

- Fragmentation can also occur with variable partition memory management.
- Called **external fragmentation**.



Storage Placement Strategies

- **Best fit.** Use the hole whose size is equal to the need, or if none is equal, the whole that is larger but closest in size. Creates small holes that cant be used
- **First fit.** Use the first available hole whose size is sufficient to meet the need. Creates average size holes (what do we mean by first?)

Storage Placement Strategies

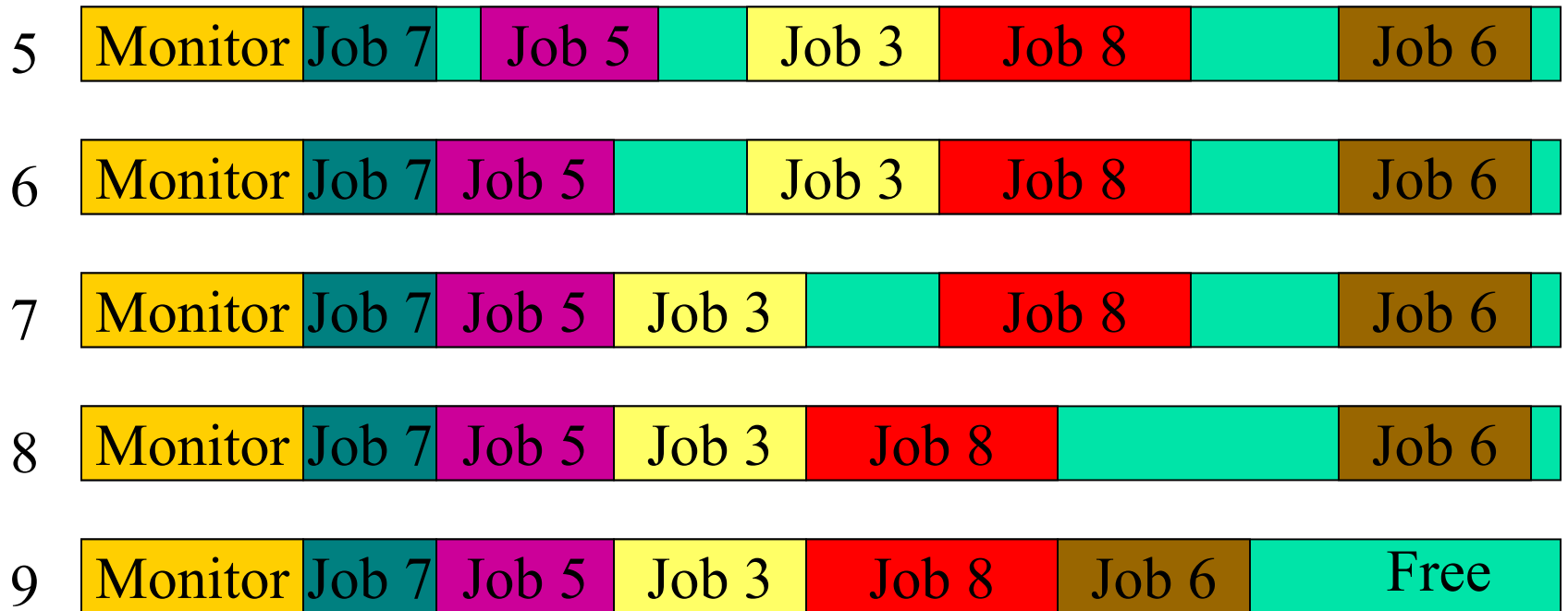
- **Worst fit.** Use the largest available hole. Gets rid of large holes making it difficult to run large programs
 - But since largest hole is used, the leftover may be more useful for putting other programs than small leftover holes.

Compaction

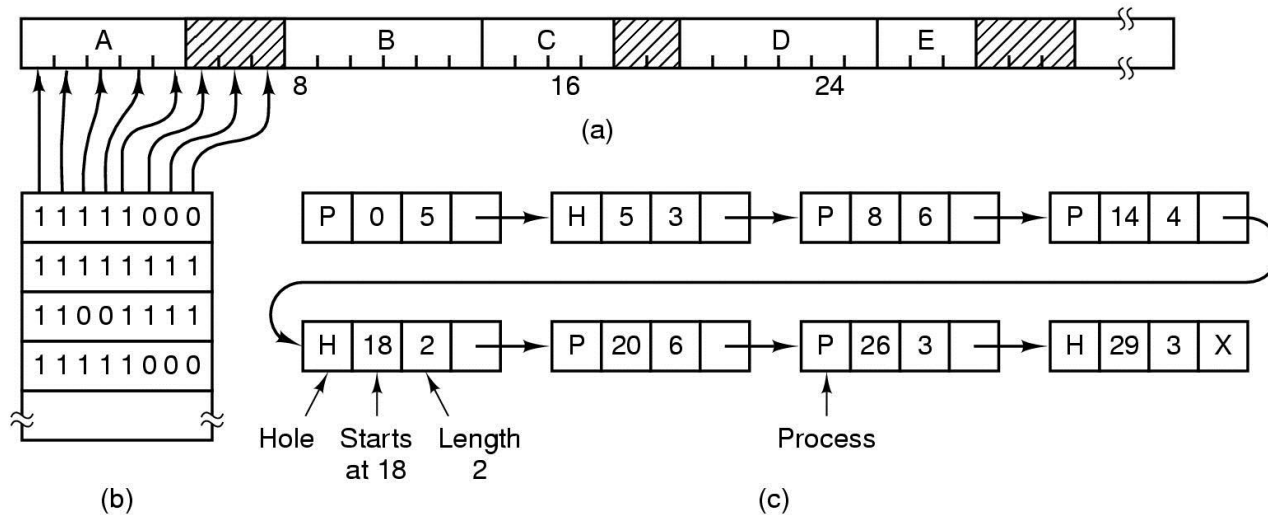
- Copy all memory space being shifted.
- Invalidates all cache info (bad)
- Very slow (bad)

Compaction

- Moving processes in memory to combine fragments of holes.

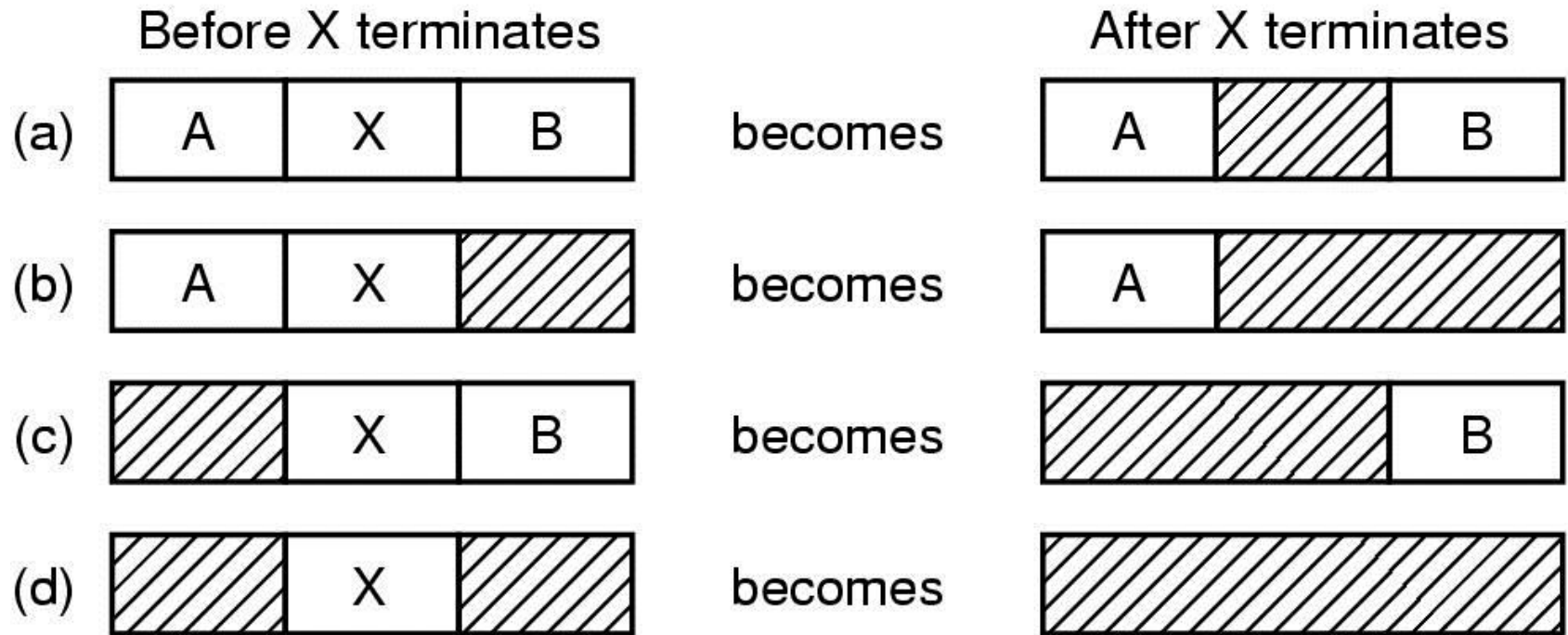


Memory Management with Bit Maps



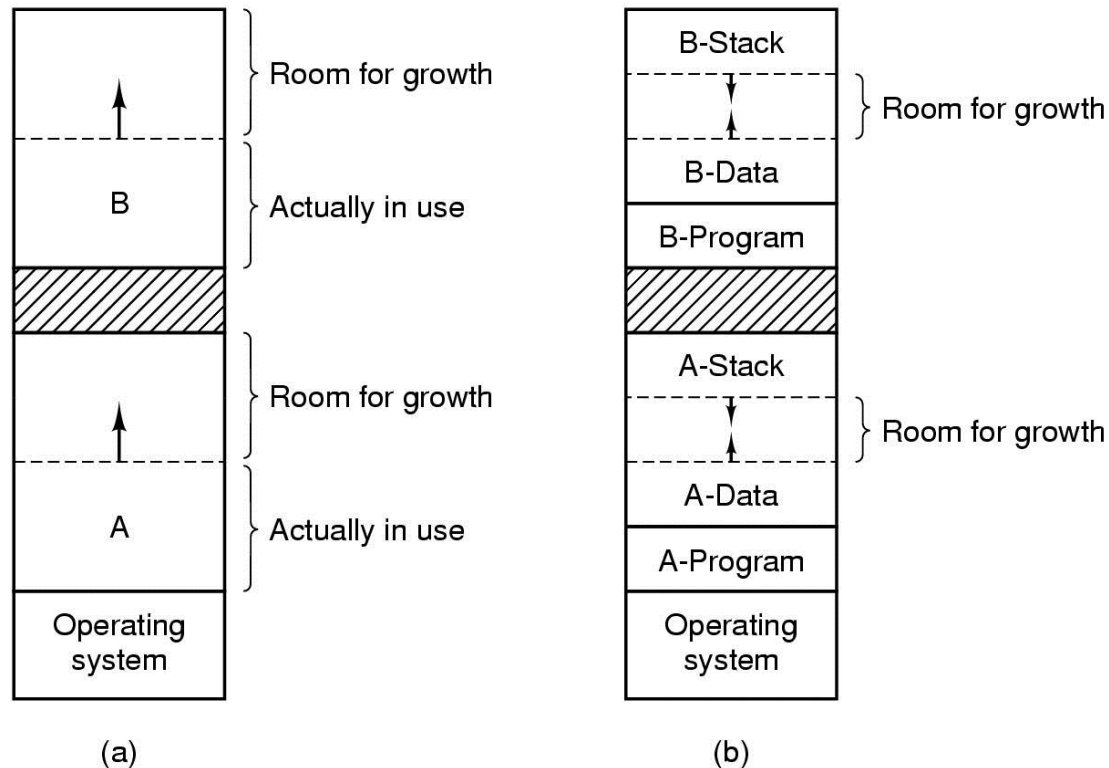
- Part of memory with 5 processes, 3 holes
 - tick marks show allocation units
 - shaded regions are free
- Corresponding bit map
- Same information as a list

Memory Management with Linked Lists



- Four possible neighbor combinations for the terminating process X
- Shaded regions are free memory.

Memory allocation for processes



- Allocating space for growing data segment
- Allocating space for growing stack & data segment

Other memory management approaches

- Could break programs into smaller units because they will fit better
 - Use multiple base registers, one for each unit
 - Examples
 - Code/Data
 - Constants/variables
- Base&bounds for stack
Base&bounds for heap
.....
- This is becoming very similar to segmented memory (which we will study later).

Storage Management Problems

- Fixed partitions suffer from internal fragmentation
 - **Internal fragmentation**: wasted space inside fixed partitions.
- Variable partitions suffer from external fragmentation
 - **External fragmentation**: wasted space outside job regions.
- Compaction suffers from overhead
- Partitions must be less in size than real memory
- Overlays are painful to program efficiently
- Swapping requires writing to disk sectors

How bad is fragmentation?

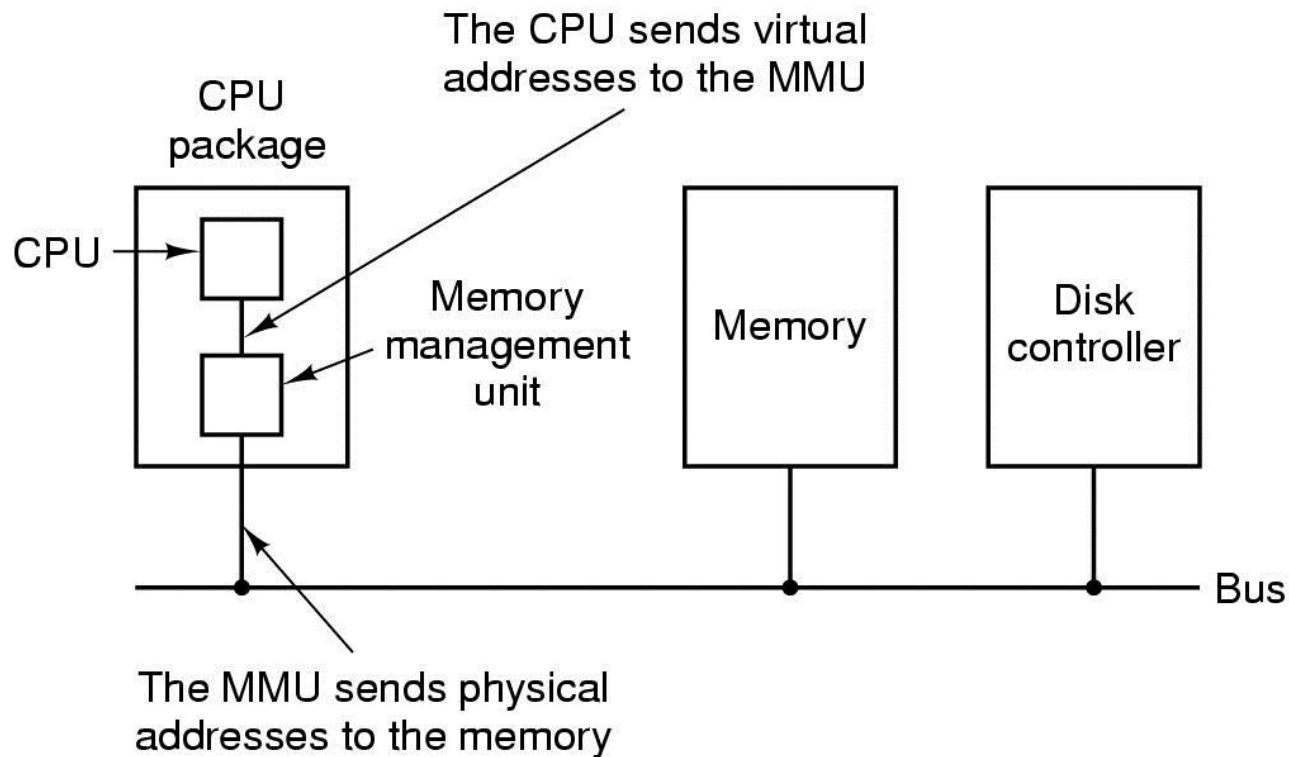
- Statistical arguments - Random sizes
- First-fit strategy
- Given N allocated blocks
- On average an additional $N/2$ blocks will be lost because of fragmentation (in addition to N)
- **Known as 50% RULE**
- i.e. $(N/2)/(N+N/2) = 33\%$ of memory may be unusable!!!

Discussion

- What schemes could be used to overcome fragmentation?
 - Make programs all the same size.
 - Divide program to fixed size blocks.
 - Why not word size block? → **efficiency. Block sizes are tuned to disk transfer rates.**
- What does the use of secondary storage for swap space imply about memory organization?
 - Goal: exploit memory hierarchy
 - Programs with physical memory size were not sufficient for doing reasonable work.

Virtual Memory & Paging

- The position and function of the MMU



Paging

- Paging is like cache
- Provide user with virtual memory that is as big as user needs
- Store virtual memory on disk
- Cache parts of virtual memory being used in real memory
- Load and store cached virtual memory without user program intervention
 - Automatic swapping: check to see if virtual address has a mapping into the cache

Paging

- Principle of locality applies.
- Reduce physical memory fragmentation by making cacheable units all the same size
 - Advantages: no compaction and swapping is made easy.
- Side effect: remove real memory deadlock possibilities by preempting real memory (i.e., deny no-preemption condition)
 - If we can swap a page out, no deadlocks can occur on a chunk of physical memory.

Paging

- What should page sizes be?
 - Large blocks → more wasted space due to internal fragmentation
 - Small blocks → more swaps
 - Somewhere in between extremes:
 - usually matches disk block sizes.

Paging

- Translate VA to PA.
 - Interpret VM addresses as they are executed in HW.
- What's the limit on process address space?
 - # of address bits. Disk space is not the limiting factor: one can always buy more disks.

How does address translation occur?

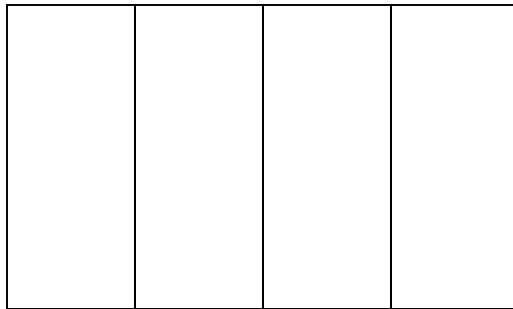
- Answer: through table lookup called a **page table**.

How does address translation occur?

- First, let's see some examples, then we'll talk more about the page tables and mapping.

Paging example set up

Cache



0 1 2 3

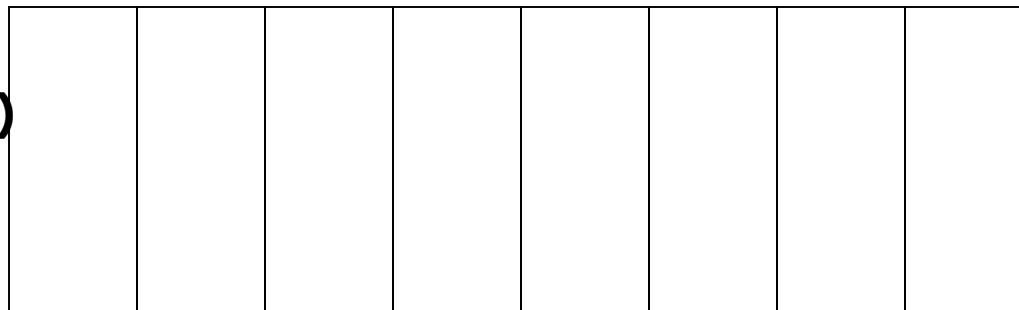
**Address
Translation/
mapping**

Page Table

VP Frame

	0
	1
	2
	3

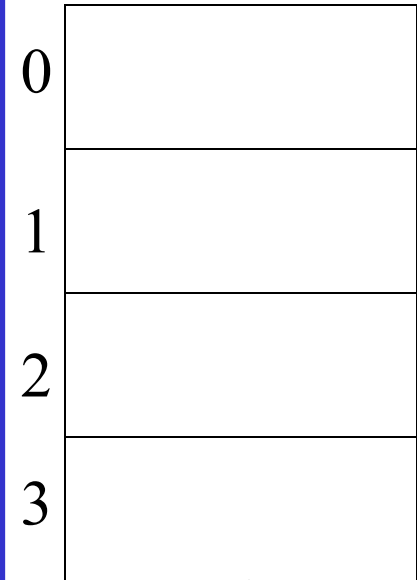
Virtual Memory Stored on Disk



0 1 2 3 4 5 6 7

**User
(process)
Address
space**

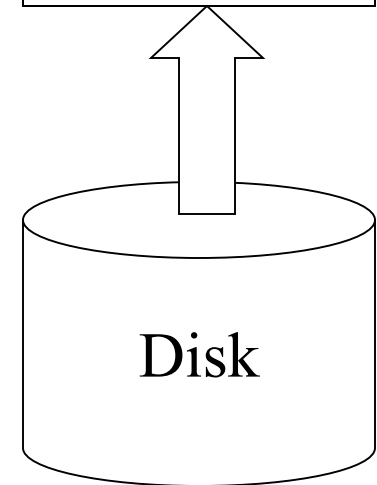
Real Memory



1

2

3



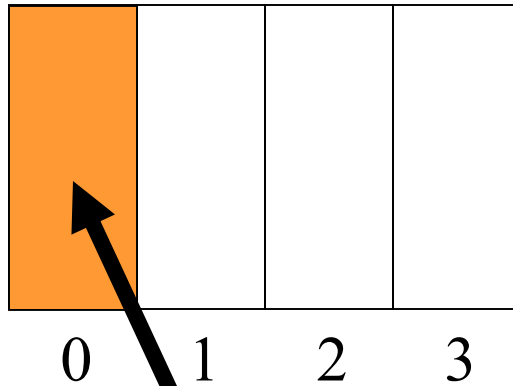
Disk

implementation

Paging

Request
Address
within
Virtual
Memory
Page 2

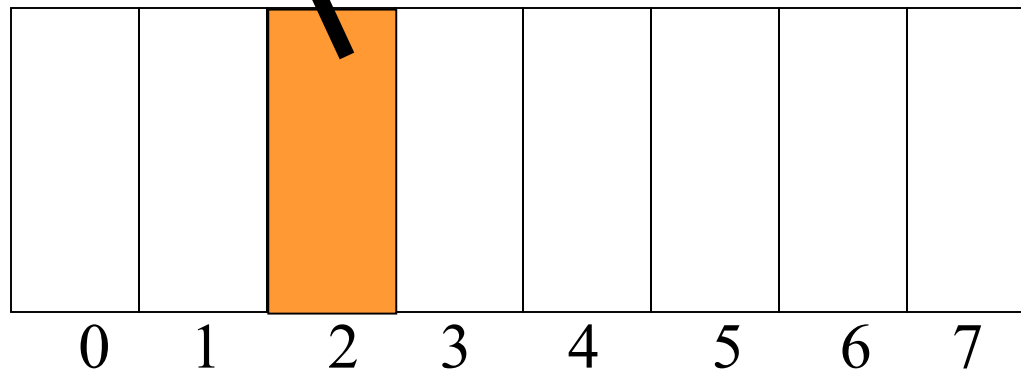
Cache



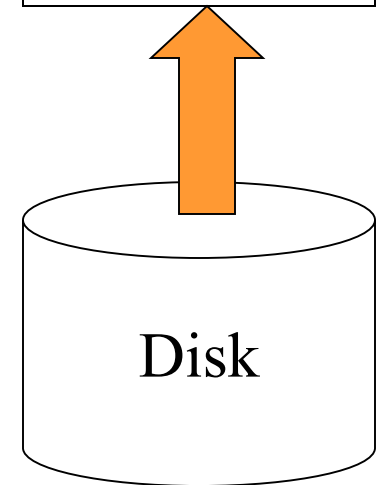
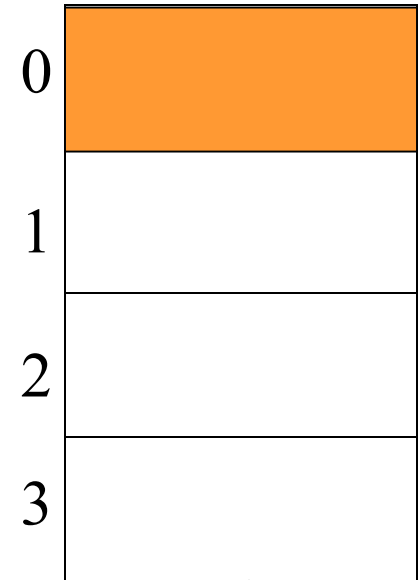
Page Table
VP Frame

VP	Frame
2	0
	1
	2
	3

Virtual Memory Stored on Disk



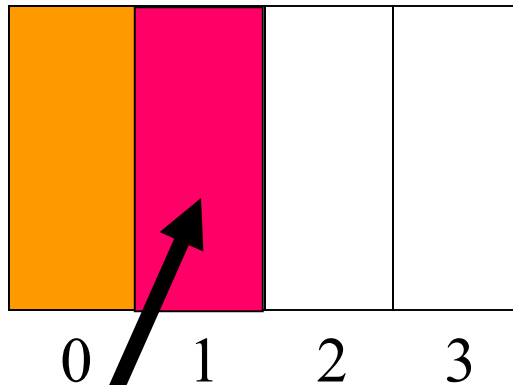
Real Memory



Paging

Request
Address
within
Virtual
Memory
Page 0

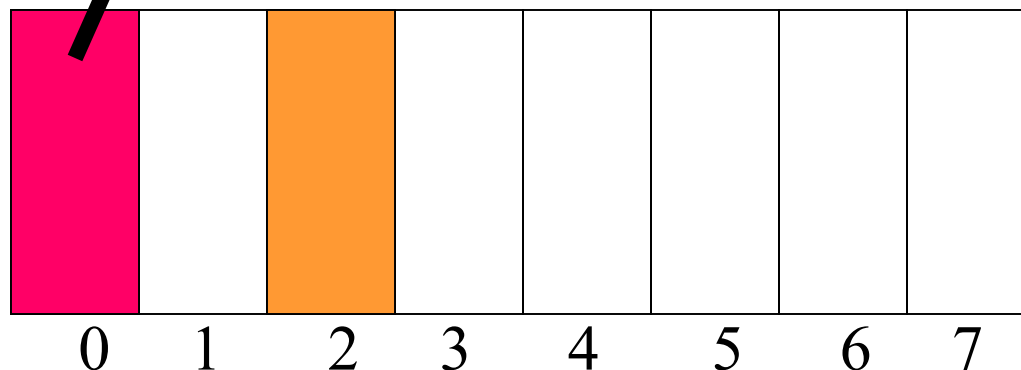
Cache



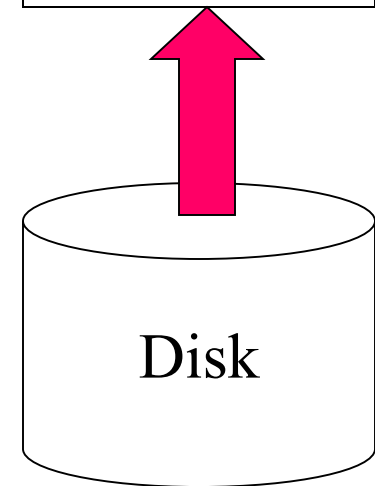
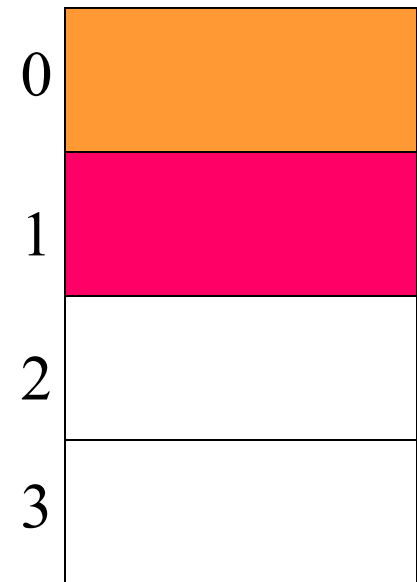
Page Table
VP Frame

2	0
0	1
	2
	3

Virtual Memory Stored on Disk



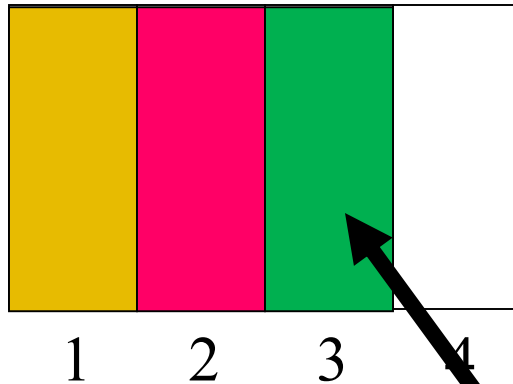
Real Memory



Paging

Request
Address
within
Virtual
Memory
Page 5

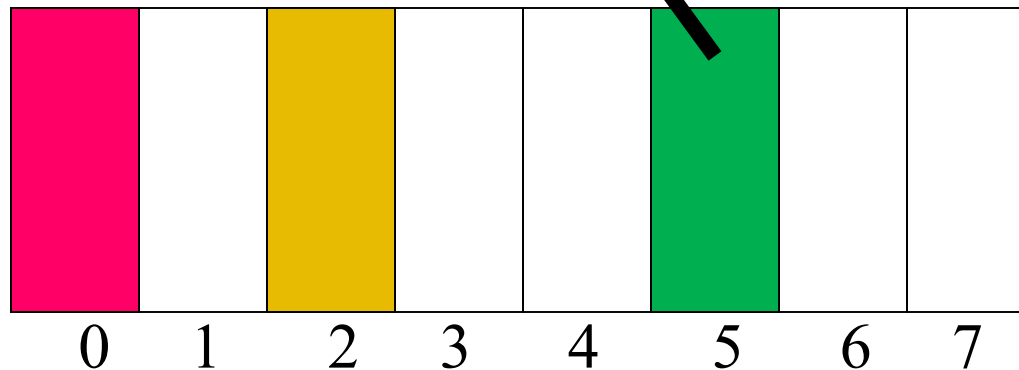
Cache



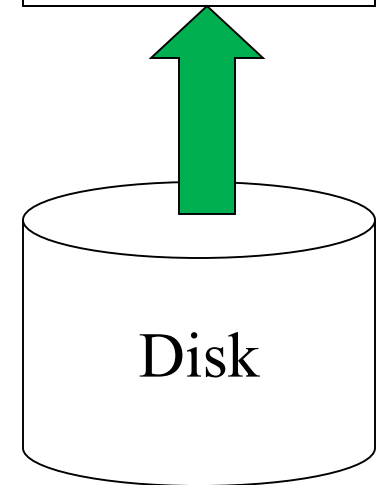
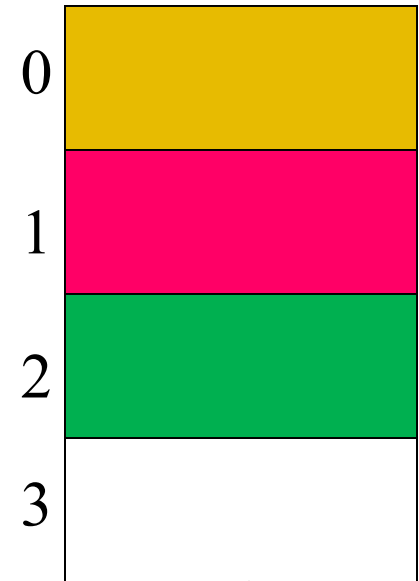
Page Table
VP Frame

2	0
0	1
5	2
	3

Virtual Memory Stored on Disk



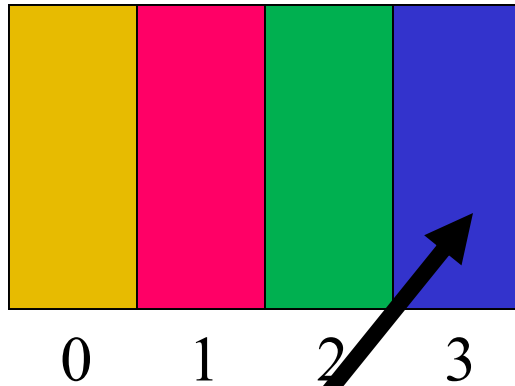
Real Memory



Paging

Request
Address
within
Virtual
Memory
Page 1

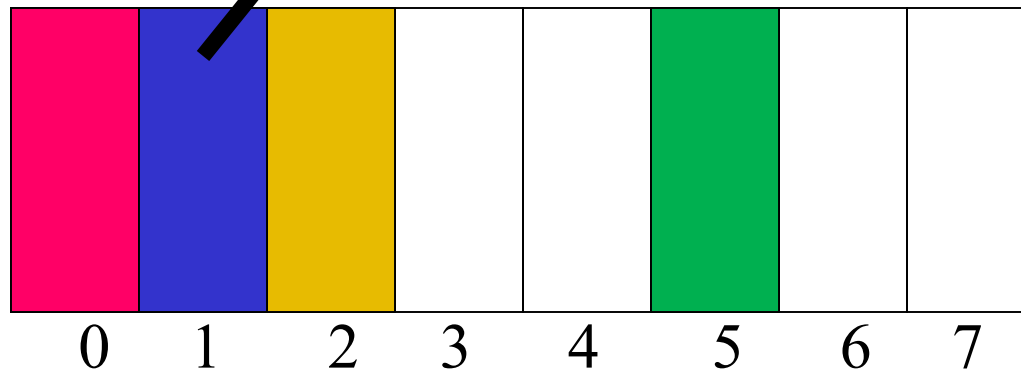
Cache



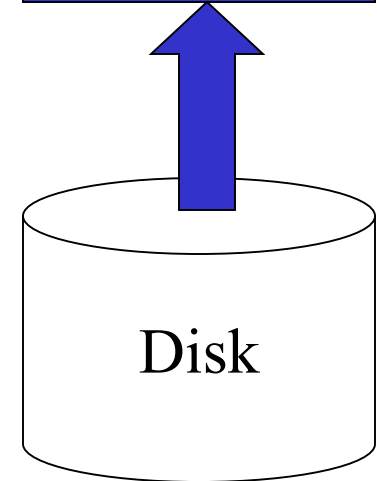
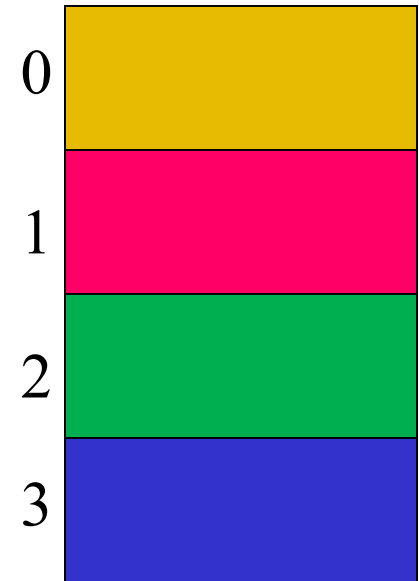
Page Table
VP Frame

2	0
0	1
5	2
1	3

Virtual Memory Stored on Disk



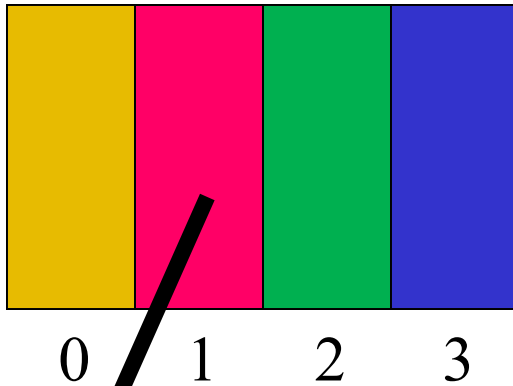
Real Memory



Paging

Request
Withdraws
Memory
Physical
Memory
7

Cache

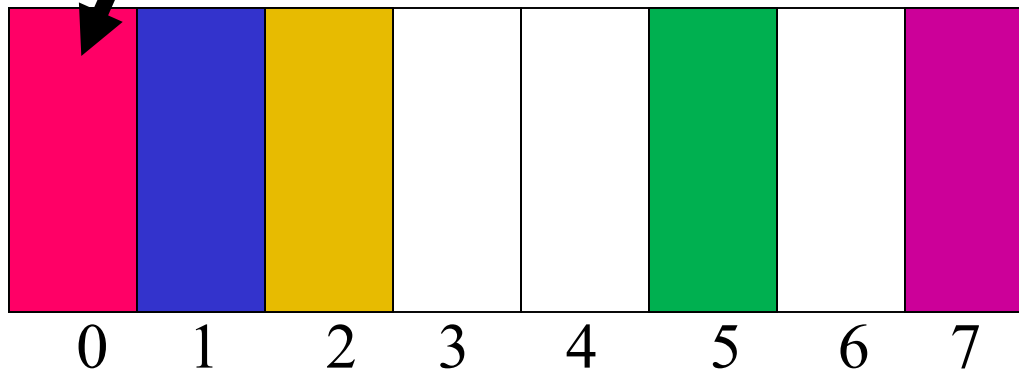


Page Table

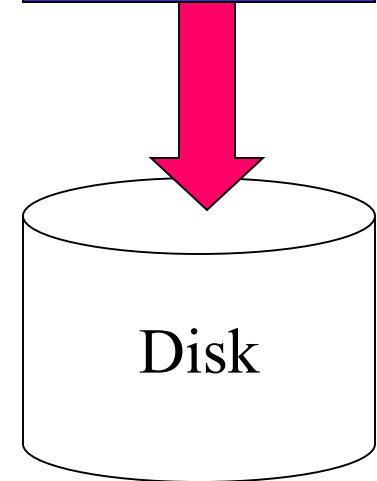
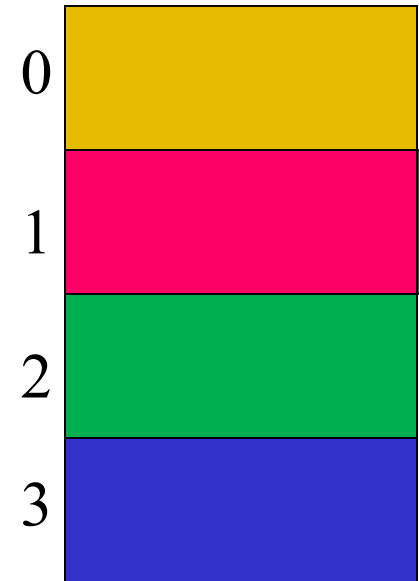
VP Frame

2	0
0	1
5	2
1	3

Virtual Memory Stored on Disk



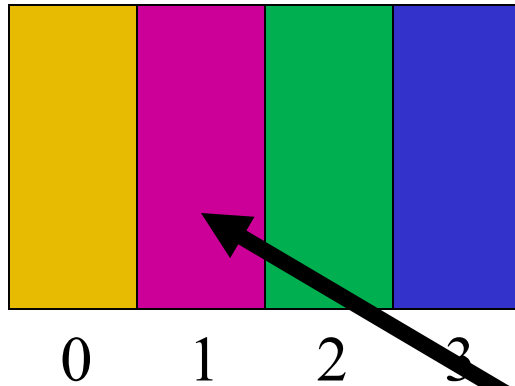
Real Memory



Paging

Load
Virtual
Memory
Page 7 to
cache

Cache

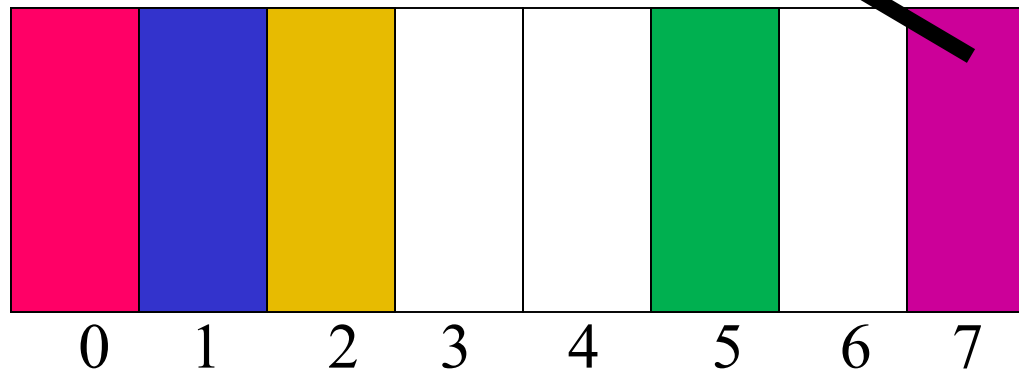


Page Table

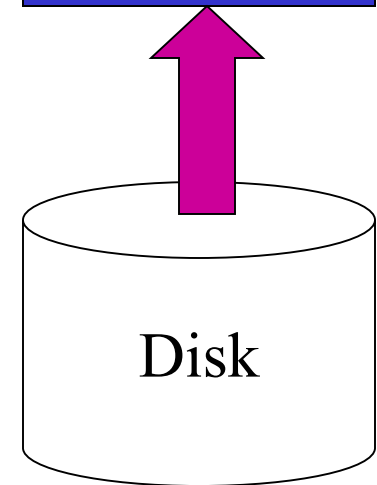
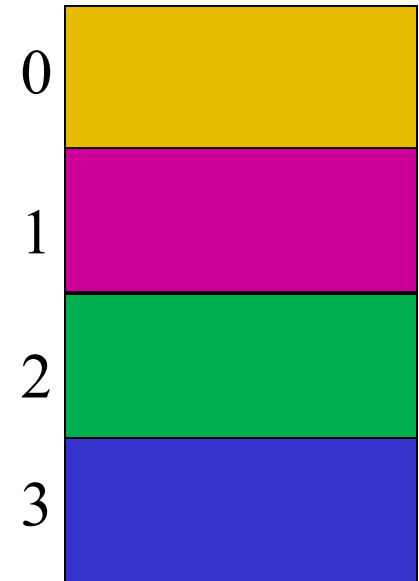
VP Frame

2	0
7	1
5	2
1	3

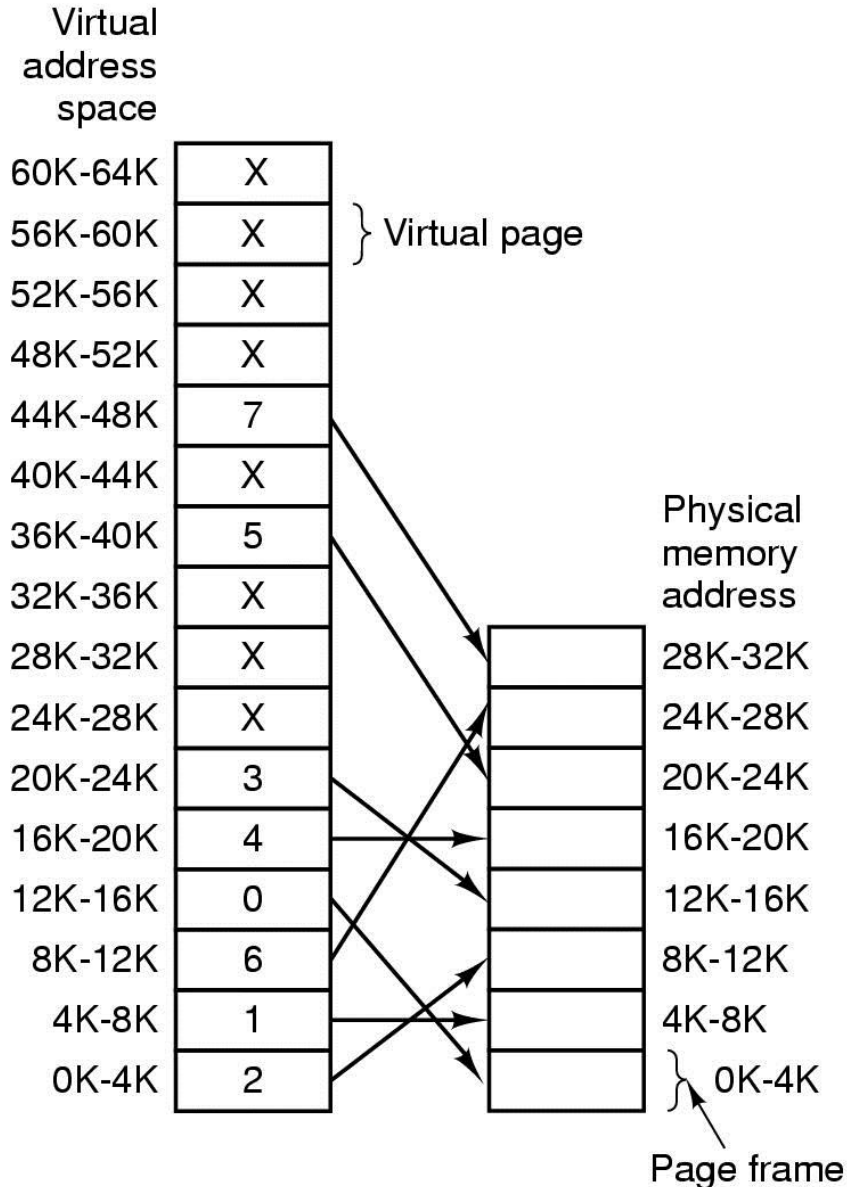
Virtual Memory Stored on Disk



Real Memory

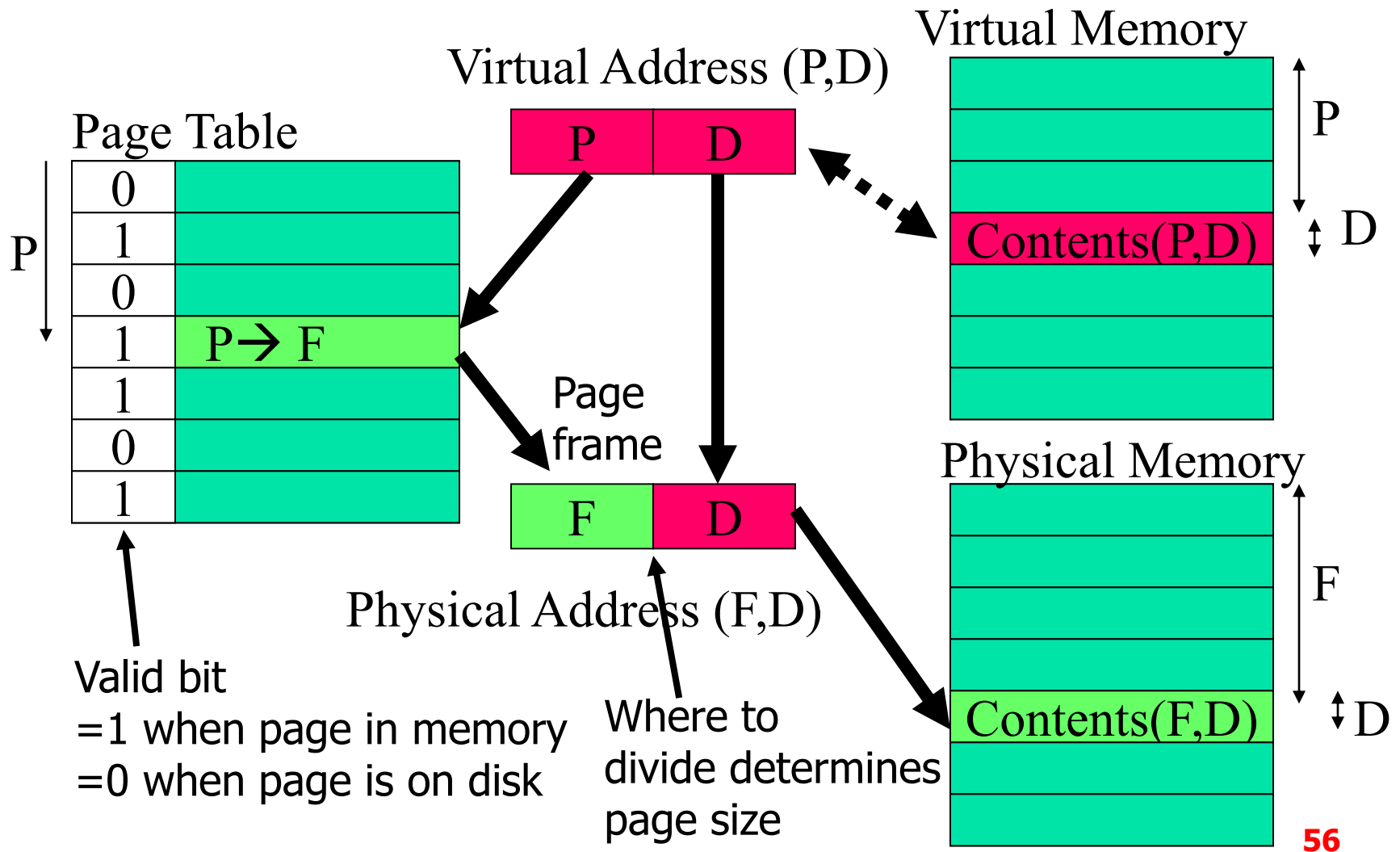


Address mapping and page tables



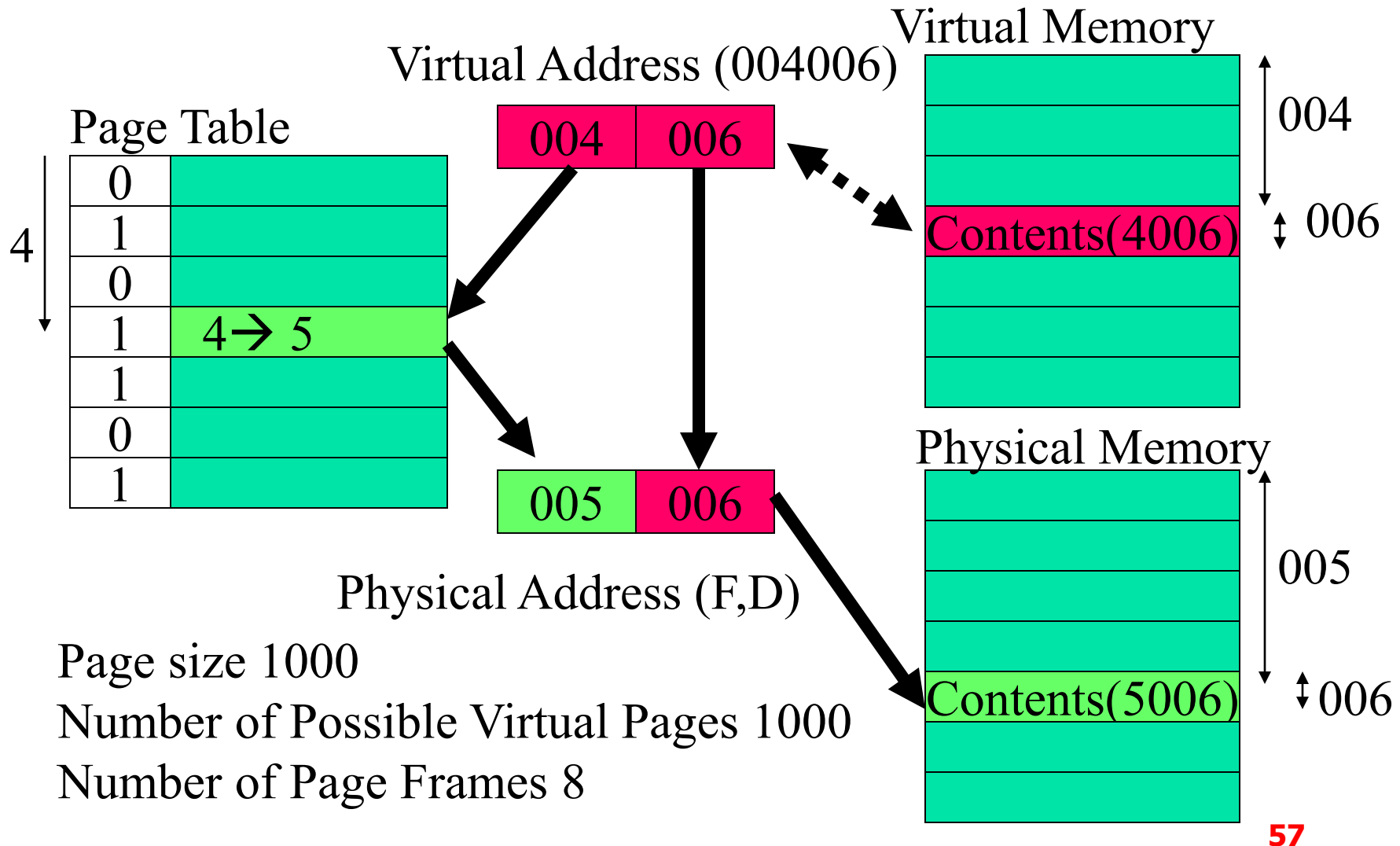
The relation between virtual addresses and physical memory addresses given by page table

Page Mapping Hardware

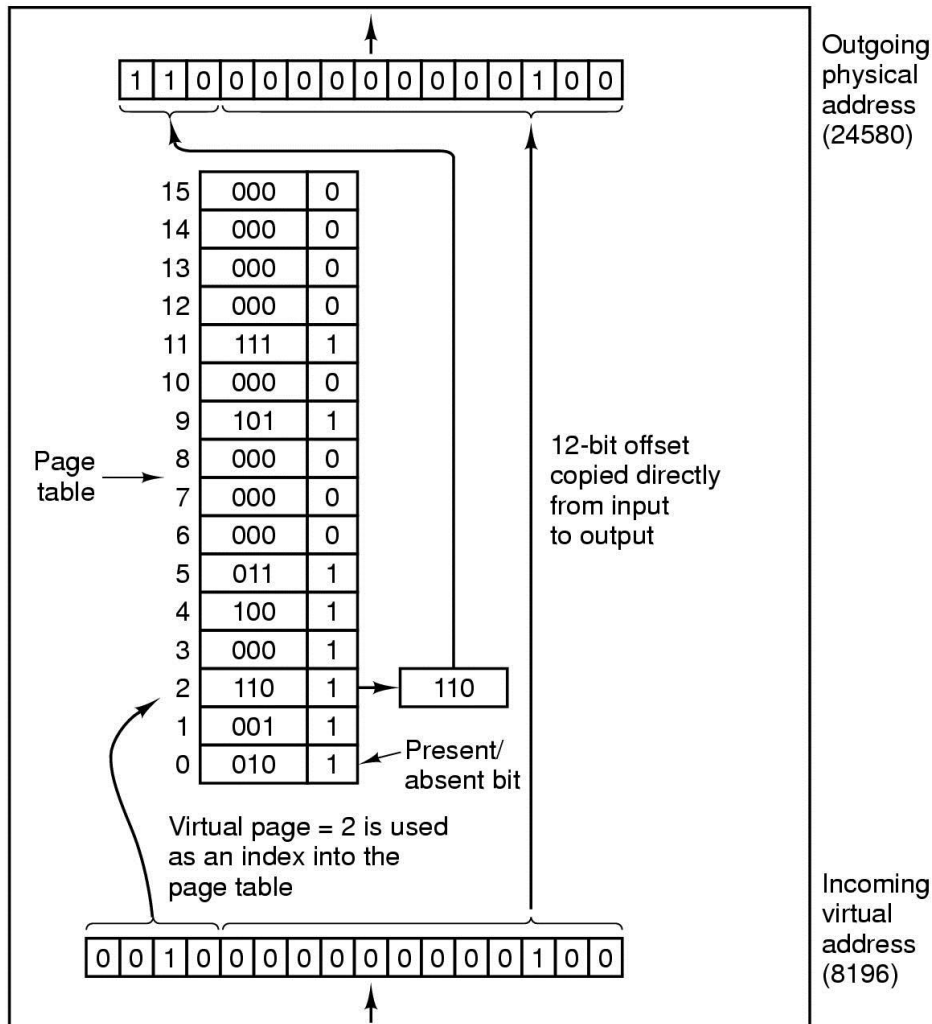


Addresses in decimal for purposes of this example.

Page Mapping Hardware




Page Tables/binary address



- Internal operation of MMU with 16 4 KB pages

Paging Issues – page sizes

- Size of page is 2^n , usually 512, 1k, 2k, 4k, or 8k

Tends to correspond to block size on disk
- VM address may have 2^{20} (1 meg) pages with 4k (2^{12}) bytes per page
 - 2^{32} bit address = 4Gigs
- 2^{20} (1 meg) 32 bit page entries take 2^{22} bytes (4 meg)
 - $2^{20} * 2^2 = 2^{22}$ bytes (used for page tables)
- Page frames must map into real memory
- Some machines go to 64 bit addresses. This results in too large page tables.
 - Solution? We'll see later.

Paging Issues – page sizes

- 32 meg max real memory (2^{25}) has 2^{13} page frames of 4k pages
- Choose page frame as 2^k disk sectors
 - Avoids external fragmentation
- NO external fragmentation
- Internal fragmentation on last page ONLY

Paging issues – context switch

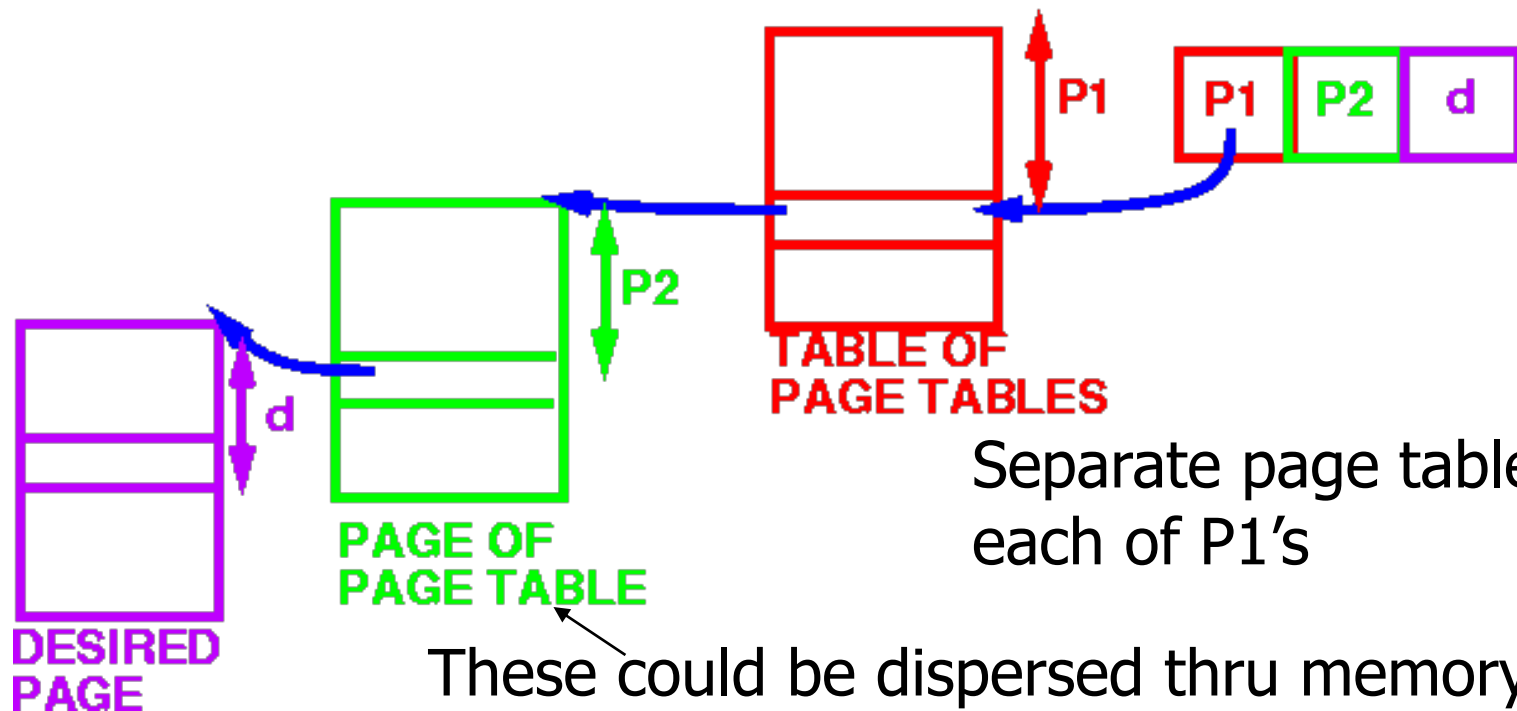
- Page tables belong to processes
 - This is how each process has its own private virtual address space
- What happens during context-switch?
 - Must swap page table information along with the process
 - Too expensive to actually replace page tables
 - Instead, keep page tables in memory and point to where each page table resides via a “page table base register”
 - Change the contents of page table base register during context switch – only one register contents is switched; fast.

Paging issues – access time

- Each address translation involves two memory accesses:
 - One to look up in the page table
 - One to access the actual data in memory
- How can paging be made faster?
 - More than one solution
 - multi-level page tables
 - inverted page tables
 - Translation lookaside buffers (TLB's)

Multilevel Paging

LOGICAL ADDRESS



These could be dispersed thru memory
Page tables can be sparse: the entire page table doesn't have to be allocated if it is not used.

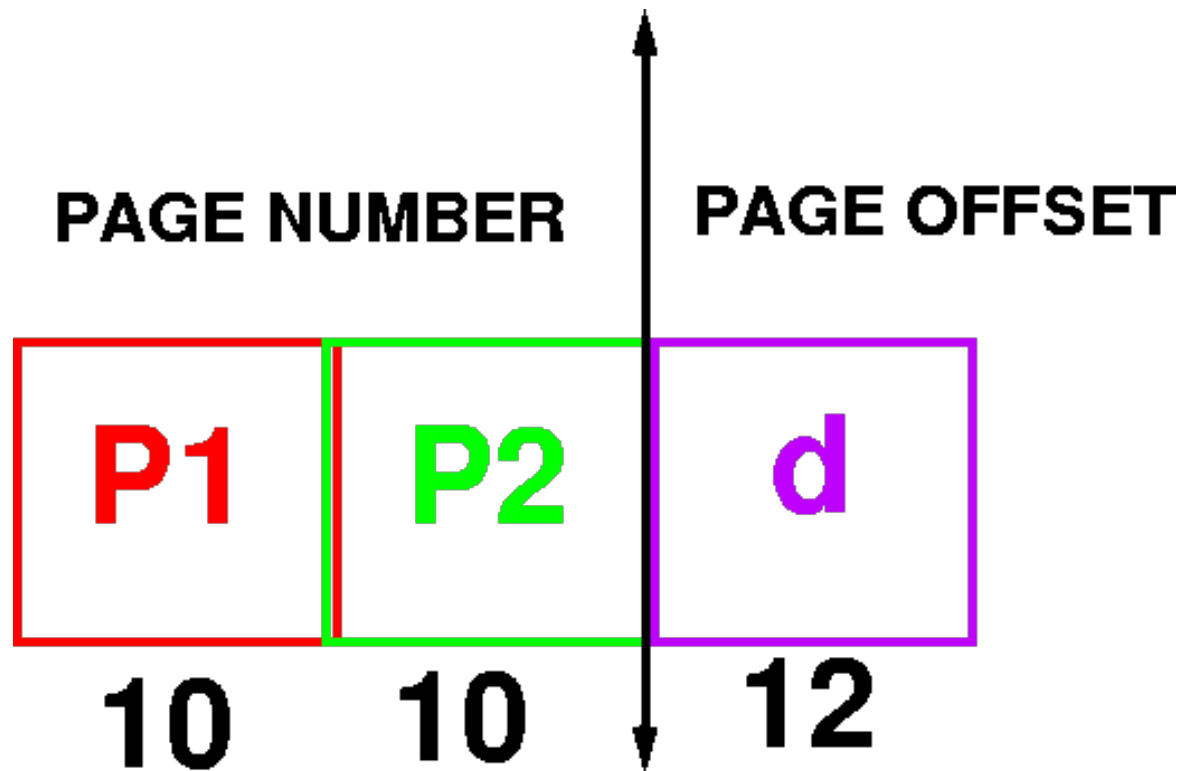
Multi-level paging

- Useful if we split the program into separate regions for stack, heap, text, etc.
- P1 could point to page tables for stack or text or heap
- **Cost**: potentially 2 page misses.
- **Advantage**: don't have to have the page table for pages that are not used.
 - When P2 table is allocated, the size could be made less than a full size of a page table. [e.g., use base and bounds for P1]

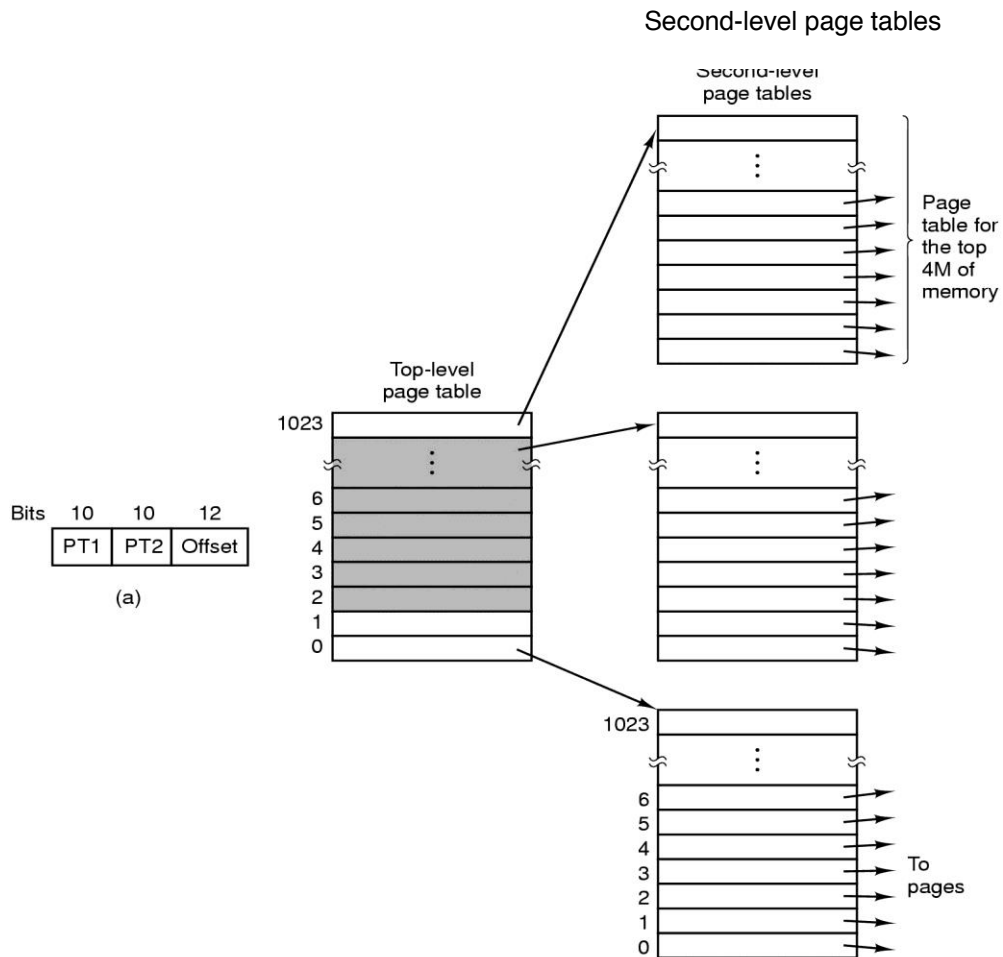
Addressing on Multilevel Page Table System

- A logical address (on 32 bit machine with 4k page size) is divided into
 - A page number consisting of 20 bits
 - A page offset consisting of 12 bits
- Since the page table is paged, the page number is further divided into:
 - A 10-bit page number
 - A 10-bit page offset

Addressing on Multilevel Page Table



Multi-level Page Tables



- 32 bit address with 2 page table fields
- Two-level page tables

AMD 64 Processor VM page table structure

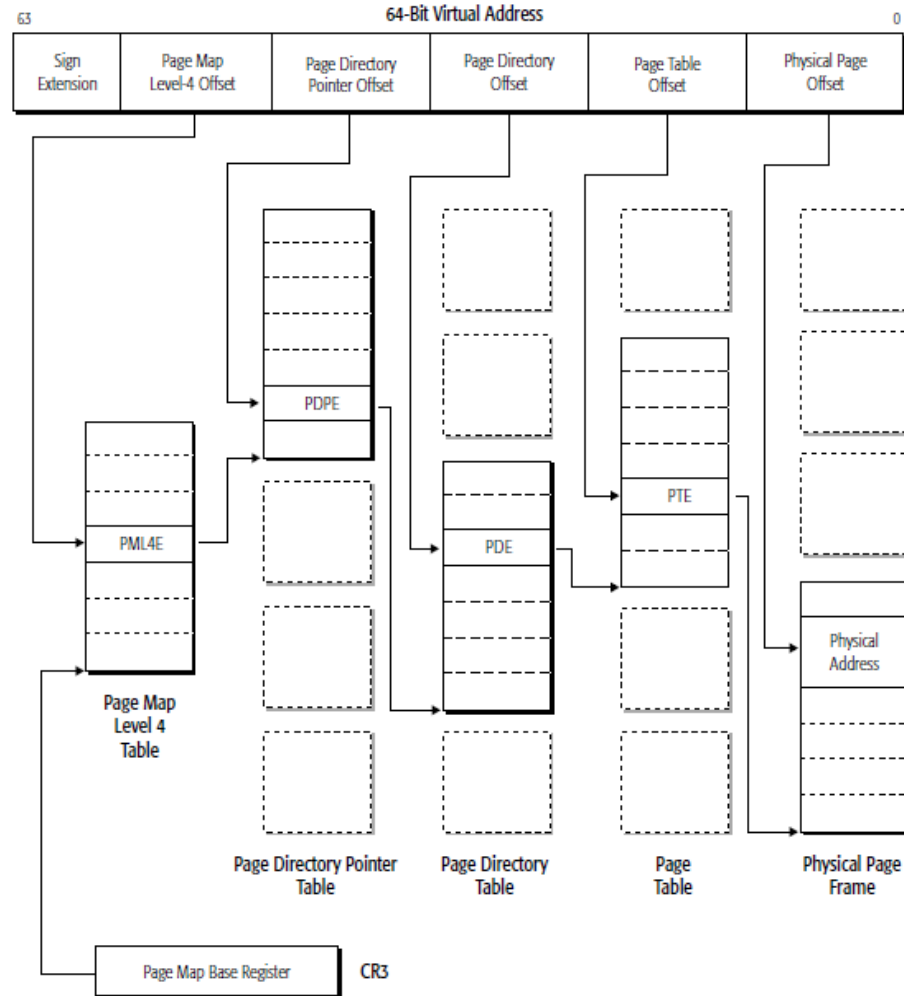


Figure 5-1. Virtual to Physical Address Translation—Long Mode

AMD 64: 4-Kbyte Page Translation

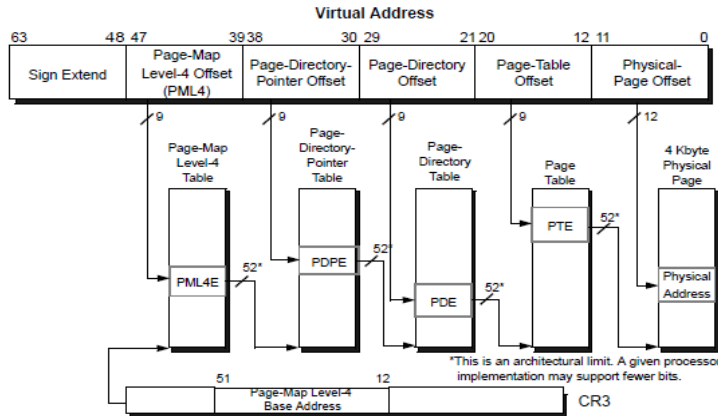


Figure 5-17. 4-Kbyte Page Translation—Long Mode

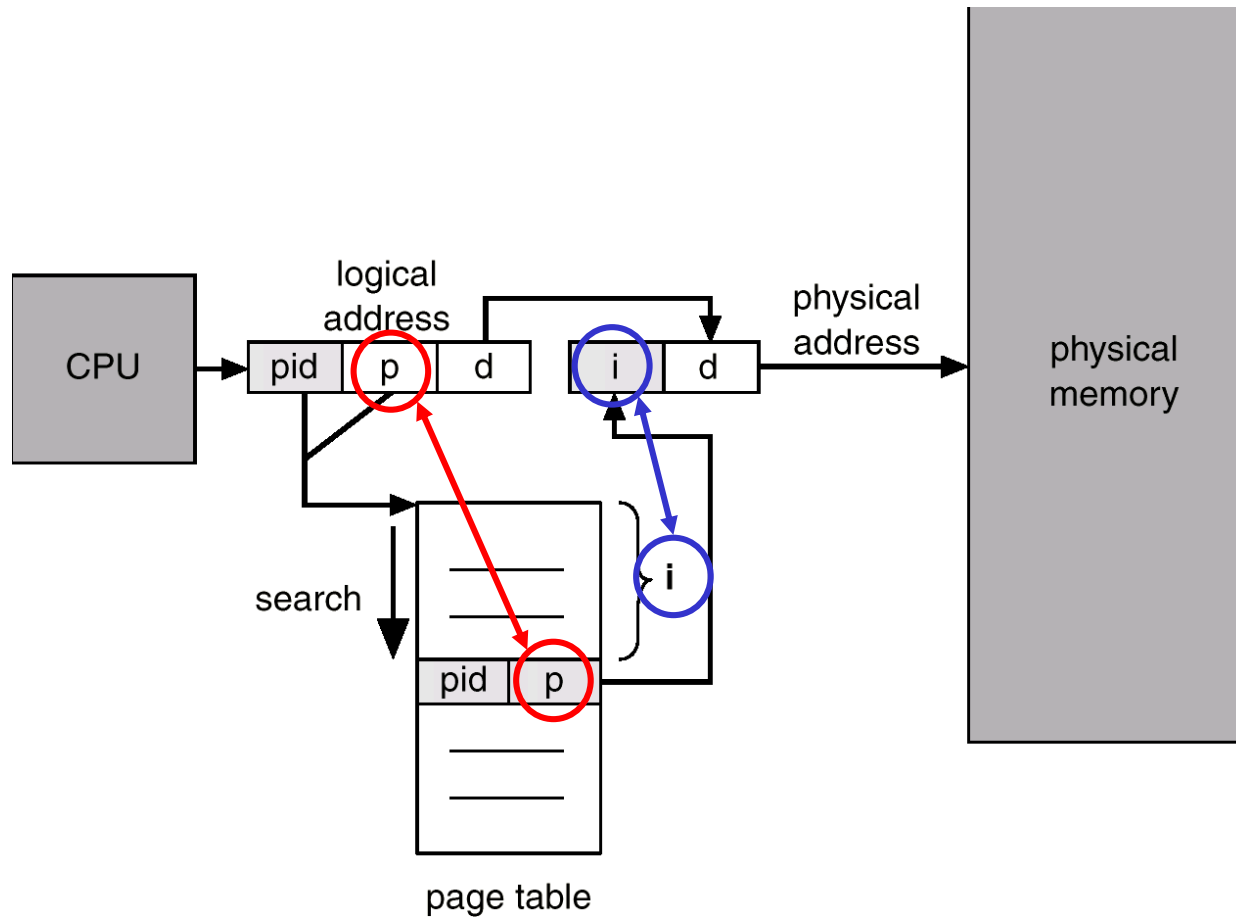
Current implementations only use the 48 bits and not the full 64 bits for virtual address. Hence:

- Bits 63–48 are a sign extension of bit 47, as required for canonical-address forms.
- Bits 47–39 index into the 512-entry page-map level-4 table.
- Bits 38–30 index into the 512-entry page-directory pointer table.
- Bits 29–21 index into the 512-entry page-directory table.
- Bits 20–12 index into the 512-entry page table.
- Bits 11–0 provide the byte offset into the physical page.

Inverted Page Table

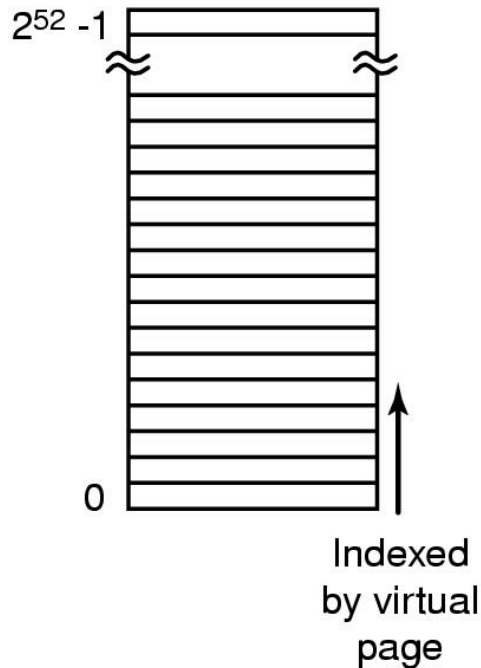
- One entry for each real page of memory; entry consists of the virtual address of the page stored in that real memory location, with information about the process that owns that page
- Decreases memory needed to store each page table, but increases time needed to search table when a page reference occurs
- Use hash table to limit the search to one -- or at most a few page-table entries

Inverted Page Table



Inverted Page Tables

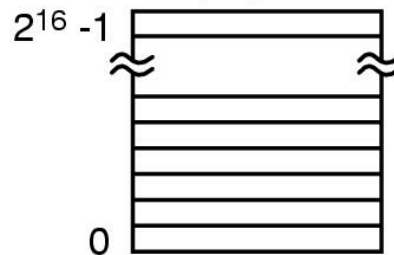
Traditional page table with an entry for each of the 2^{52} pages



64-bit virtual address; 4 KB= 2^{12} KB page size
Virtual page number field has $64-12 = 52$ bit address field

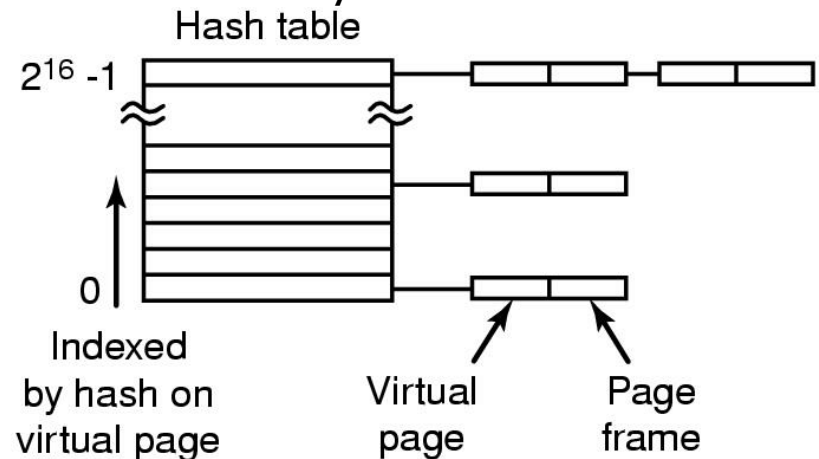
$256 \text{ MB} = 2^{28} \text{ Bytes}$
 $2^{28} / 2^{12} = 2^{16} \text{ page frames}$

256-MB physical memory has 2^{16} 4-KB page frames



Searching inverted page table is slooow...

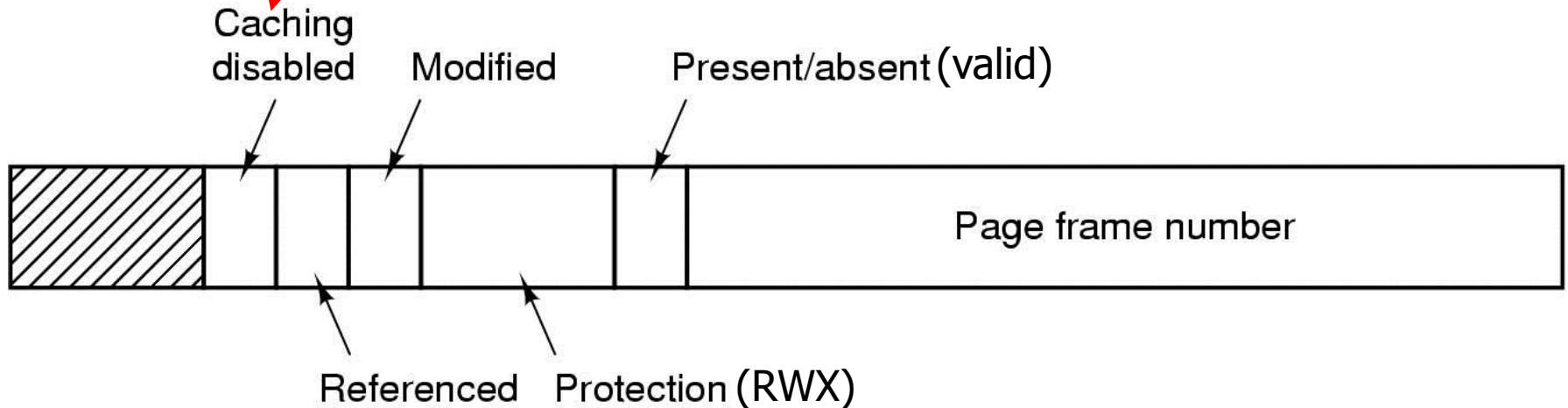
→ Use TLB & search only on TLB misses.



- Comparison of a traditional page table with an inverted page table

What's in a page table entry?

For pages that map to device registers instead of memory.
You don't want to check status in a cached/old copy. So, don't cache this page.



Typical page table entry

The disk address of page on disk is not part of the page table: it is kept in OS owned software tables that are used by page fault handler.

Translation look-aside buffer (TLB)

- All the previous schemes (regular page tables, multi-level page tables, or inverted page tables) require extra memory access to do address translation.
- To avoid extra memory access during address translation, a cache can be used → translation look-aside buffer (TLB).
- TLB is an associative (content-addressable) memory that caches the once translated physical address.
- TLB is on the processor chip.

TLB's are on the processor chip

- HP's Superscalar RISC Processors:
PA 8000

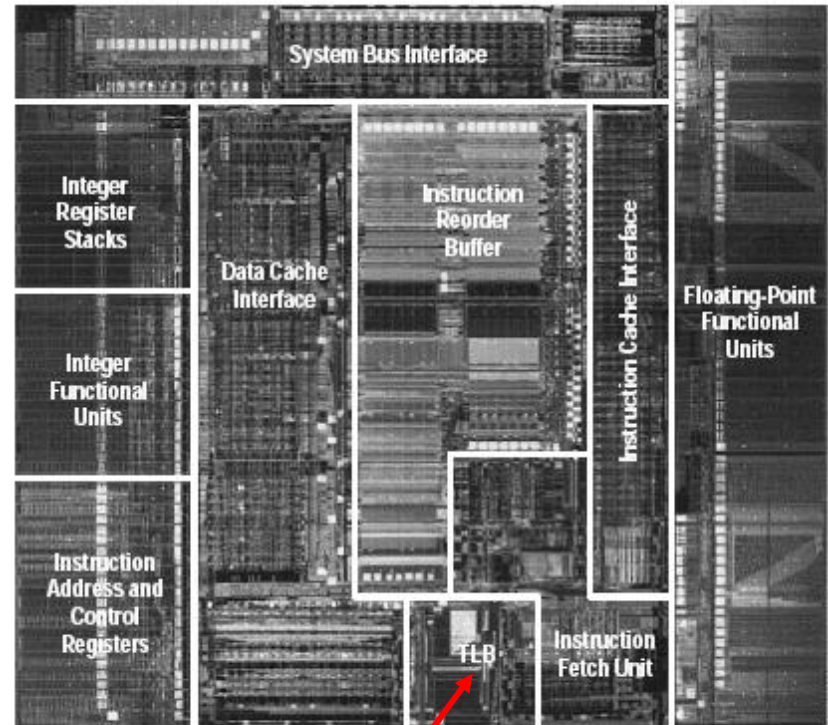
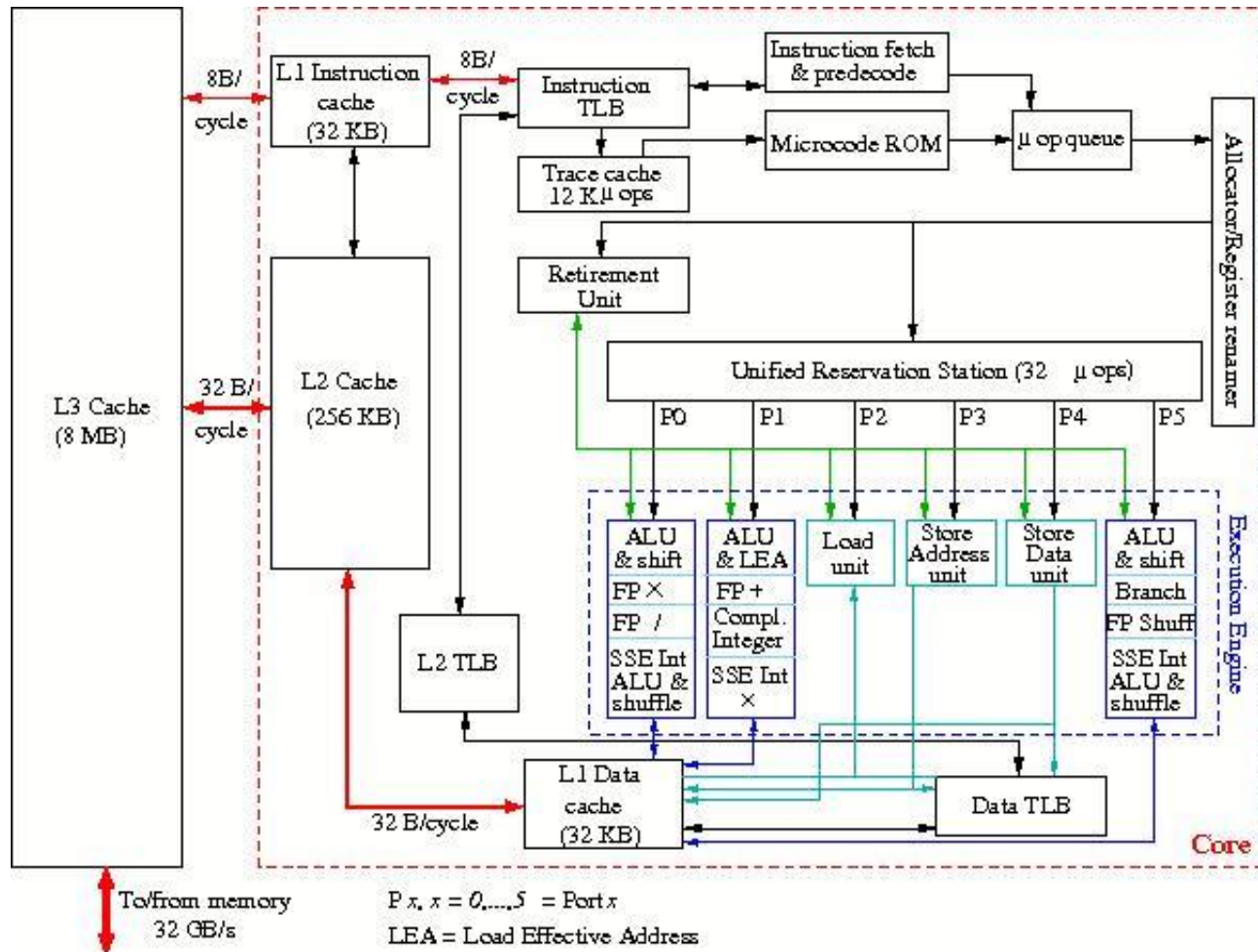


Fig. 2. PA 8000 CPU with major areas labeled.

TLB

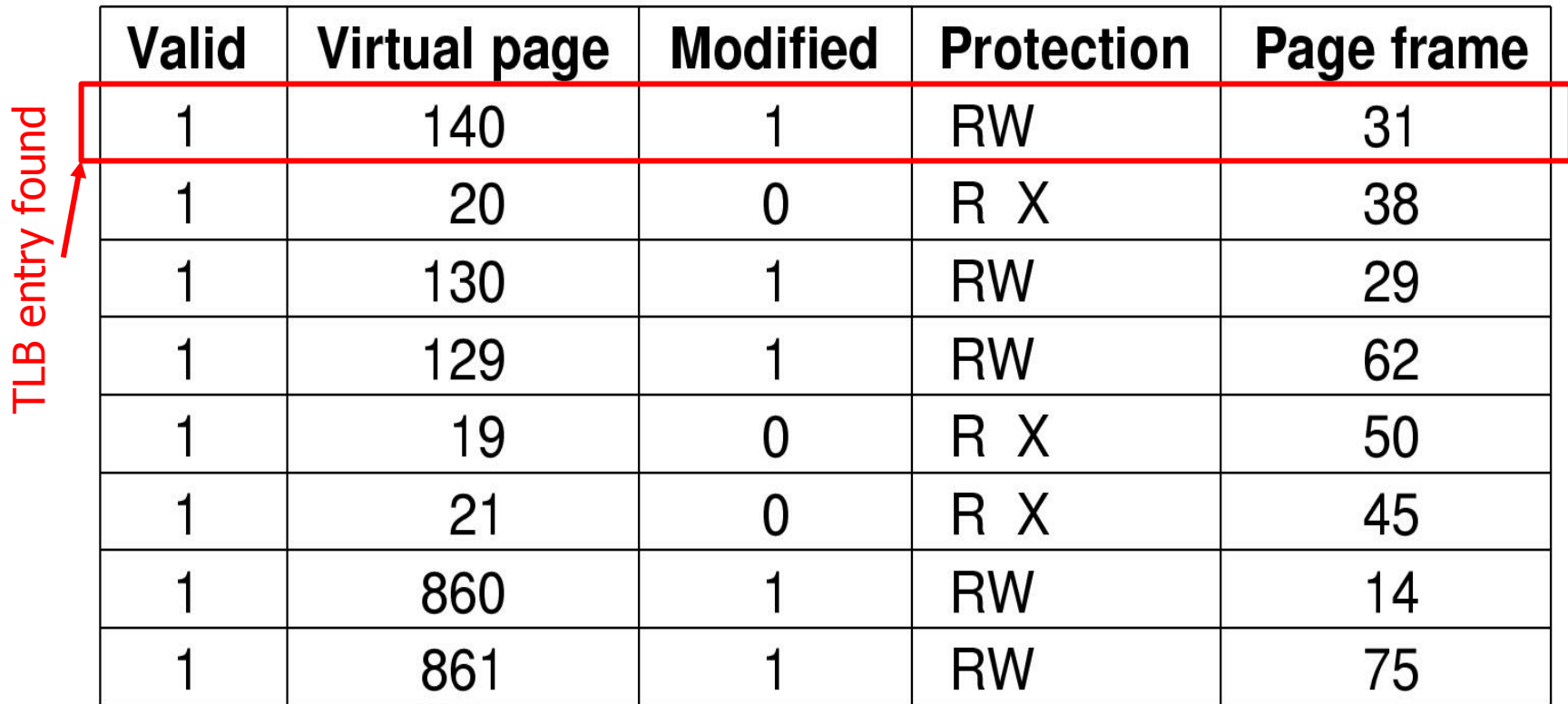
Intel Xeon Nehalem Processor



TLB contents

Example: VPN = 140

Search key



Valid	Virtual page	Modified	Protection	Page frame
1	140	1	RW	31
1	20	0	R X	38
1	130	1	RW	29
1	129	1	RW	62
1	19	0	R X	50
1	21	0	R X	45
1	860	1	RW	14
1	861	1	RW	75

A TLB to speed up paging

Paging Implementation Issues

- Page hit ratio: percentage of time page found in associative memory (TLB)
- If not found in associative memory, must load from page tables: requires additional memory reference

Effective Access Time

- Associative lookup time = ε time units
- Memory cycle -- t
- Hit ratio -- α
- Effective access time

$$\begin{aligned}
 t_{eff} &= \left(\underbrace{t}_{\text{access page}} + \underbrace{\varepsilon}_{\text{access address in cache}} \right) \alpha + \left(\underbrace{t}_{\text{access page}} + \underbrace{t}_{\text{miss, so access address in memory}} + \underbrace{\varepsilon}_{\text{access address in cache}} \right) (1 - \alpha) \\
 &= (2t + \varepsilon - t\alpha)
 \end{aligned}$$

Effective Access Time

- $t_{eff} = 2t + \varepsilon - t\alpha$
 - if $\varepsilon \approx t\alpha$ then the eat will always be 2 memory cycles.
 - If ε is small (small associative access time) and α is large (large hit ratio), then t_{eff} will be close to 1 memory cycle.

Example

- Suppose associative lookup = 20 nsec
- memory access = 100 nsec
- Hit ratio = 80%
- Then effective access time

$$\begin{aligned}t_{eff} &= 0.80 \times (20 + 100) + 0.20 \times (20 + 100 + 100) \\&= 0.80 \times 120 + 0.20 \times 220 \\&= 140\end{aligned}$$

Example continued

- Percent slowdown =

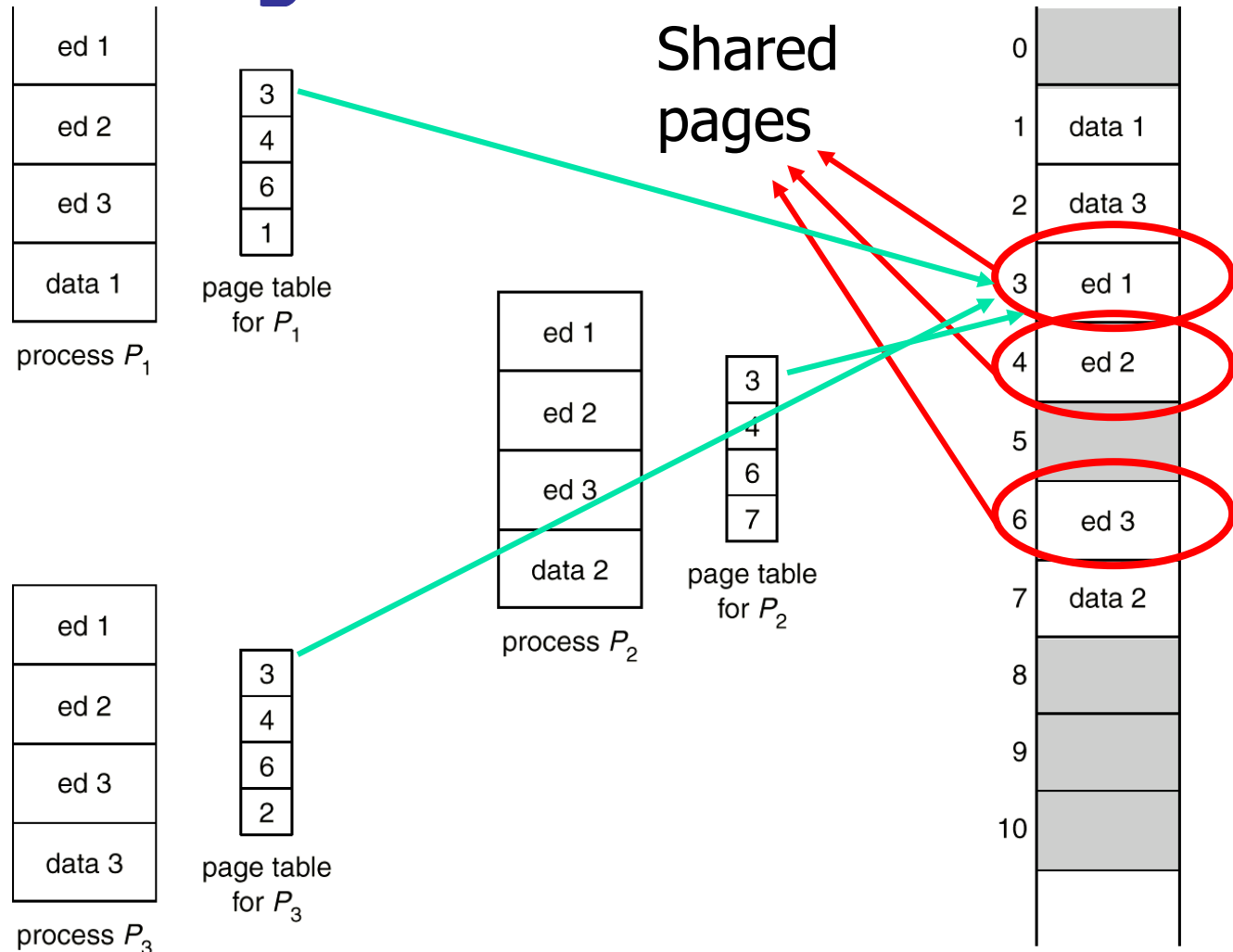
$$\left(\underbrace{140}_{\text{with VM}} - \underbrace{100}_{\text{without VM}} \right) / 100 = 40\%$$

- That is a lot of slowdown!
 - Implies, the hit ratio must be very high—80% is not high enough!

Sharing Pages

- Code and data can be shared by mapping them into pages with common page frame mappings
- Most OS's reserve a certain part of the VM address space for sharing.

Shared Pages



Shared pages

- If virtual addresses are stored anywhere in the code or data, page numbers must correspond.
- What happens when a page is swapped in and out?
 - You have to update all process page tables that are sharing it. → **a lot of work!**
 - Implies that practically you don't swap shared pages out.

Protection

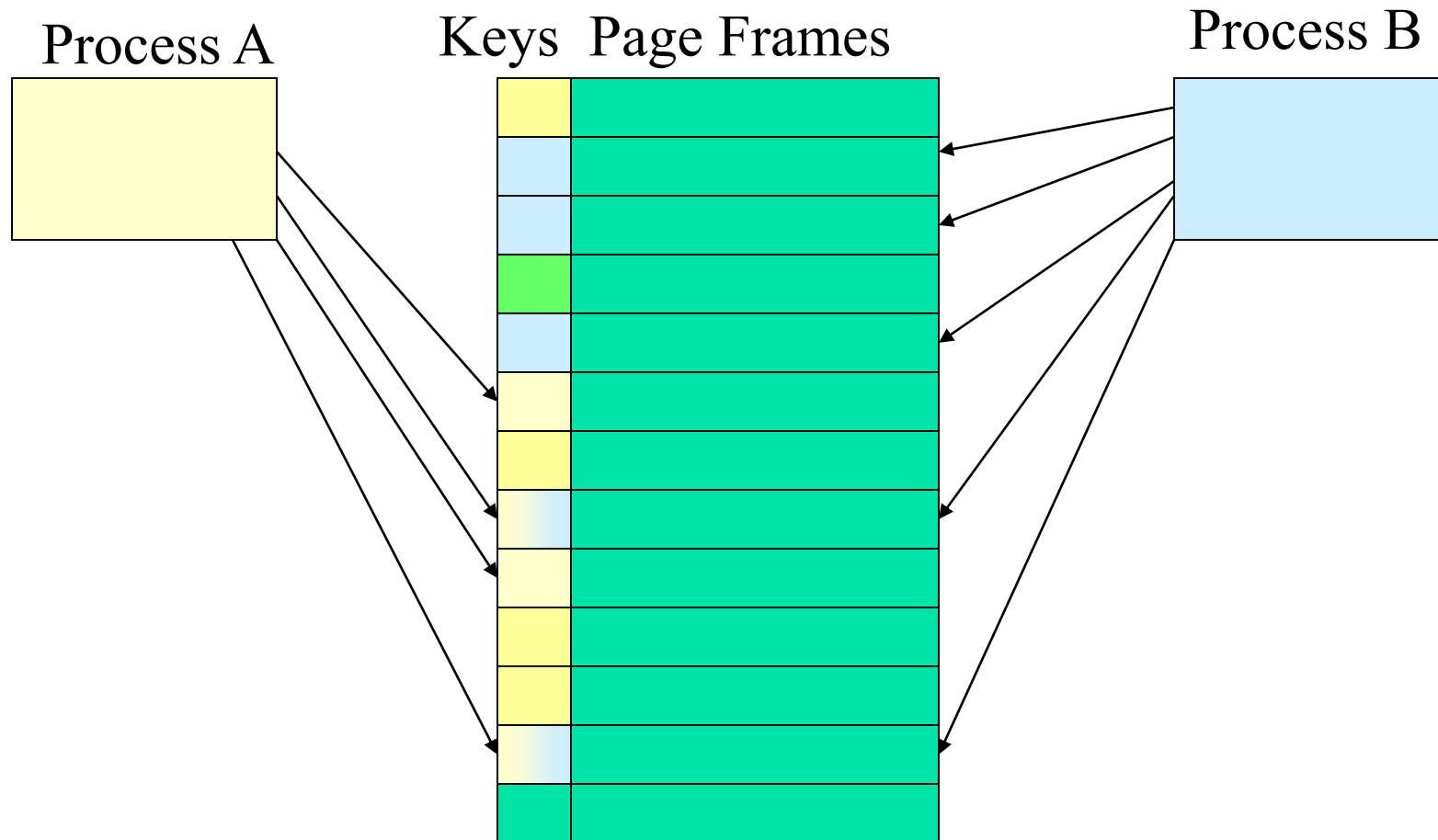
- Can add **read, write, execute protection** bits to page table to protect memory
- Check is done by hardware during access
- Can give shared memory location different protections from different processes by having different page table protection access bits.

Protection

- Alternatively, can associate protection **lock** with page frame. Each process has its own **key**.
- If the key fits the lock, the process may access the page frame
- Typically many different keys can fit a lock using a priority numbering scheme
- (e.g. Key 3 fits all locks 3, 7, 15
Key 4 fits 5,6,7,12,13,14,15.)

Can you figure out how this works?

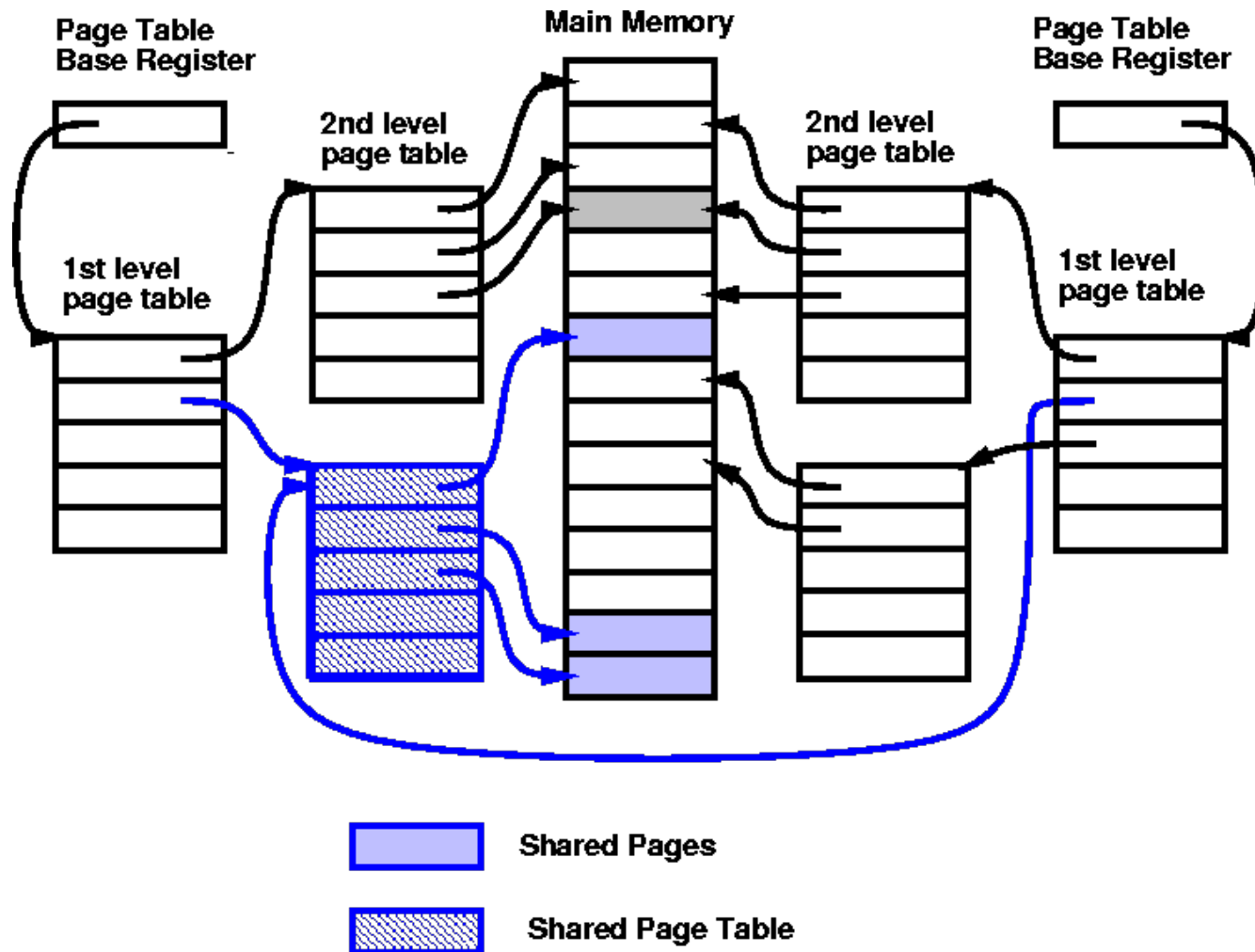
Keys and Locks



Sharing in Two Level Paging

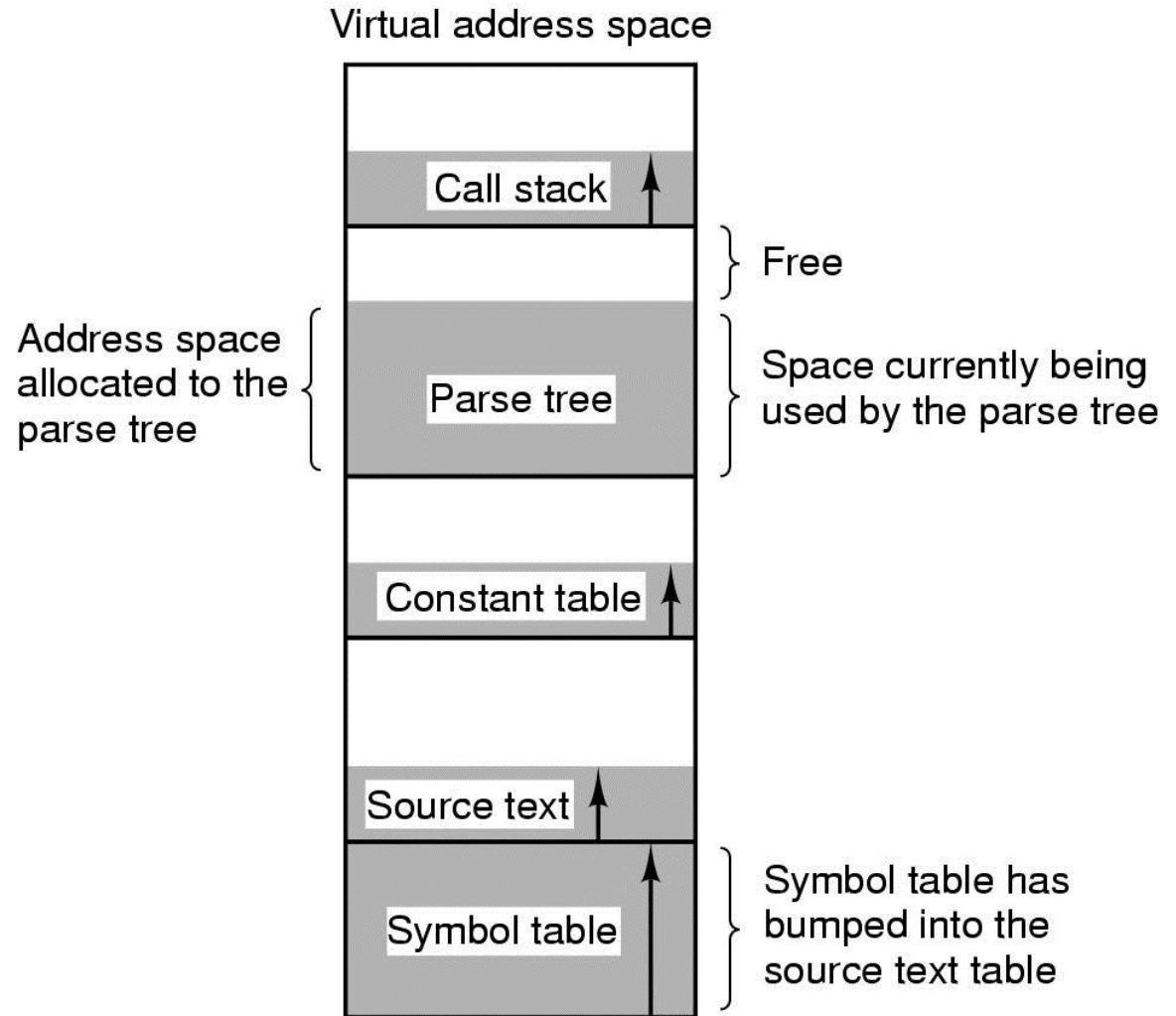
- Allows sharing of data by using first level page tables of different VM's to share second level page table
- Allows second level page tables to be paged

Sharing in Two Level Paging



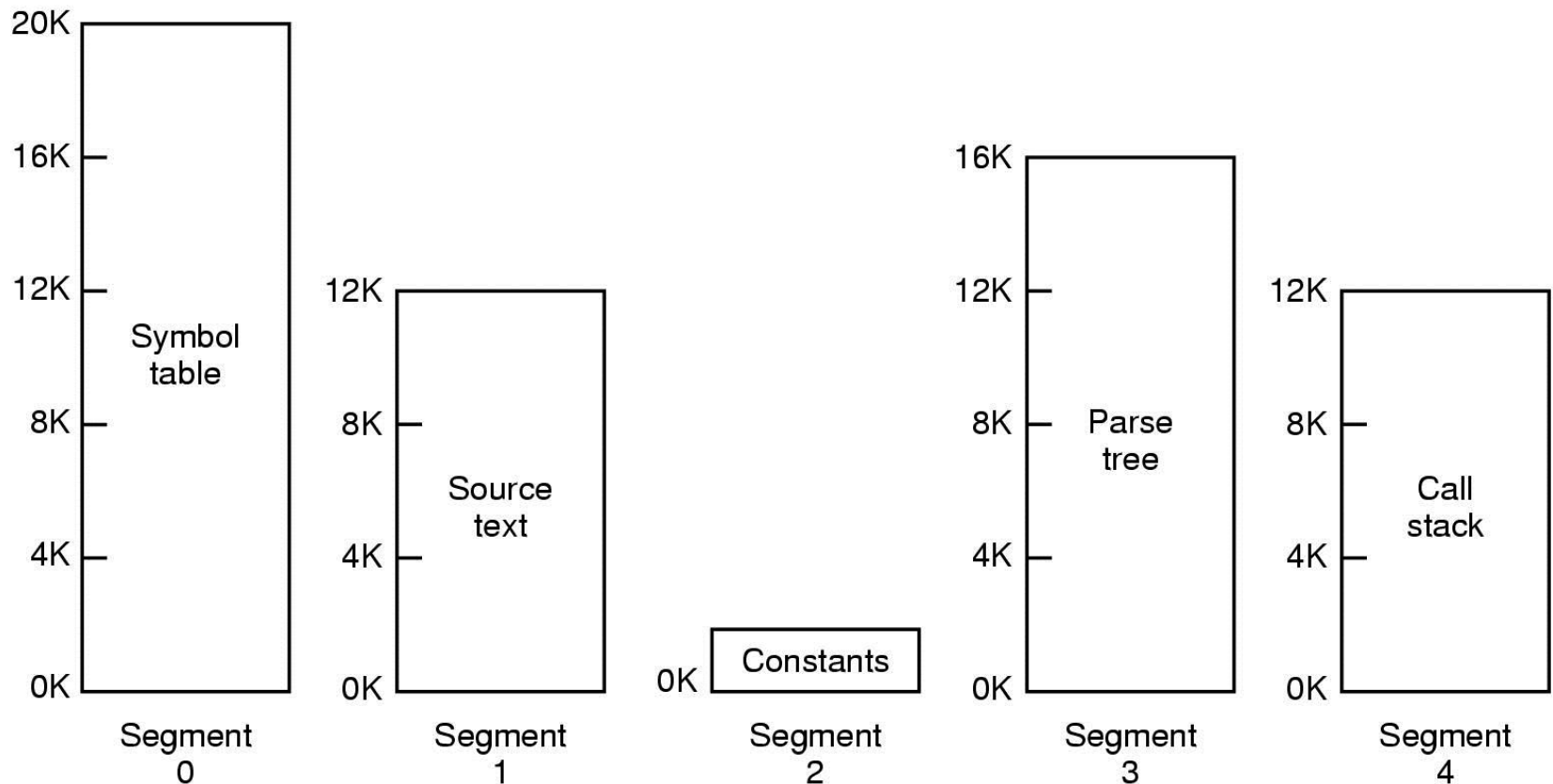
Segmentation-motivation

- With a single address space, managing growing regions becomes difficult.
- One-dimensional address space with growing tables
- One table may bump into another



Segmentation

- Allows each table to grow or shrink, independently



Segments

- Provide a user with a two dimensional virtual memory instead of a linear virtual address space.
 - Dimension 1: a collection of segments
 - Dimension 2: an address space or segment
 - Addresses look like (s,d)
 - Each VM can be different sizes
 - (virtual) pages are contiguous in memory, not so with segments

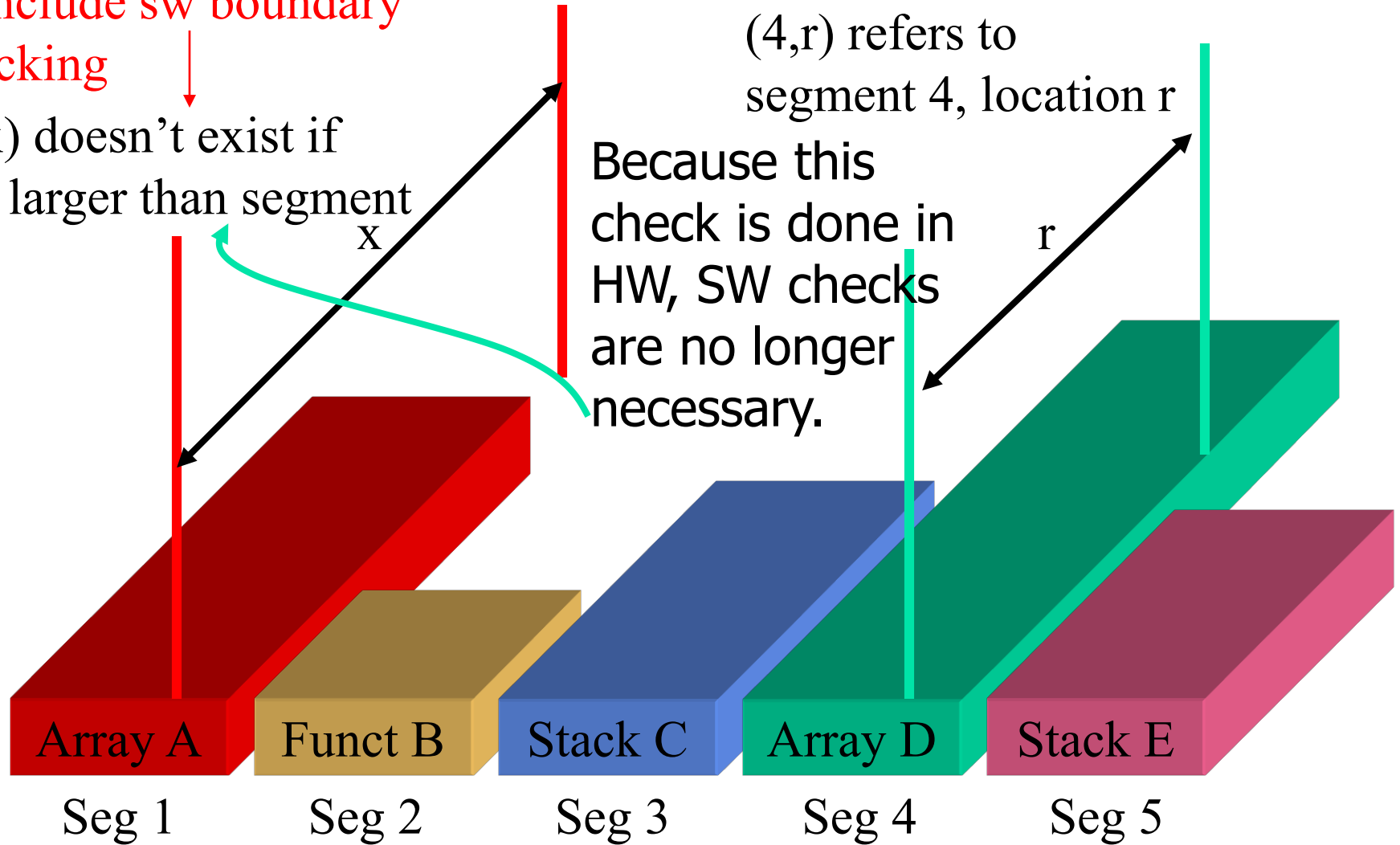
Segmentation

This check can be done
by hardware, so no need
to include sw boundary
checking

(1,x) doesn't exist if
x is larger than segment
x

(4,r) refers to
segment 4, location r

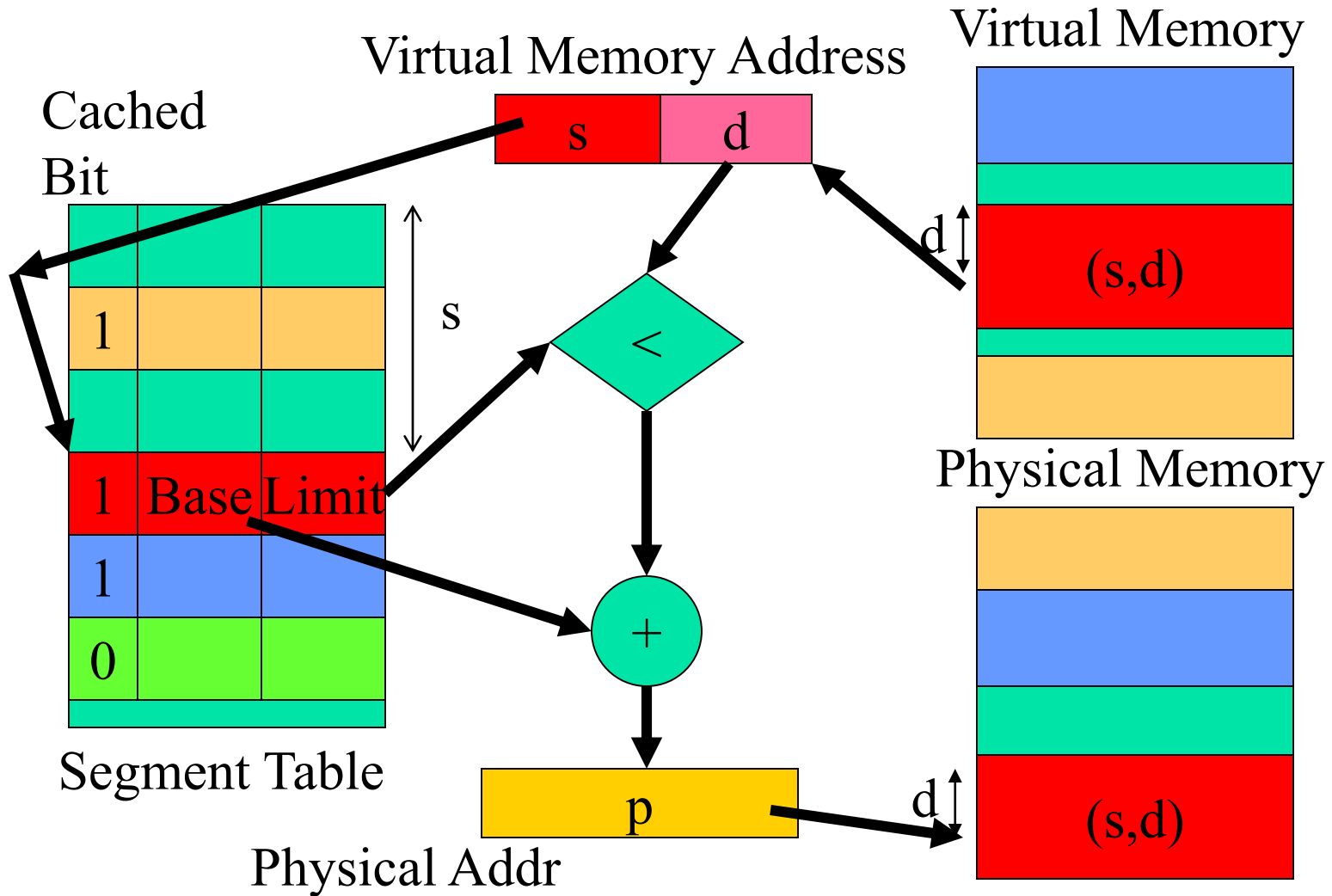
Because this
check is done in
HW, SW checks
are no longer
necessary.



Use of Segments

- Segment for each user entity: stack, code, data, array
- Protect each user entity independently: stack, code, data, array
 - can have separate read, write, etc protection (unlike pages)
- Allow each segment to grow independently

Segment Mapping Hardware



Use of Segments

- Avoid paging inefficiency: many pages include data representing programming entities that may not be used immediately
- Many attempts to study efficiency of paging vs. segmentation
 - so far results inconclusive

Segmentation Implementation Issues

- Can cache segment table in registers
- Or can keep segment table in memory at per process location given by a **segment table base register**
- Length of segment table given by **segment table length register**
 - we need this, because the number of segments might vary by process.
 - This is not the case for paged memory: size of the page table is determined by the virtual address space and page size.

Segmentation Implementation Issues

- Segment table base register and segment table length register changed at context switch time
- Caching scheme (TLB) uses
 - Associative registers/content addressable memory, etc
- Segment hit ratio: percentage of time segment found in associative memory

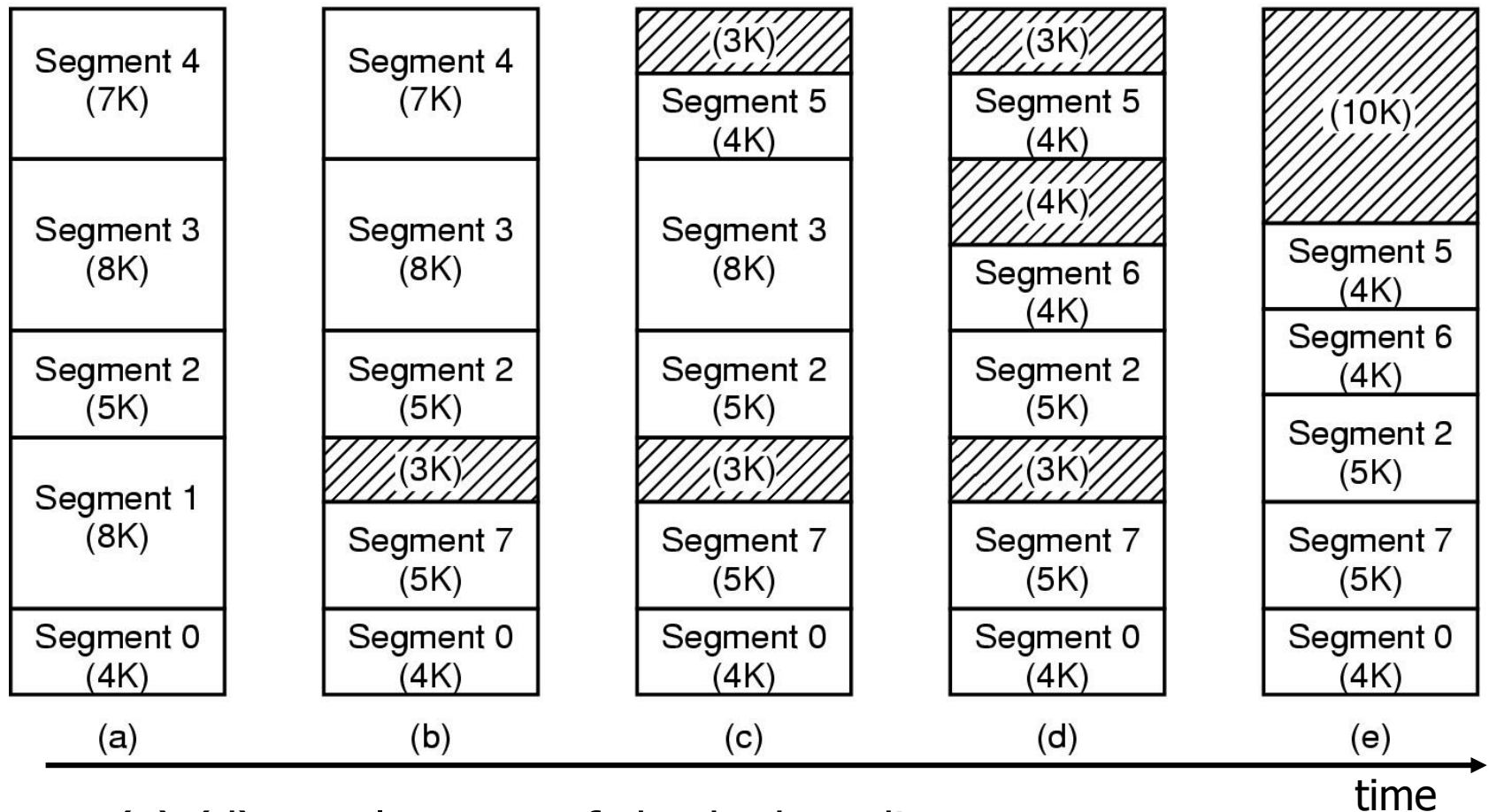
Segmentation Implementation Issues

- If not found in associative memory, must load from segment tables:
 - Requires additional memory reference
- Suffers external fragmentation - use compaction? How to do this?

Two ways:

- Move data in memory
- incremental compaction while segments are being swapped in and out. Swap them out and swap them in into new locations.

Segmented memory suffers from external fragmentation



- (a)-(d) Development of checkerboarding
- (e) Removal of the checkerboarding by compaction
- pure segmentation just like variable partitioning

Segmentation/paging comparison

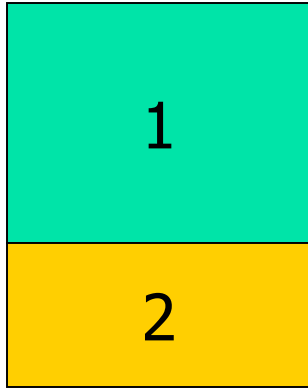
Consideration	Paging	Segmentation
Number of address spaces	1	Many
Can the total address space exceed the size of physical memory?	Yes	Yes
Can procedures and data be distinguished and separately protected?	No	Yes
Why was this technique invented?	To get a large linear address space without having to buy more physical memory	To allow programs and data to be broken up into logically independent address spaces and to aid sharing and protection

Discussion

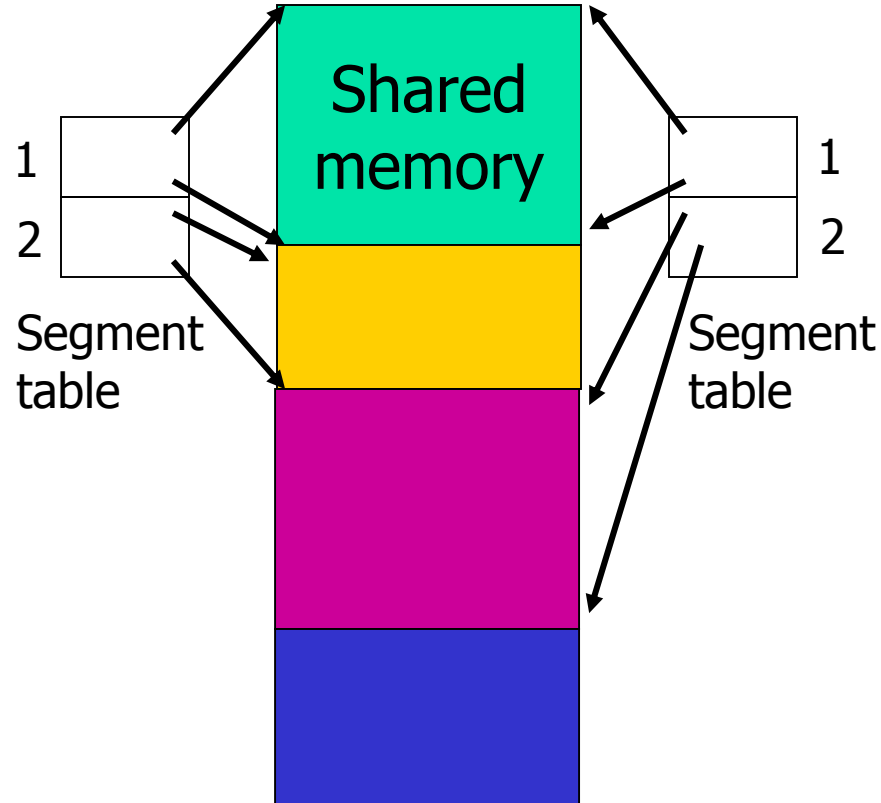
- Representing objects at run-time issues:
 - Protection
 - can be done just like paging system: protection associated with pages or lock and key associated with frames.
 - Sharing
 - with segmentation no forced protection or sharing. Just protect or share what you want.

Sharing Segments

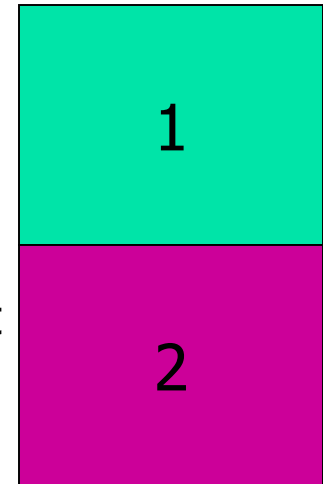
Logical segmented memory



Physical memory



Logical segmented memory



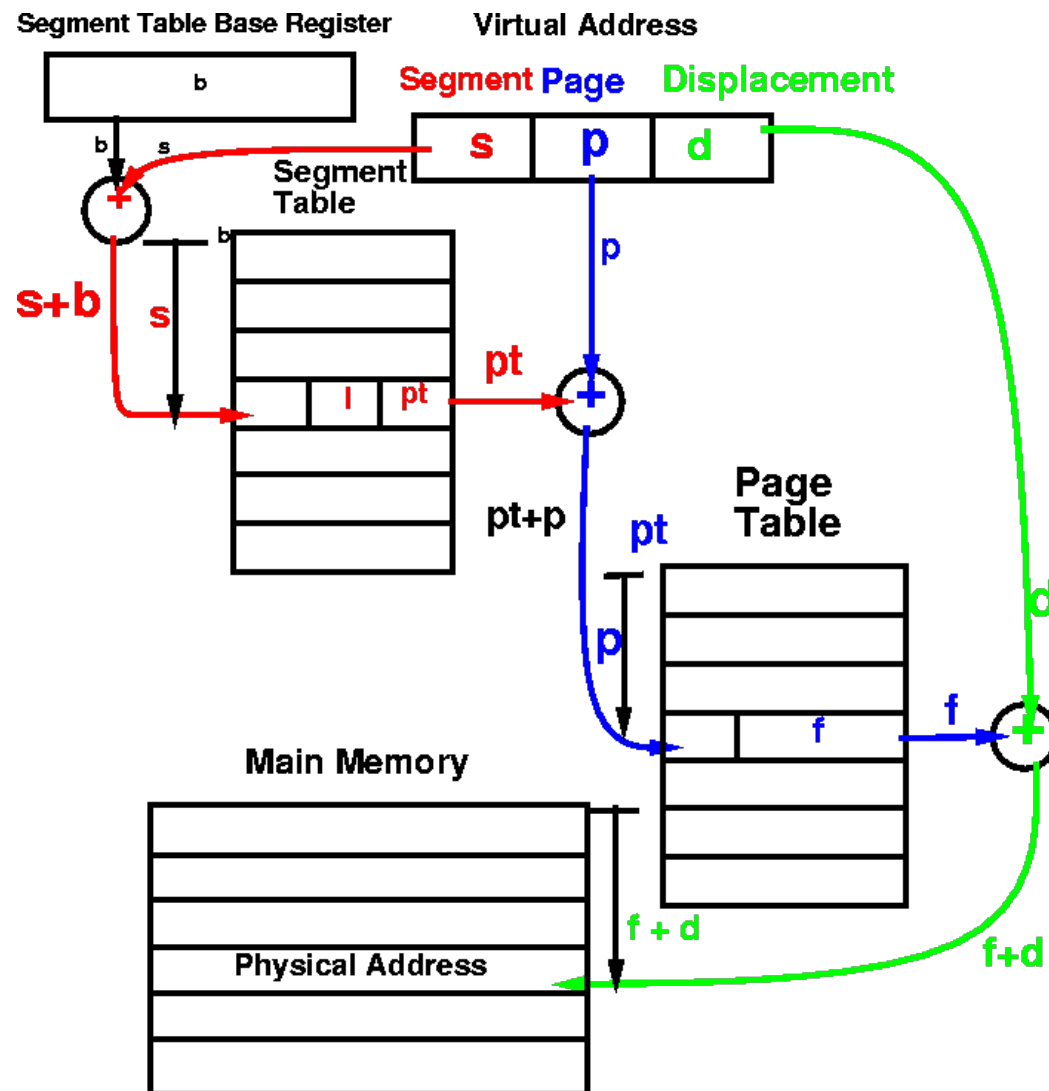
Protection

- Can add protection information to segment table entries to protect memory
- Check is done by hardware during access
- Can give particular representation of software construct its own protection
- Shared segments can have different protections from different processes by having different segment table protection access bits

Hybrid Schemes: Segmented Paged Virtual Memory

- If segments are too big to fit into memory, **segments can be paged**.
- Overcomes external fragmentation problem of segmented memory
- We don't need to worry about compaction.
- On average, half a page per segment is wasted by internal fragmentation (the last page)
- Allows sharing of software components
- Allows two dimensional view of memory
- Reduces in-memory page table size

Segmented Paged Virtual Memory



Paged memory address translation steps

1. Look-up VA in the TLB
2. If TLB hit → get PA from TLB; done
3. If TLB miss → look-up VA in the page table;
 1. If valid bit =1, then get PA from PT; done
 2. If valid bit =0, raise page-fault exception.

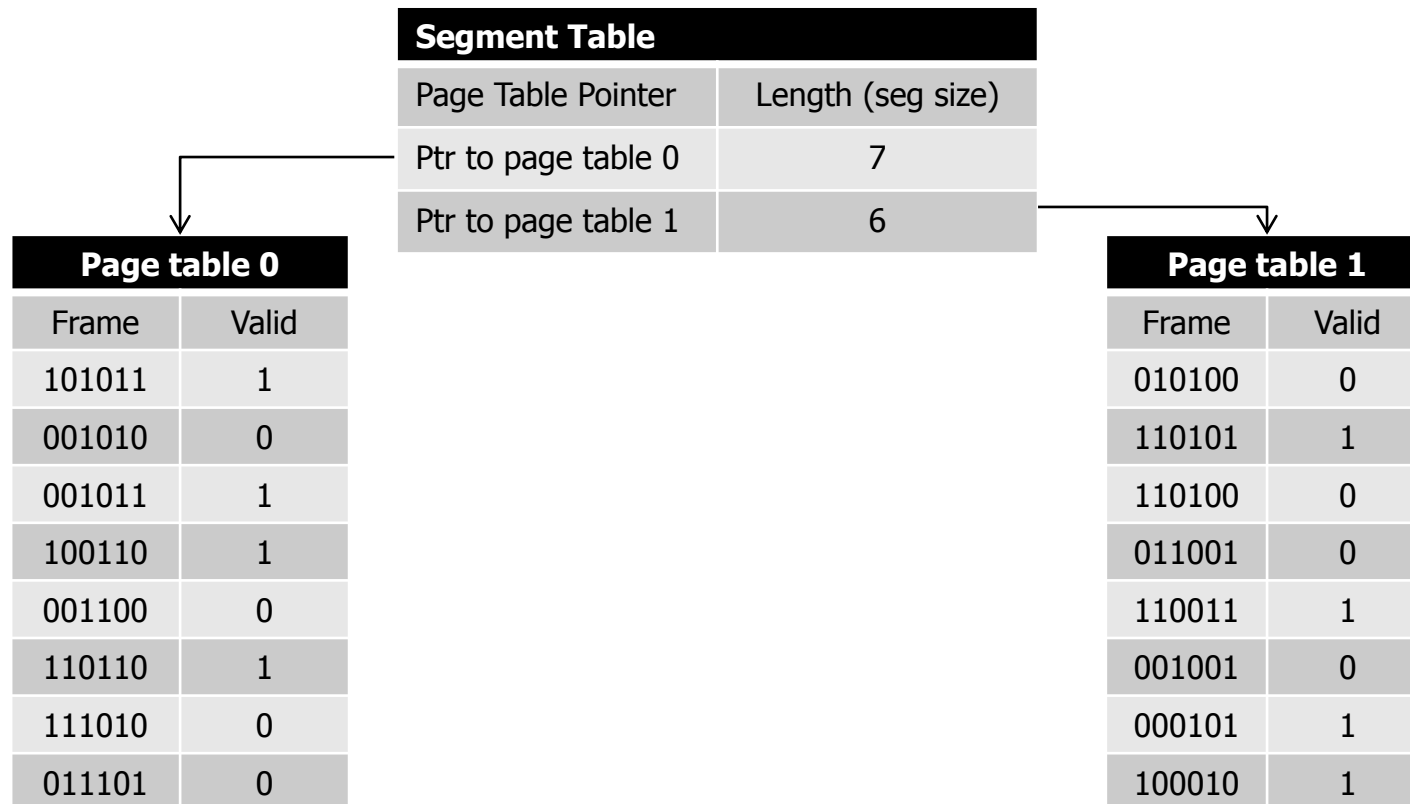
Segmented memory address translation steps

1. Look-up VA in TLB
2. If TLB hit → get PA from TLB; done
3. If TLB miss → look-up VA in segment table (ST)
 1. If valid bit = 1
 1. if $VA < \text{bounds}$ for this segment then $PA = \text{physical segment address} + \text{offset}$
 2. If $VA > \text{bounds}$ then raise segmentation violation exception
 2. If valid bit = 0 then raise segment-fault exception

Exam question

- A system using demand-paged segments has a 16-bit virtual address space with 2 segments per process and a page size of 2^{12} bytes. The content of the segment and page tables is specified below (all values binary). Segment length is in increments of the page size.

Segment and page tables

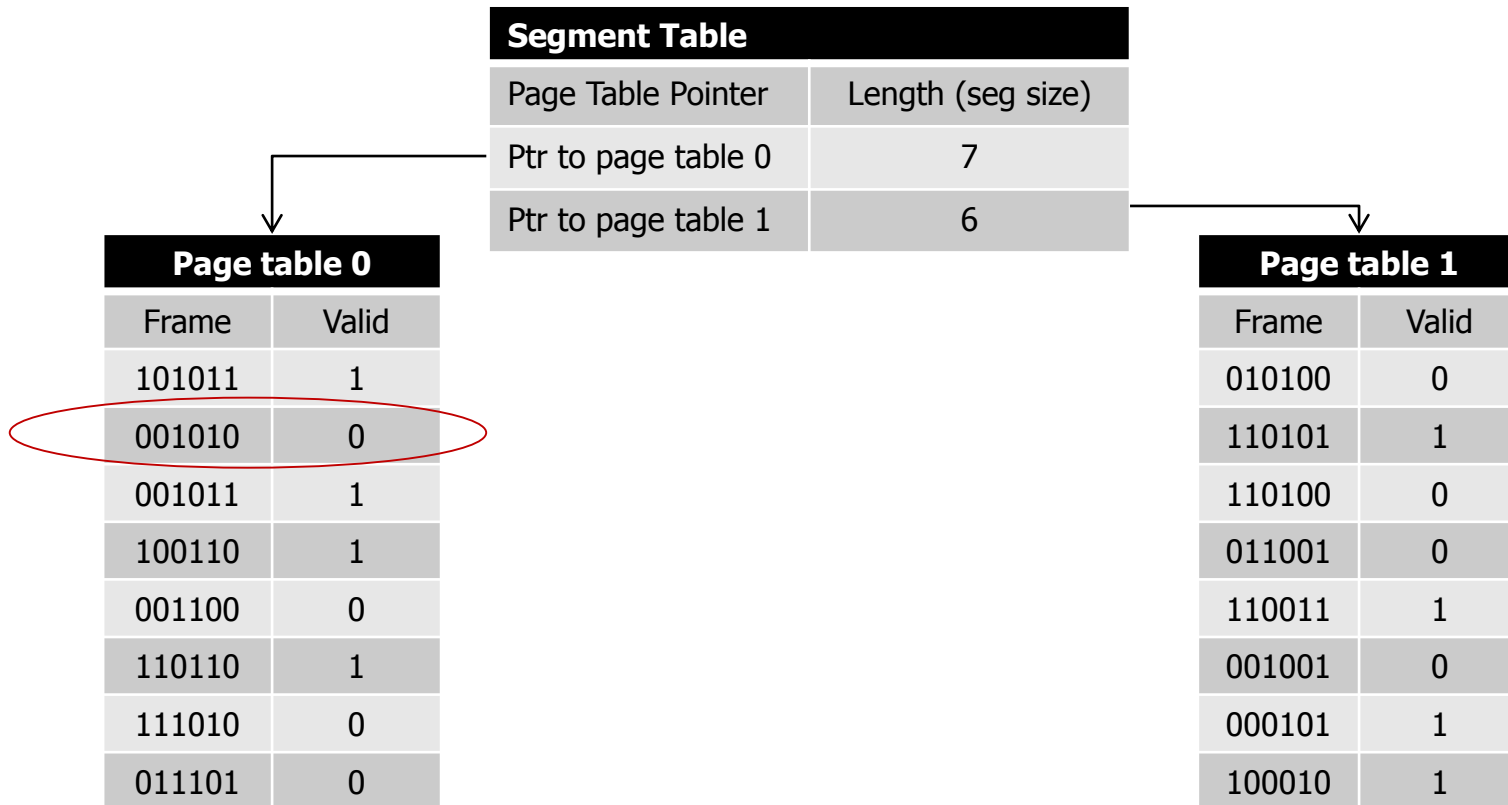


- Find translation for virtual address
 - 0001 0100 0101 0111

Divide address into its parts

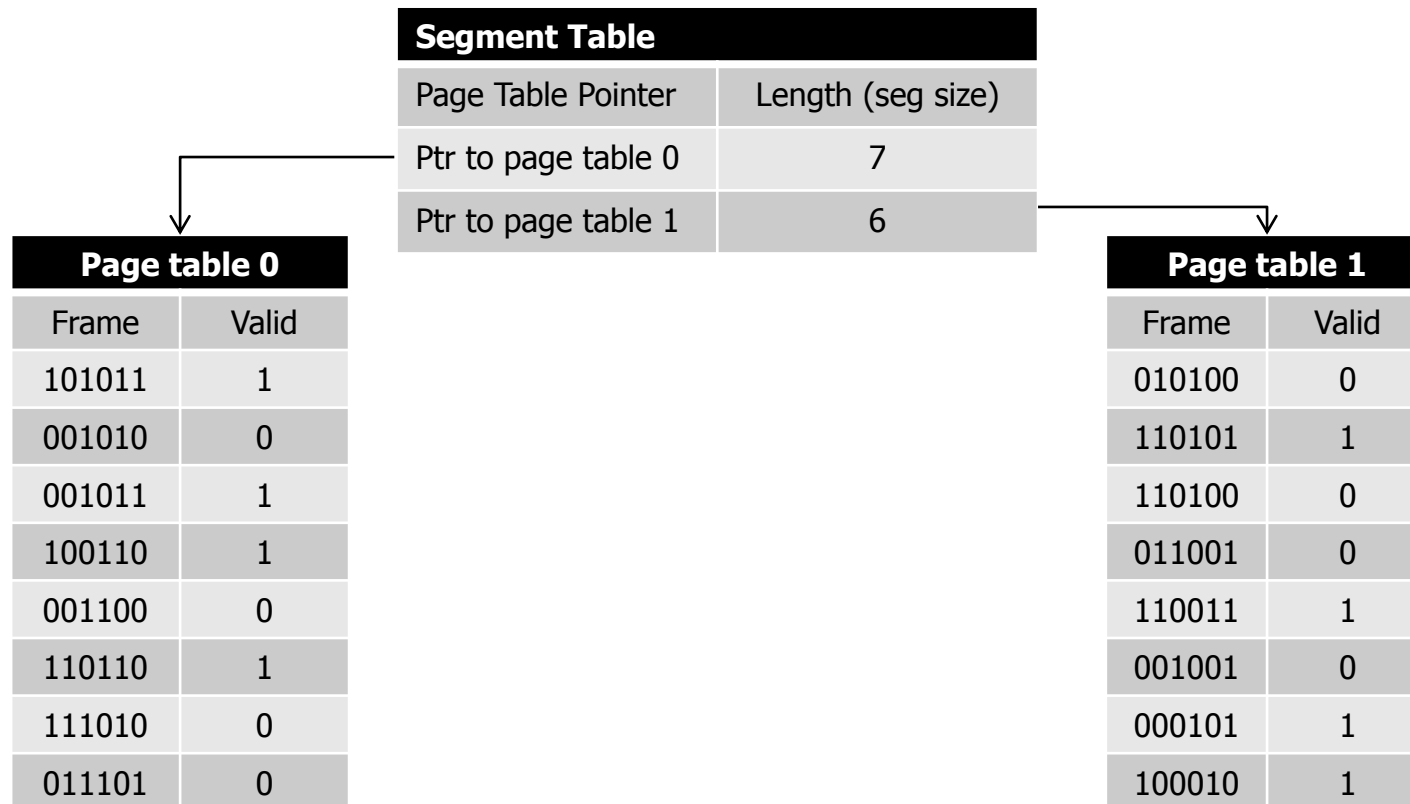
- 2 segments: segment identifier needs 1 bit
- Page size 2^{12} : 12-bits for offset
- Page tables have 8 entries: 3 bits
- So, the virtual address is divided into
 - 0, 001, 0100 01010111
 - Segment 0
 - Page 001
 - Offset 010001010111

Segment and page tables



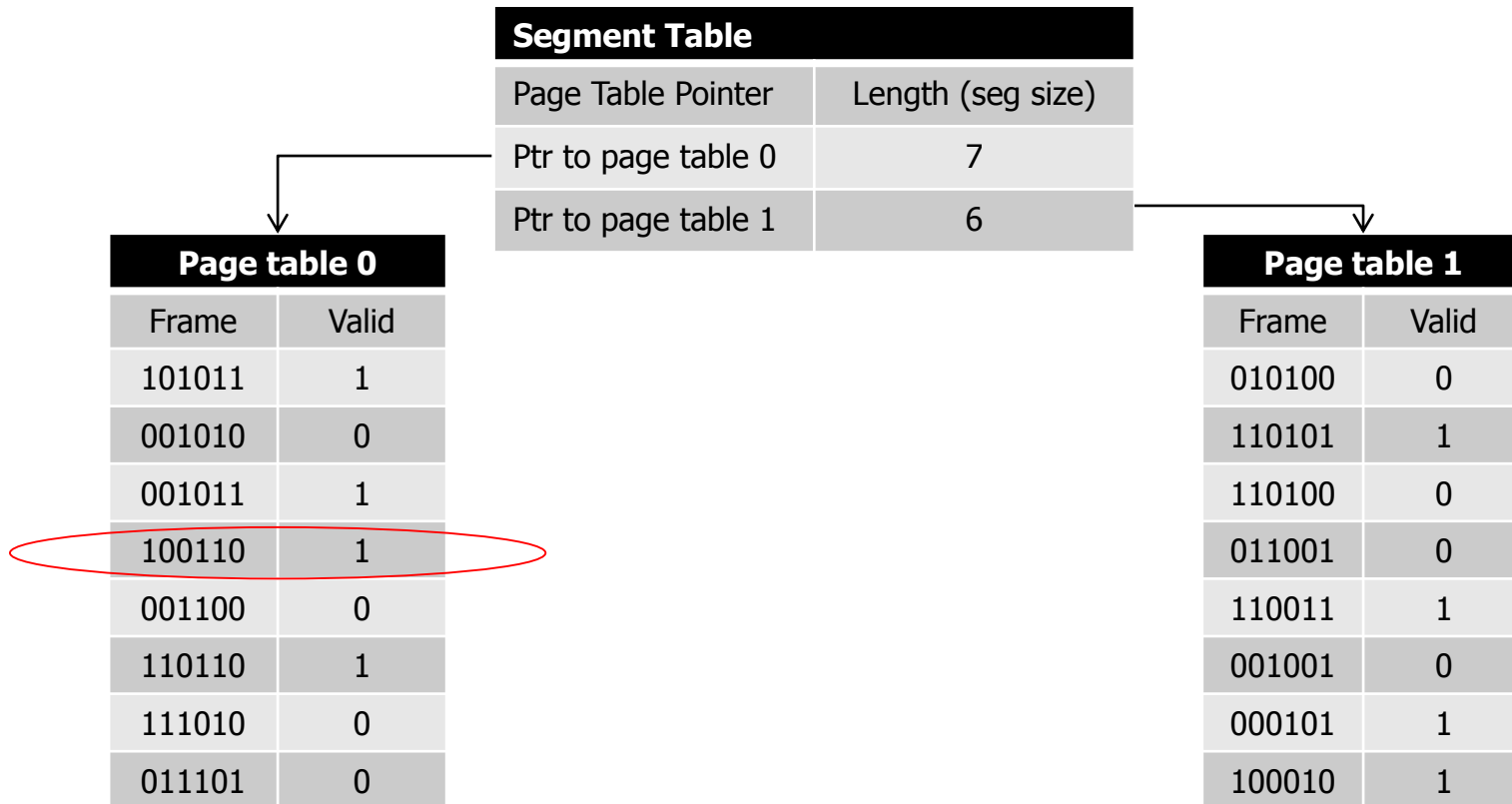
- Segment # 0 and page 001 → valid bit = 0
 - Page is not in memory → this address generates a page fault.

Segment and page tables



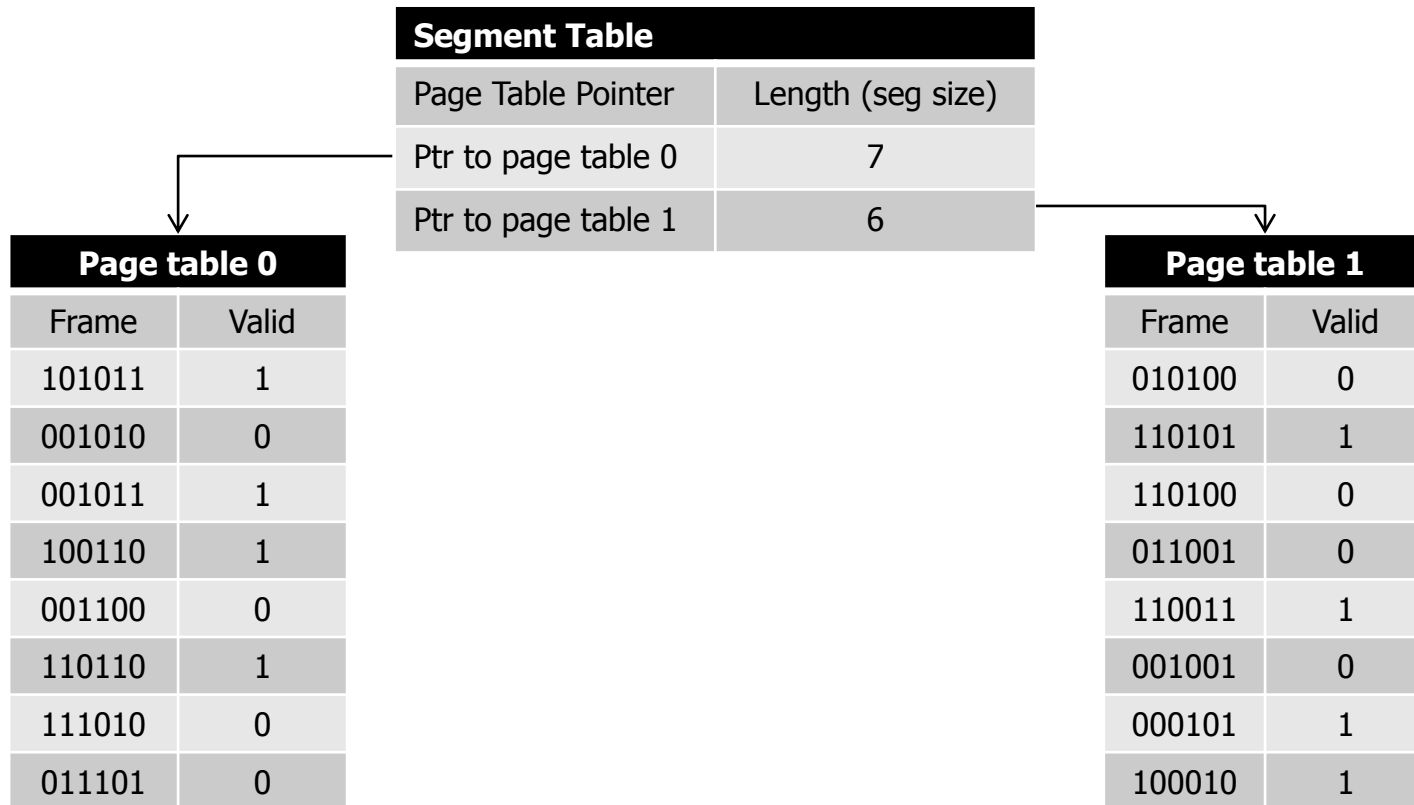
- Find translation for virtual address
 - 0011 0010 1100 0111

Segment and page tables



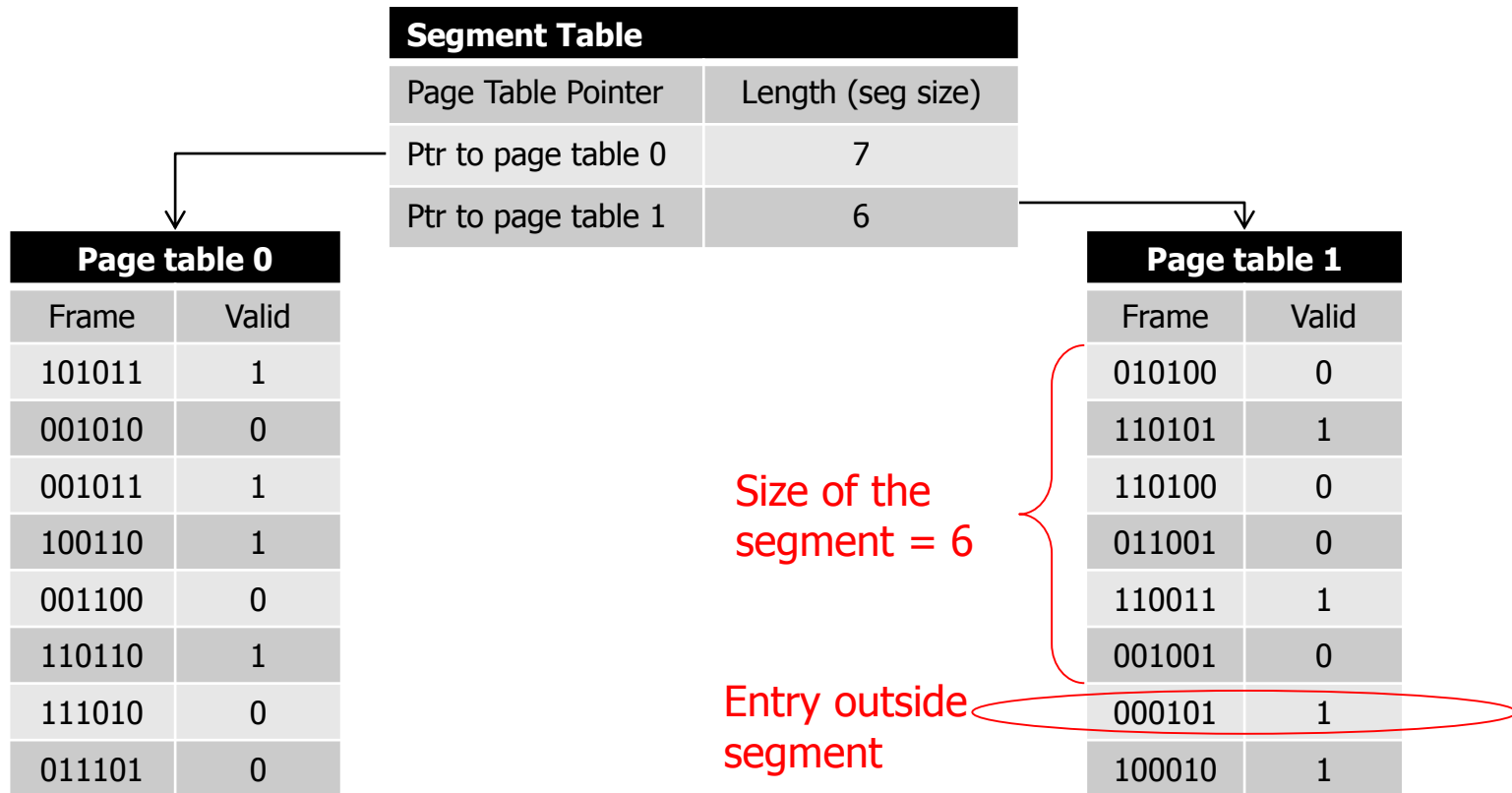
- Divided into its parts: S, P, D (segment, page, displacement)
 - VA = 0,011,0010 11000111
 - Seg # = 0, page # = 3
 - Entry 3 in PT for segment 0 is 100110 and the valid bit is 1
 - PA = Frame # concat with offset = 100110 001011000111

Segment and page tables



- Find translation for virtual address
 - 1110 0100 1111 1111

Segment and page tables

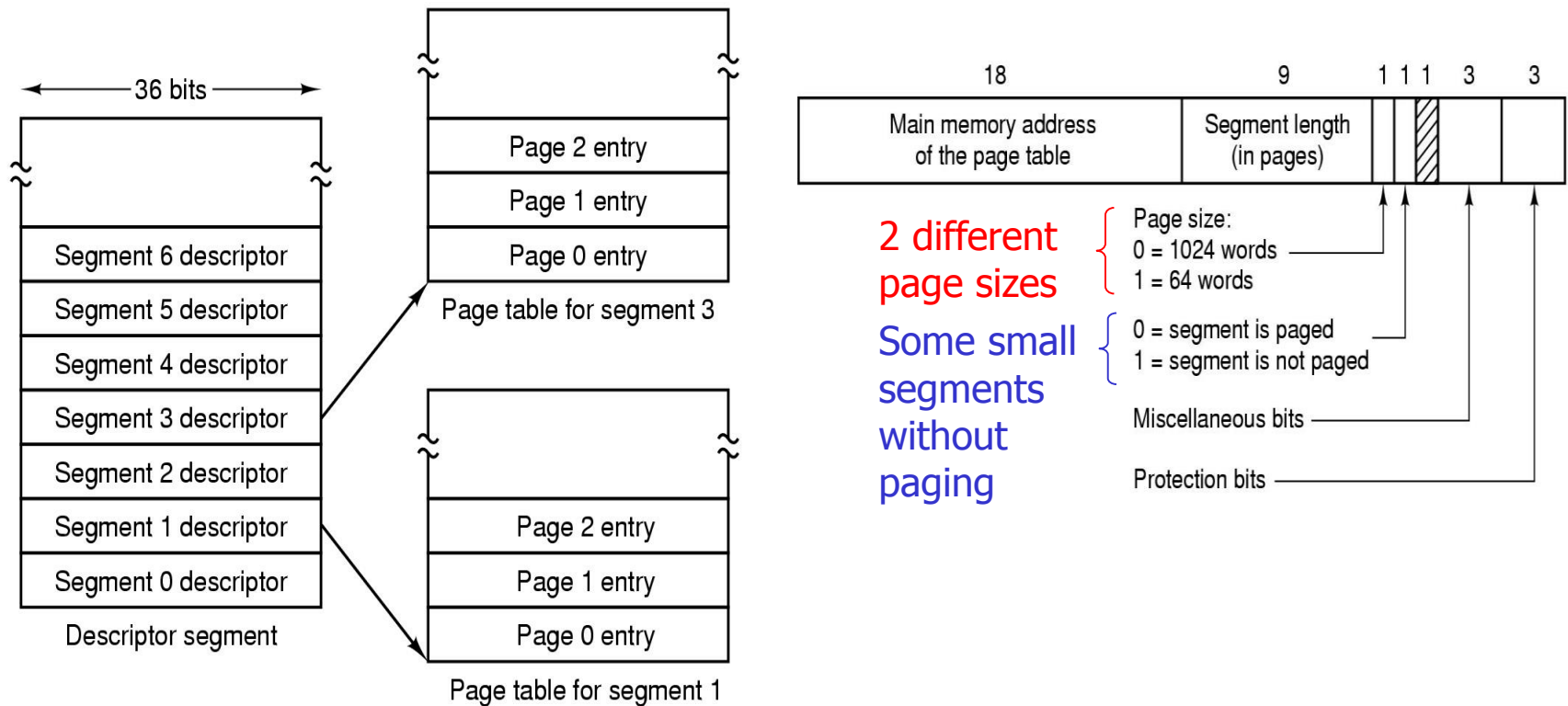


- Divided into its parts: S, P, D (segment, page, displacement)
- 1,110, 0100 1111 1111 → segment 1, page 6.
- The length of the page table for segment 1 = 6, but page 6 is the 7th entry in the page table (starting from 0).
- So, this VA generates segmentation fault.

Real-life examples

- MULTICS – first large scale time-shared system developed at MIT.
 - Many modern ideas about OS were developed
- Memory management architecture of Intel Pentium chips.
 - Segmented paged memory possible.
 - Popular OS's (linux, windows) implement just paged memory on these chips.

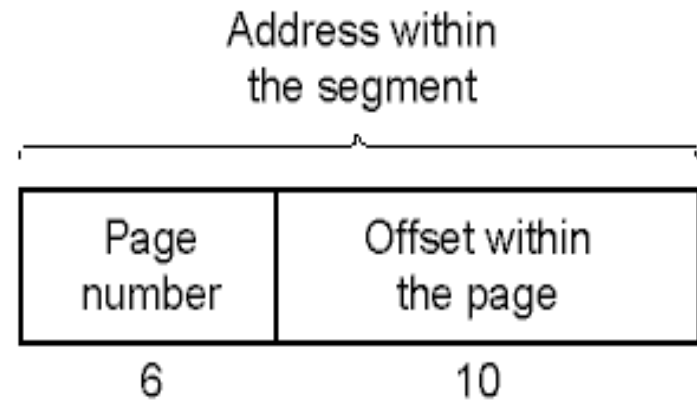
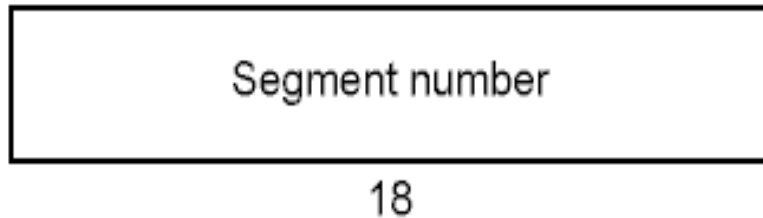
Segmentation with Paging: MULTICS



- Descriptor segment points to page tables
- Segment descriptor – numbers are field lengths

Segmentation with Paging: MULTICS

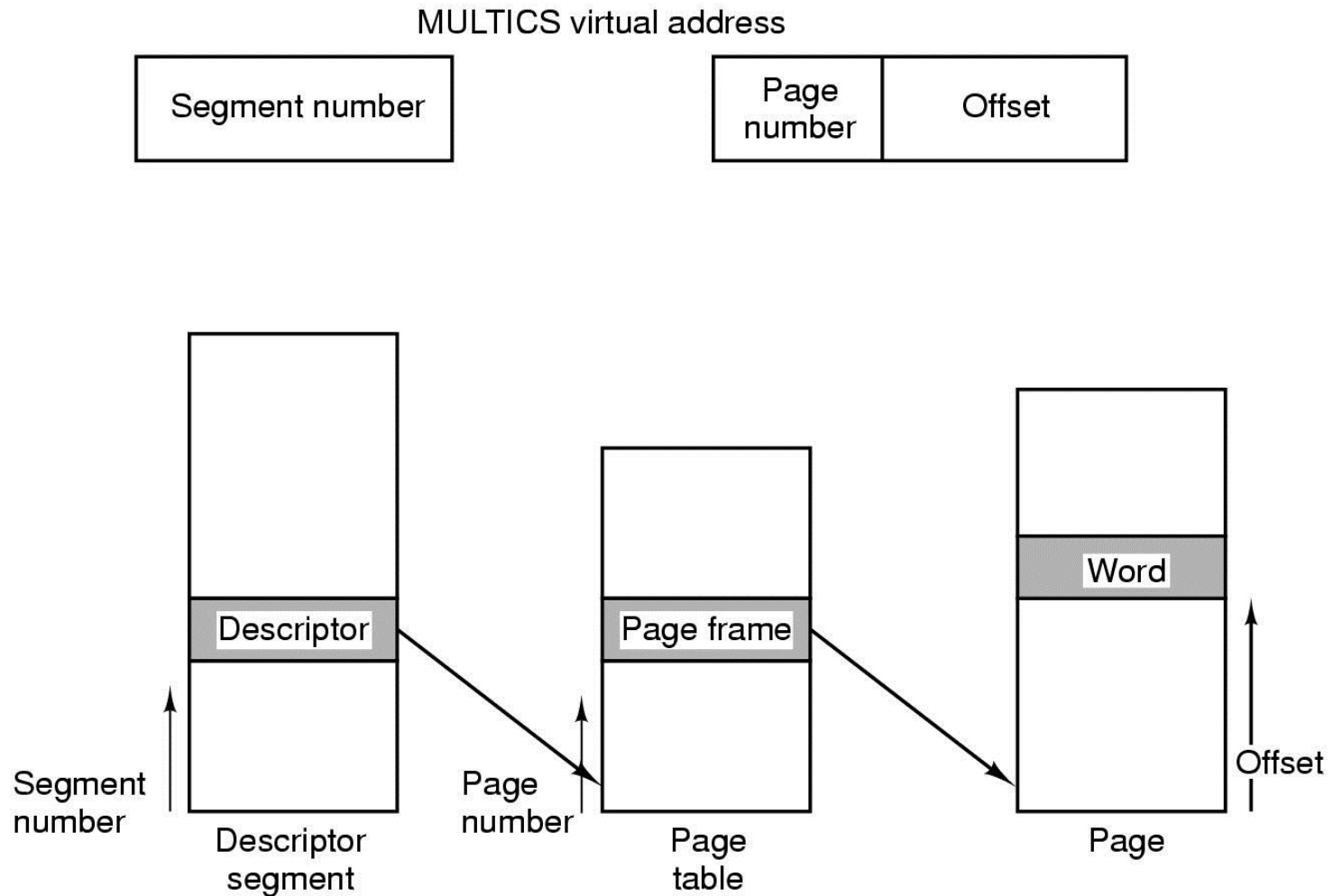
$2^{18} = 256\text{K}$ segments possible



$2^{16} = 64\text{K}$ words addressed within a segment

A 34-bit MULTICS virtual address

Segmentation with Paging: MULTICS



- Conversion of a 2-part MULTICS address into a main memory address

Segmentation with Paging: MULTICS

Comparison field					Is this entry used?
Segment number	Virtual page	Page frame	Protection	Age	↓
4	1	7	Read/write	13	1
6	0	2	Read only	10	1
12	3	1	Read/write	2	1
					0
2	1	0	Execute only	7	1
2	2	12	Execute only	9	1

Age: For LRU replacement:
we'll see later

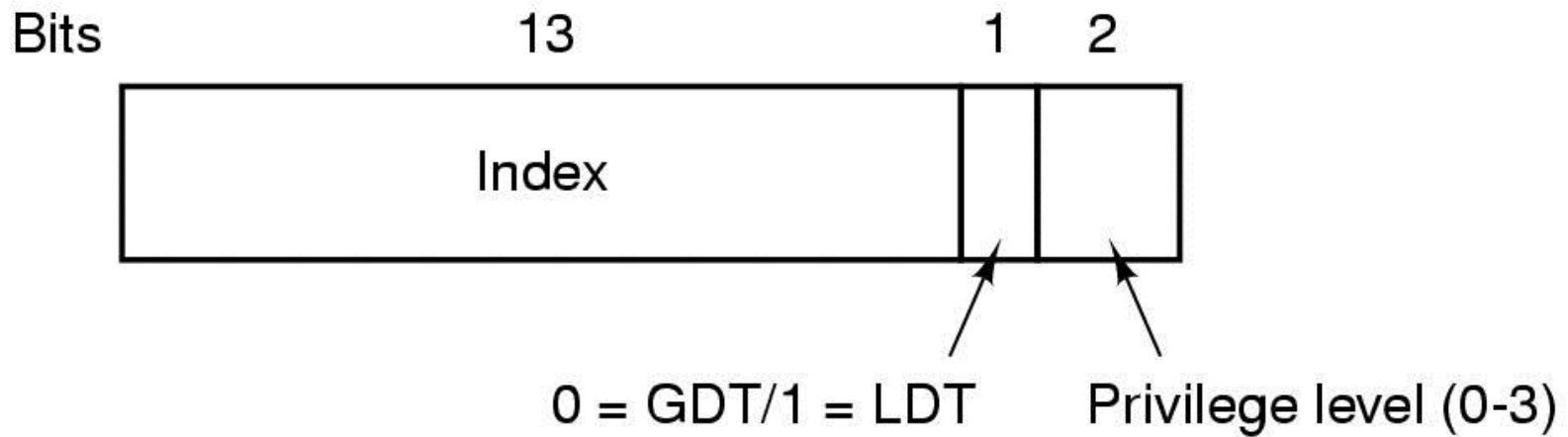
- Simplified version of the MULTICS TLB
- Existence of 2 page sizes makes actual TLB more complicated

Modern systems with segmented paged memory: Pentium

- Two tables:
 - LDT: local descriptor table: owned by processes
 - GDT: Global descriptor table: system segments including the OS itself.
- 16K independent segments each with memory space of 1G 32-bit words.

Segmentation with Paging: Pentium

Segment Selector: loaded into one of 6 segment registers



CS: Code selector (for code segment)

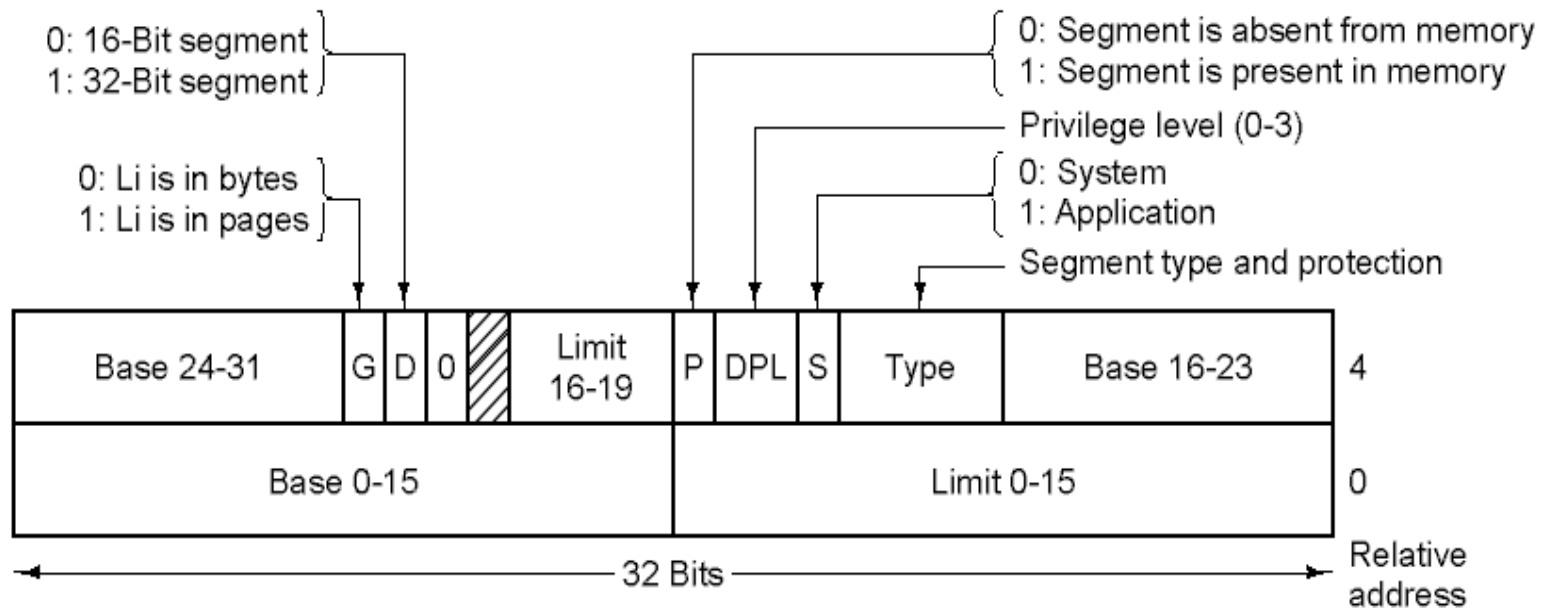
DS: Data selector (for data segment)

13 bits for the segment table entry
(seg table size = $2^{13} = 8K$ entries)

A Pentium selector

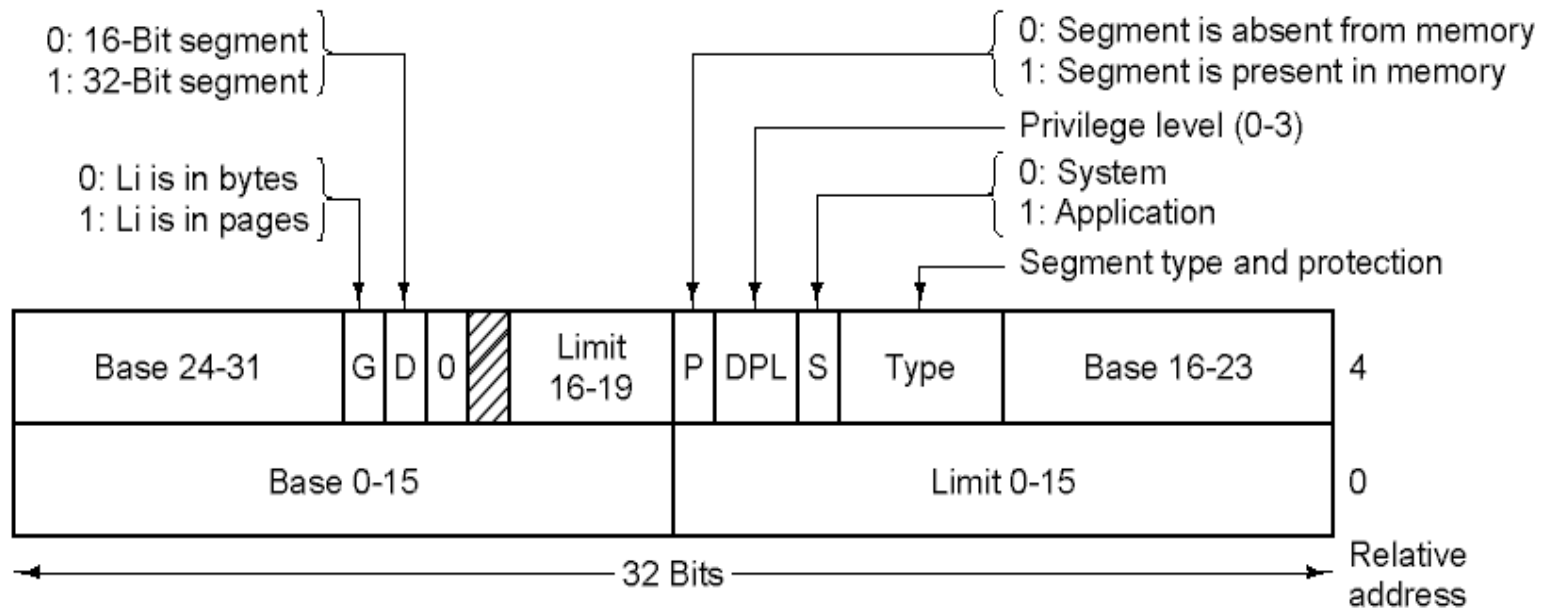
Segmentation with Paging: Pentium

- Pentium code segment descriptor (i.e., segment table entry): 8 bytes = 2 words
- Base (32 bits) and limit (20 bits) fields spread all over the segment descriptor.



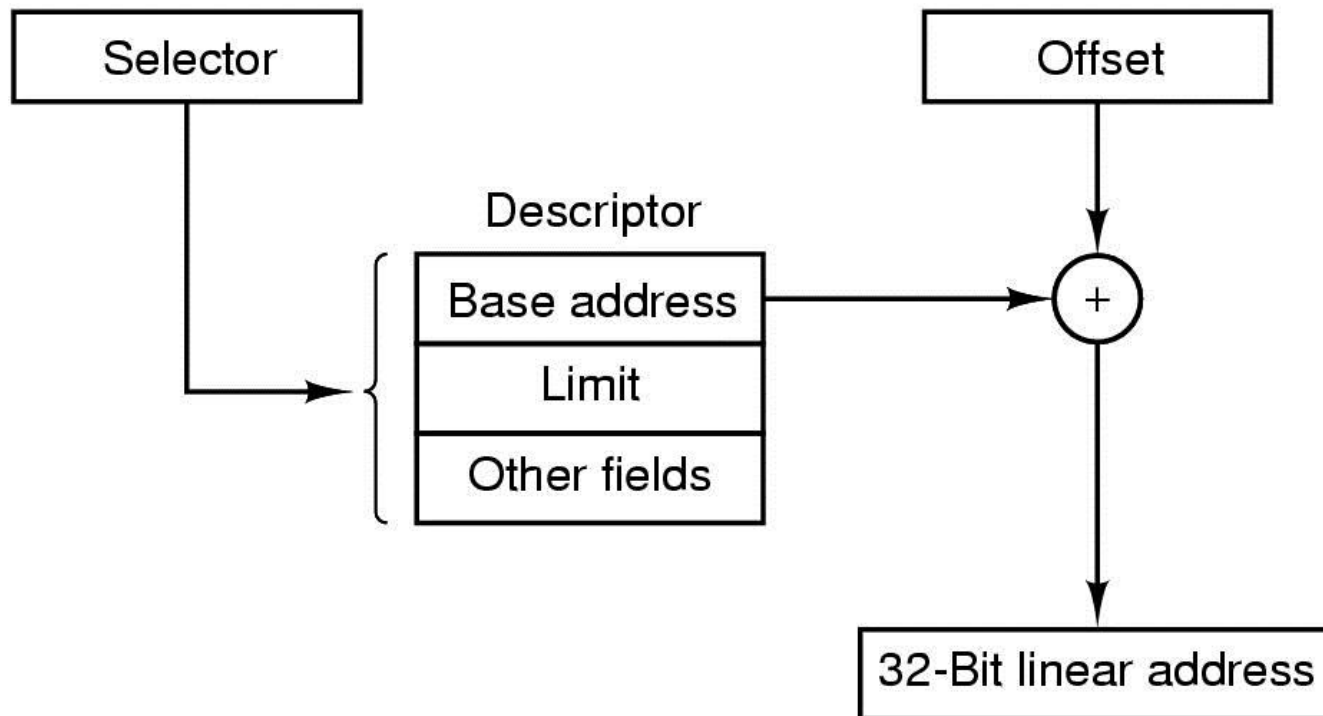
Segmentation with Paging: Pentium

- G bit is granularity bit:
 - G=0: segment size in bytes (max 1MB)
 - G = 1: segment size given in number of pages.
- Pentium page size = 4KB = 2^{12} , therefore, 20 bit segment limit is sufficient for segment sizes up to 2^{32} bytes.



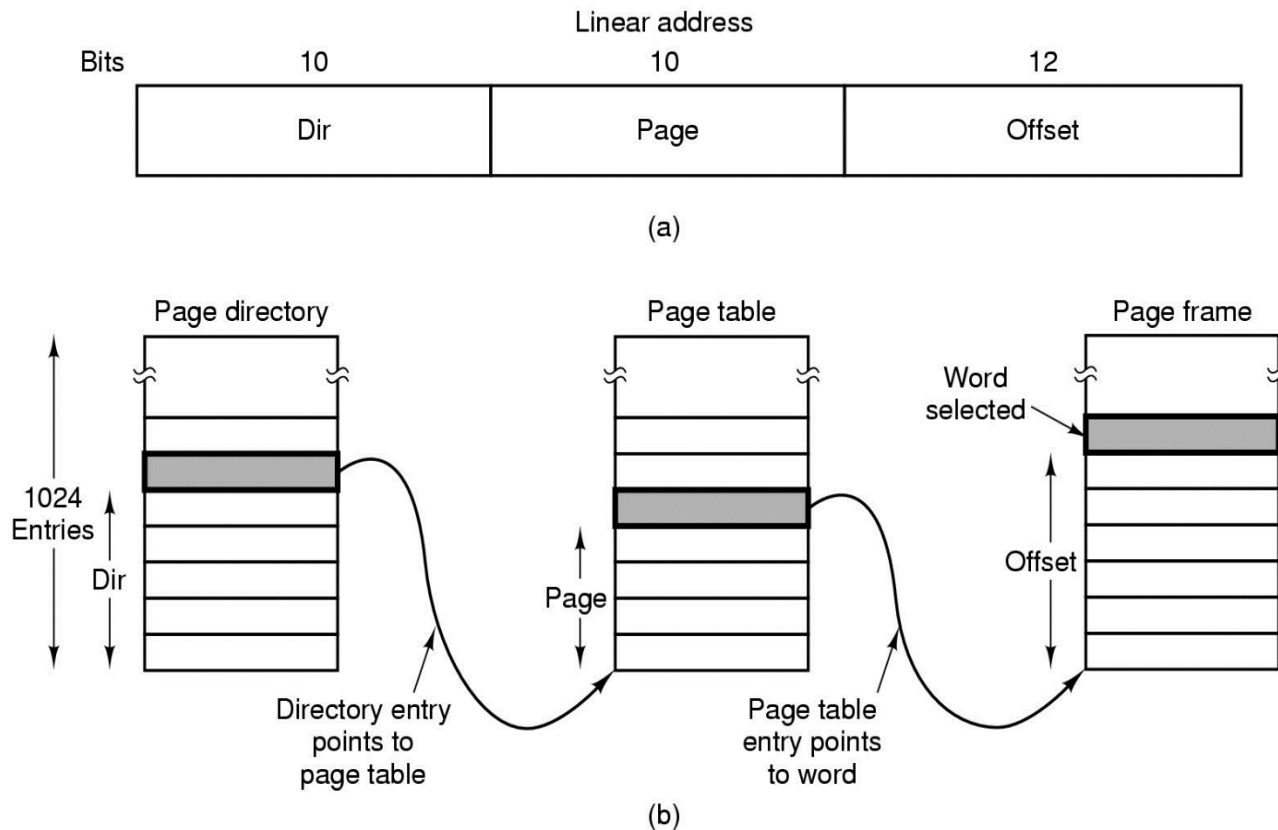
Segmentation with Paging: Pentium

- Conversion of a (selector, offset) pair to a linear address
- If paging is disabled (G bit = 0), then we have pure segmentation: offset is added to base address to obtain physical address.



Segmentation with Paging: Pentium

- If G bit = 1, then we have paged segmented memory: physical address obtained via page table lookup.
- Occurs on TLB miss.



Segmentation with Paging: Pentium

- Setting the base=0 and limit to max for 32 bits, results in a pure paged memory management system for Pentium.
- Most current OS's for Pentium work in this mode.
- OS/2 was the only OS that took advantage of the segmented paged MMU architecture of the Pentium.

Segmentation with Paging: Pentium

■ Protection on the Pentium

