Lecture 3

Memory Management

Memory Management

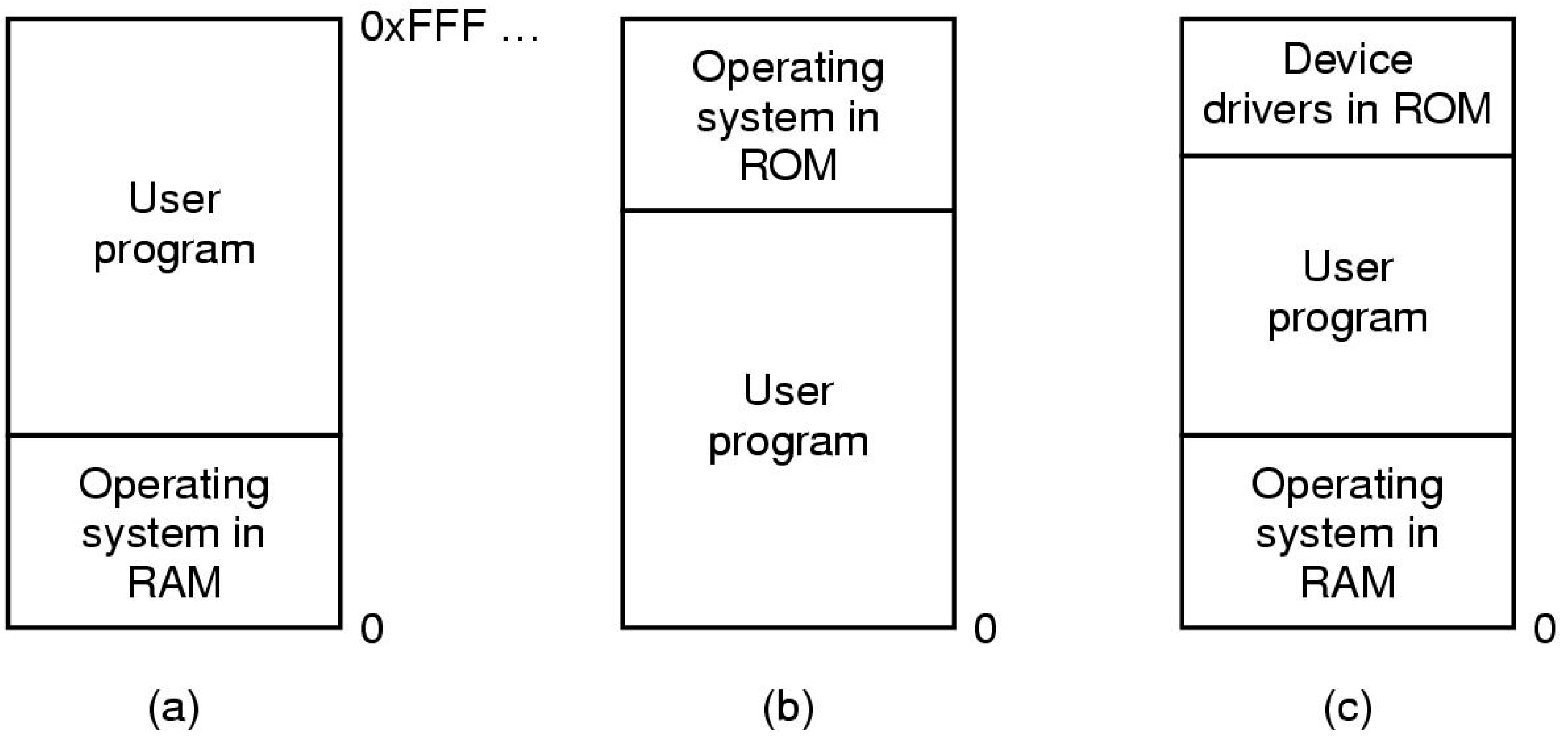
- Ideally programmers want memory that is
  - large
  - fast
  - non volatile

- 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

Basic Memory Management

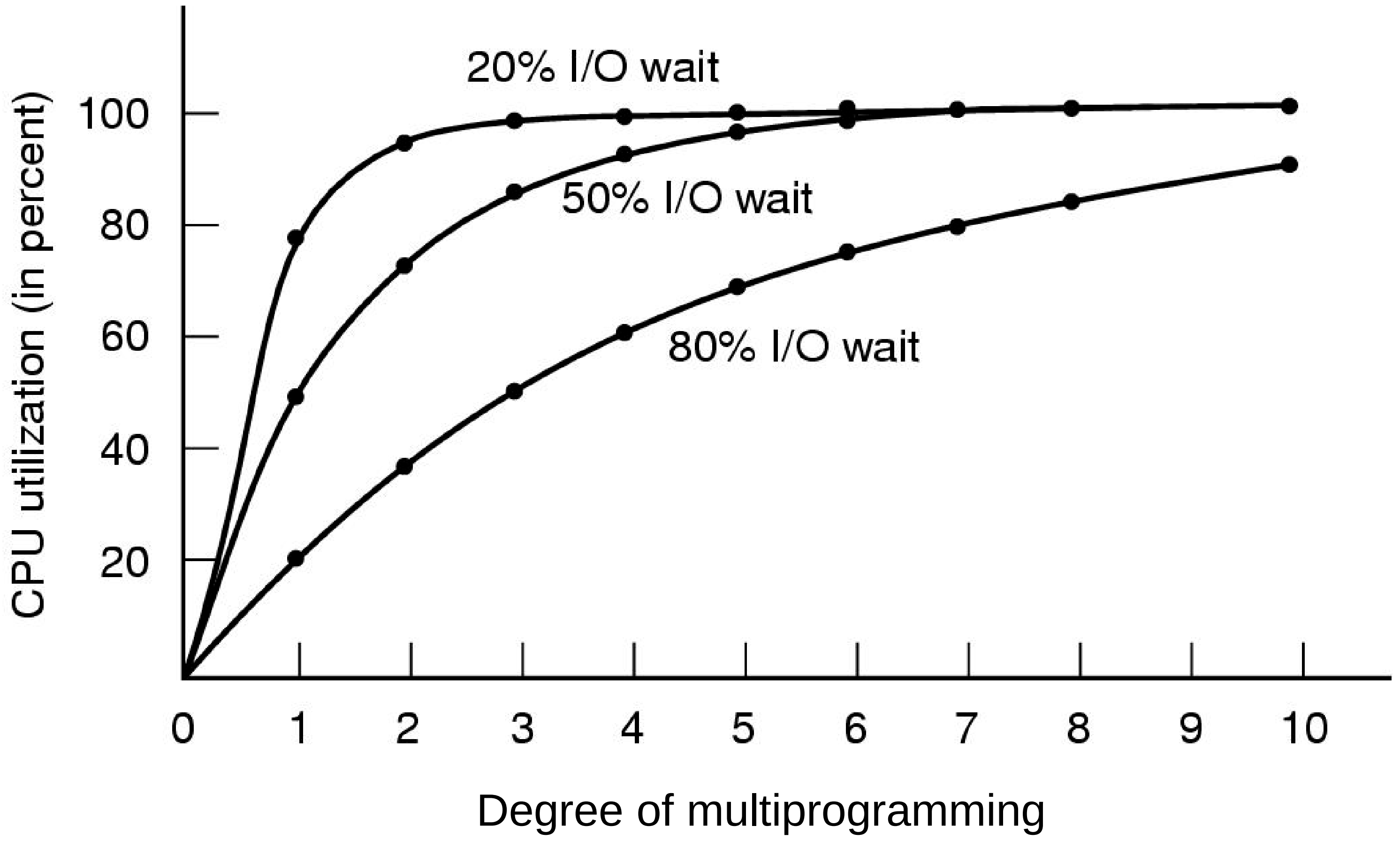
Monoprogramming without Swapping or Paging



Three simple ways of organizing memory

- an operating system with one user process

Modeling Multiprogramming



CPU utilization as a function of number of processes in memory

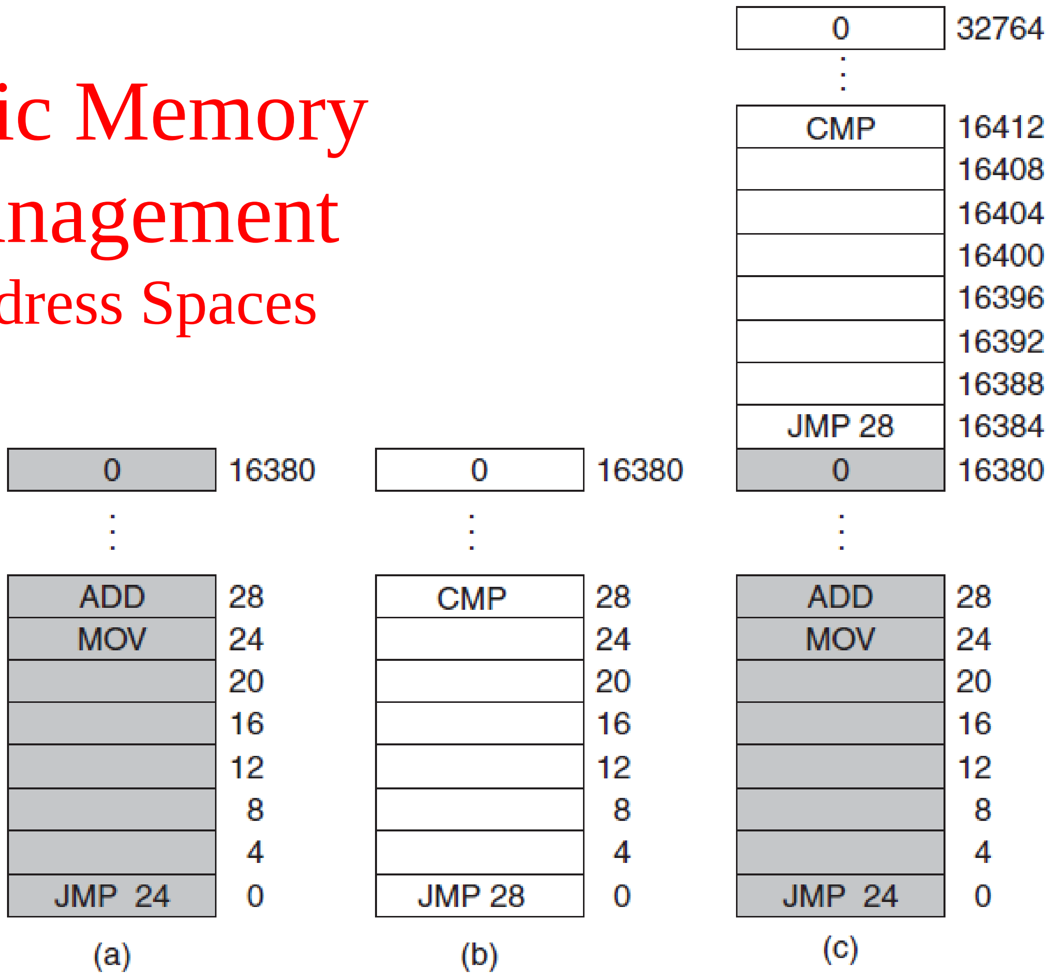
Relocation and Protection

- Cannot be sure where program will be loaded in memory
  - address locations of variables, code routines cannot be absolute
  - must keep a program out of other processes’ partitions

- Use base and limit values
  - address locations added to base value to map to physical addr
  - address locations larger than limit value is an error

Basic Memory Management

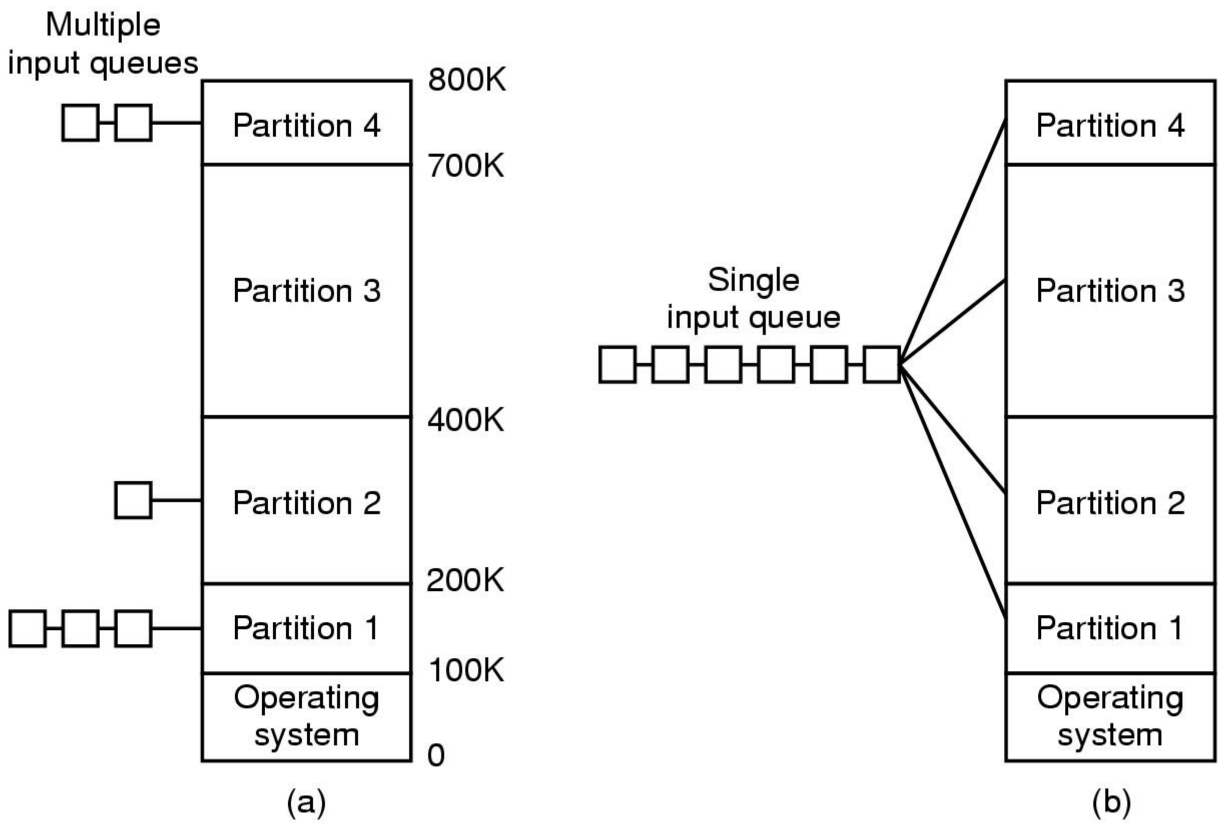
Address Spaces



Relocation problem.

(a) A 16-KB program.(b) Another 16-KB program. (c) The two programs loaded consecutively into memory.

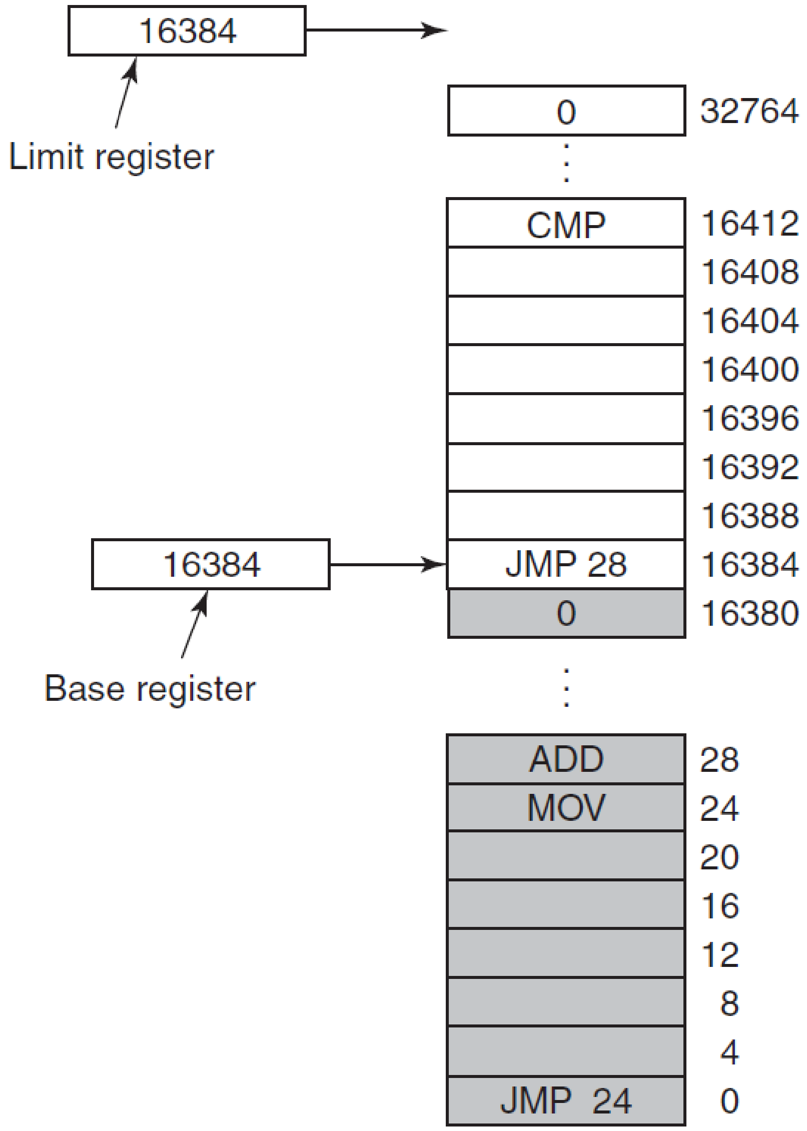
Multiprogramming with Fixed Partitions



- Fixed memory partitions
  - separate input queues for each partition
  - single input queue

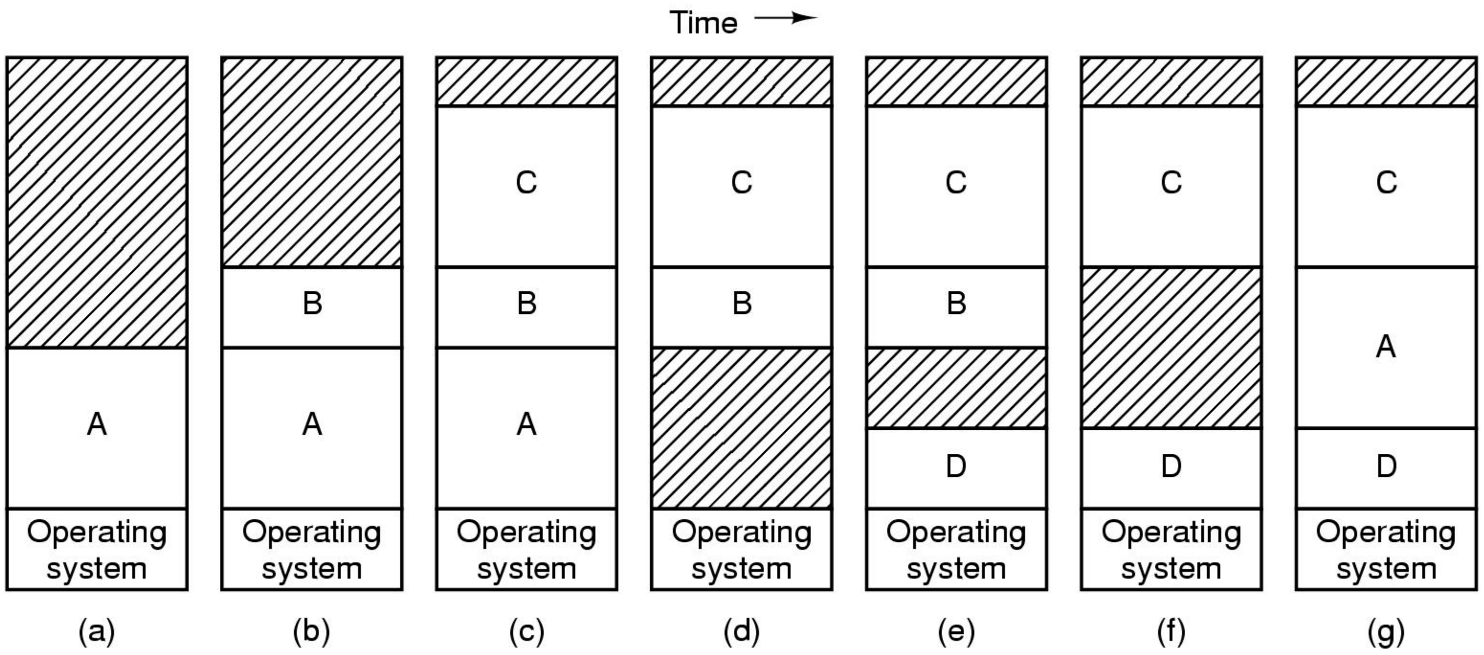
Basic Memory Management

Address Spaces



Base and limit registers can be used to give each process a separate address space.

Swapping (1)

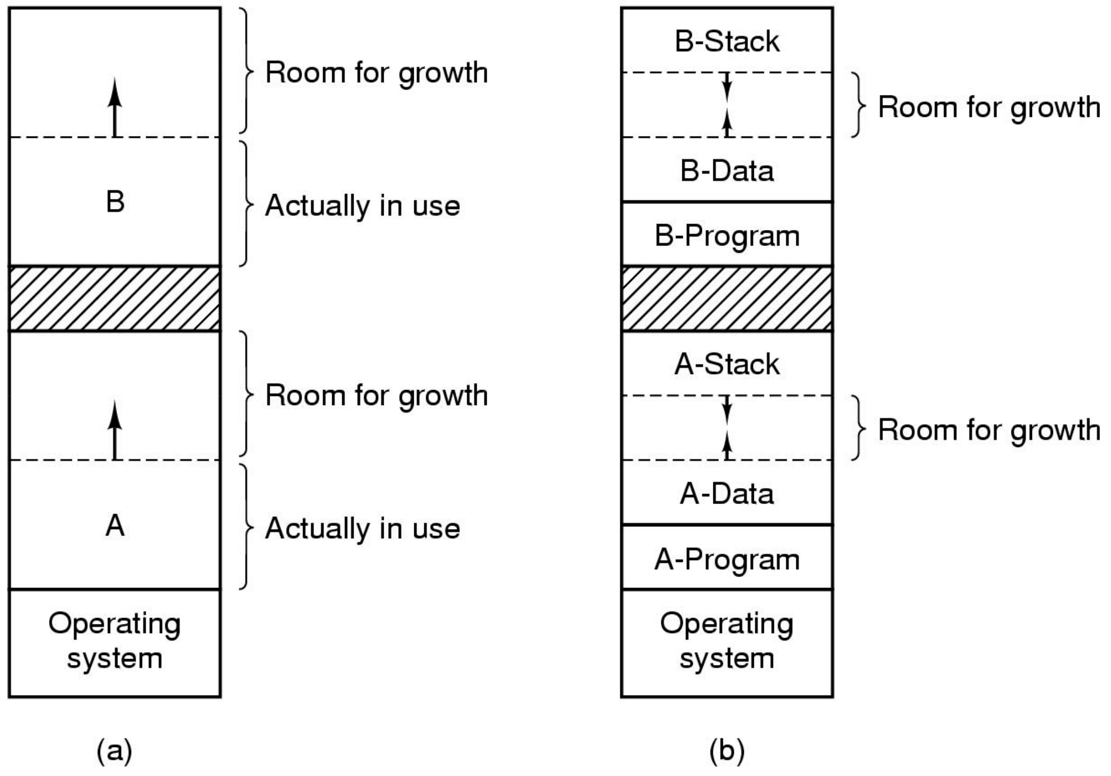


Memory allocation changes as

- processes come into memory
- leave memory

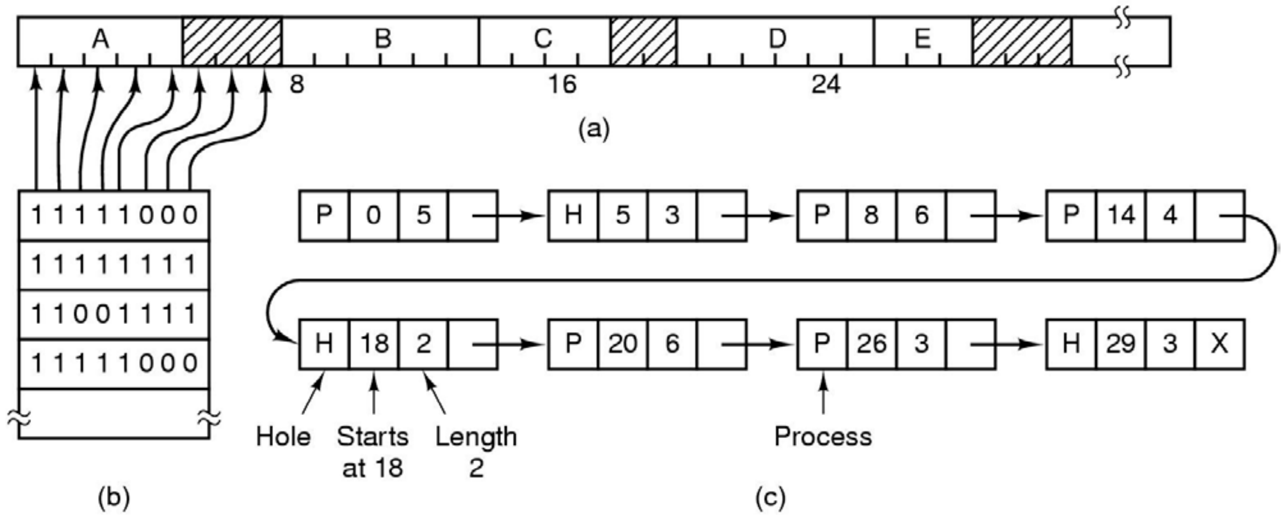
Shaded regions are unused memory

Swapping (2)



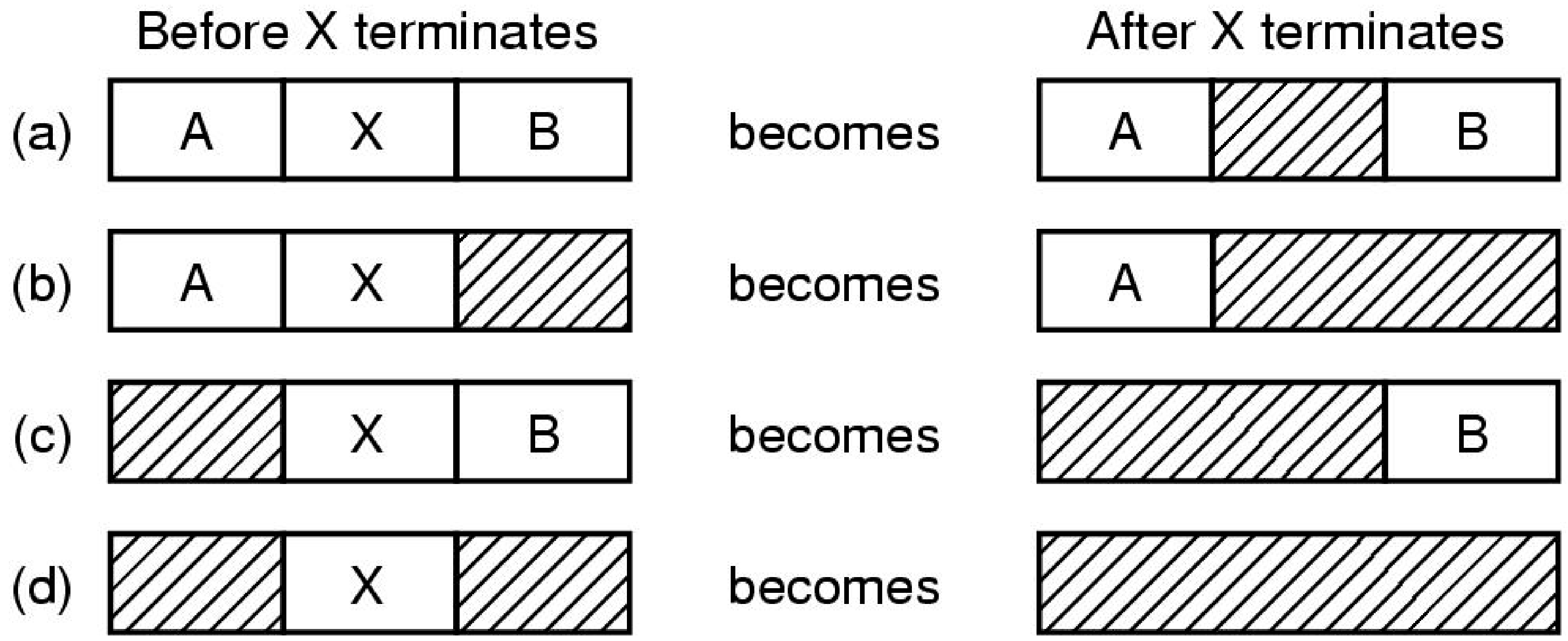
- Allocating space for growing data segment
- Allocating space for growing stack & data segment

Memory Management with Bit Maps



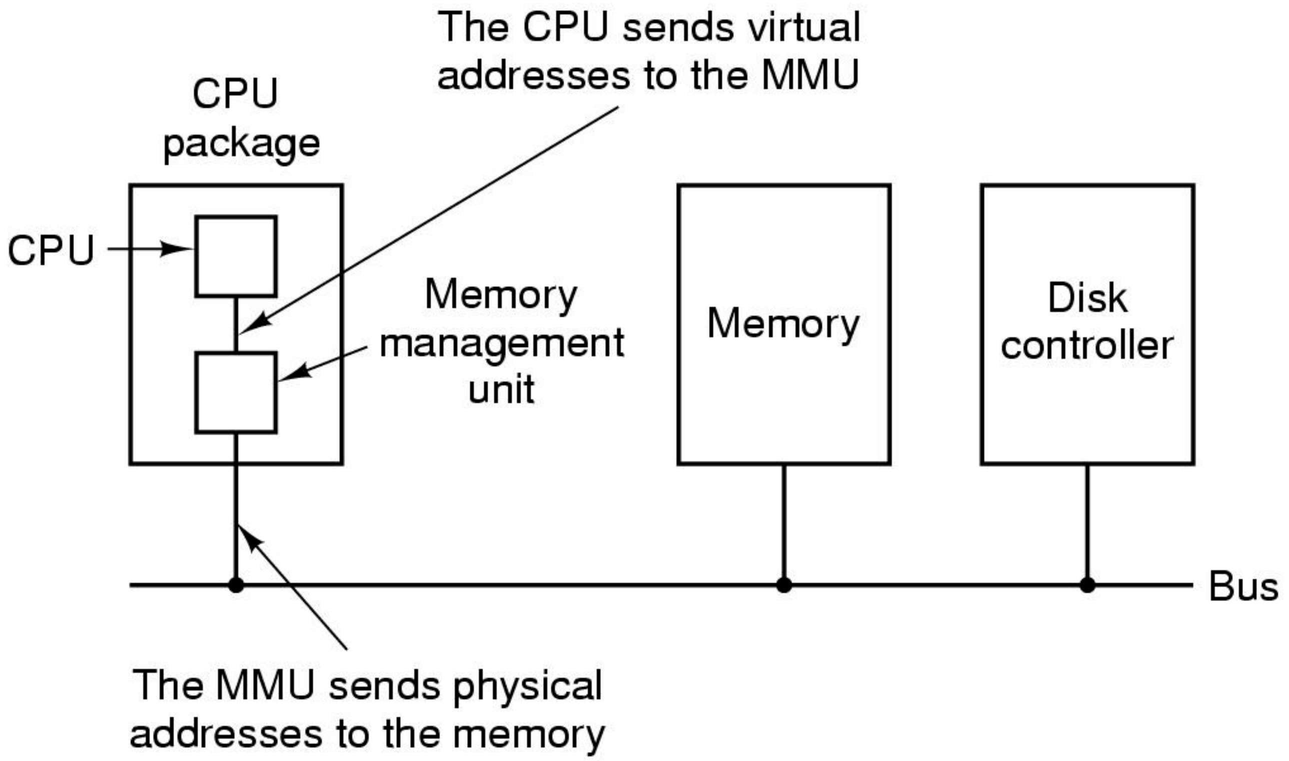
- Part of memory with 5 processes, 3 holes
  - tick marks show allocation units
  - shaded regions are free
- Corresponding bit map
- Same information as a list

Memory Management with Linked Lists



Four neighbor combinations for the terminating process X

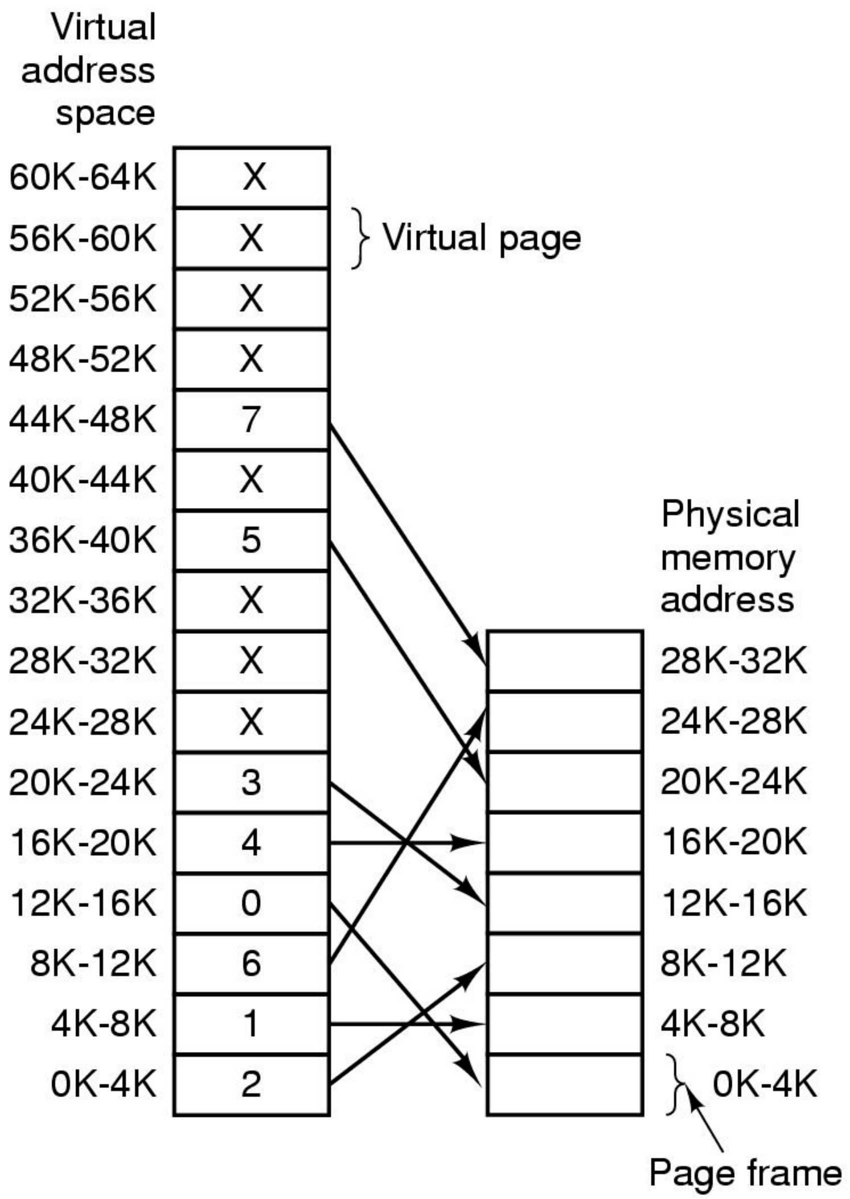
Virtual Memory
Paging (1)


The CPU sends virtual addresses to the MMU
CPU package
CPU
Memory management unit
Memory
Disk controller
Bus
The MMU sends physical addresses to the memory

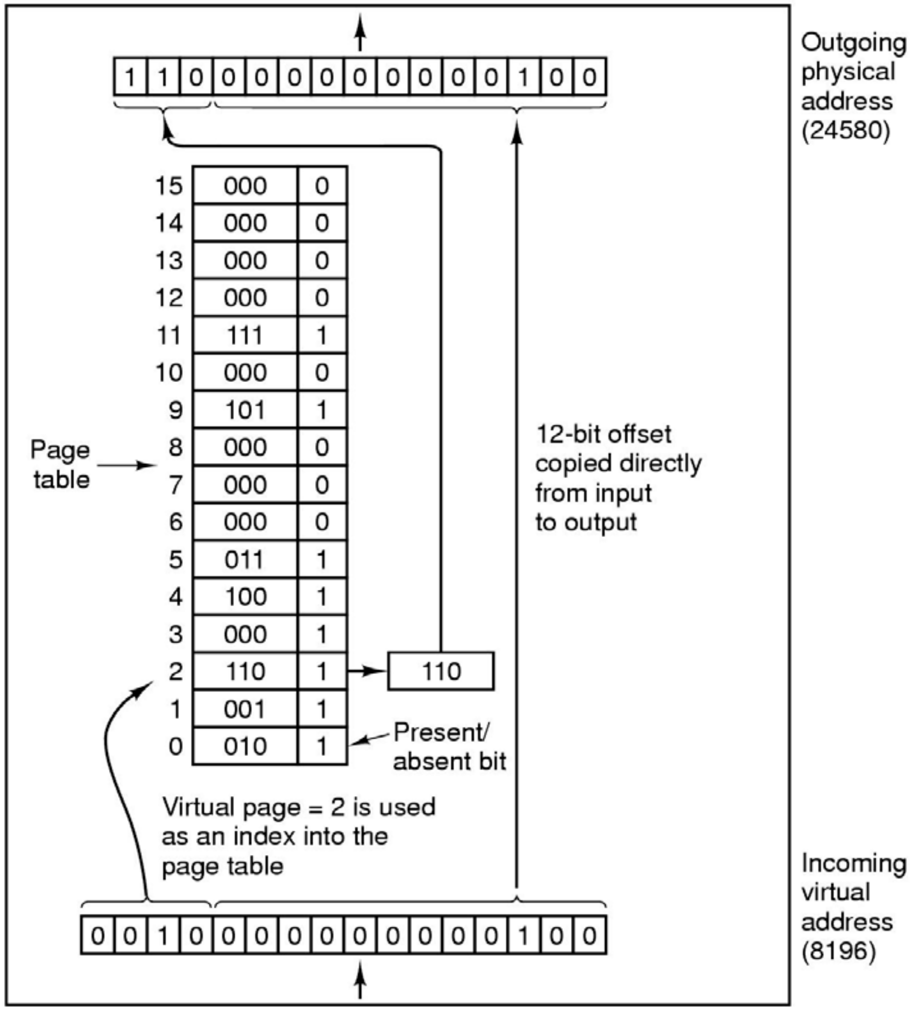
The position and function of the MMU

Paging (2)

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

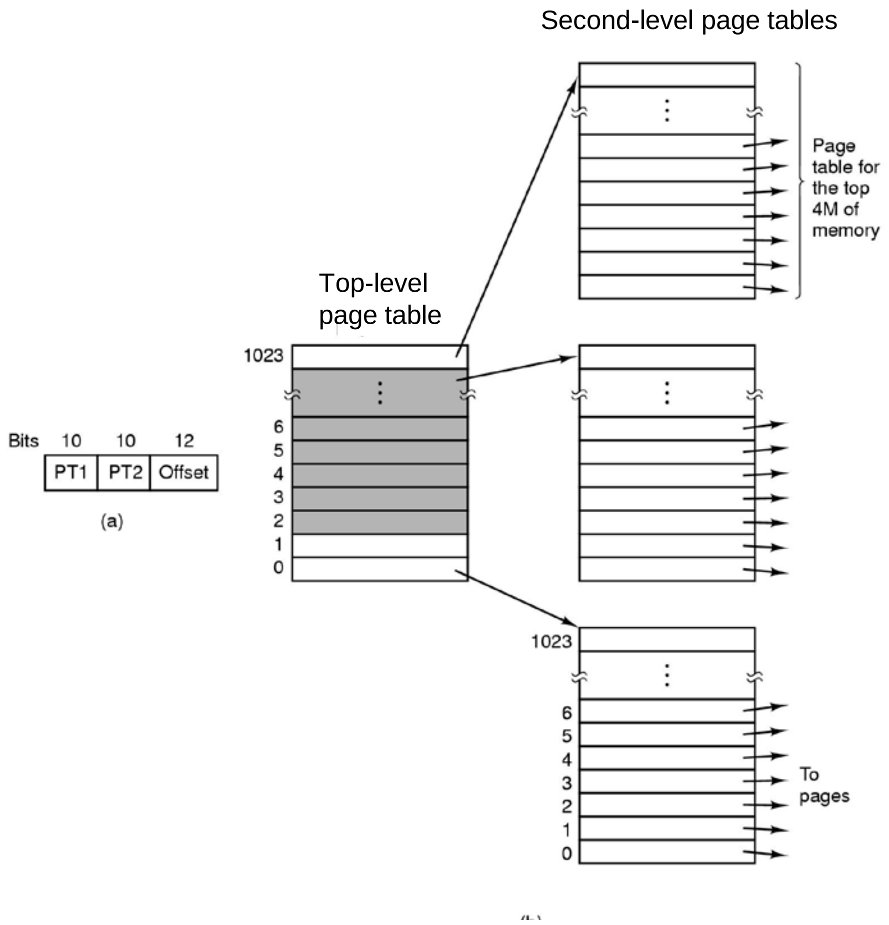


Page Tables (1)



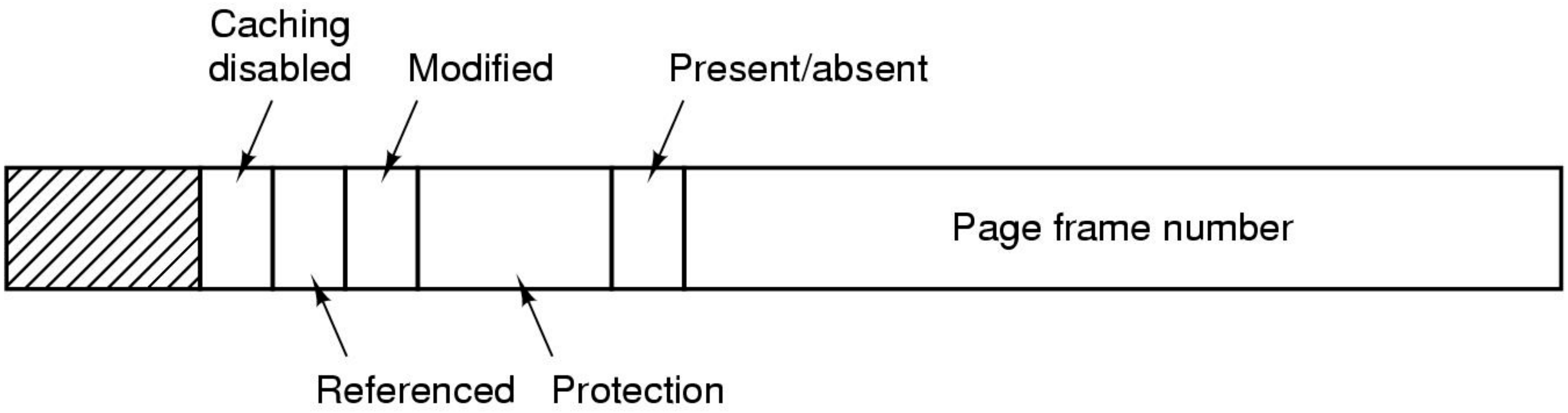
Internal operation of MMU with 16 4 KB pages

Page Tables (2)



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

Page Tables (3)



Typical page table entry

TLBs – Translation Lookaside Buffers

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

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

- 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

- 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

Least Recently Used (LRU)

- Assume pages used recently will 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 <u>every memory reference</u> !!
- Alternatively keep counter in each page table entry
  - choose page with lowest value counter
  - periodically zero the counter

Page Size (1)

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 (2)

- Overhead due to page table and internal fragmentation

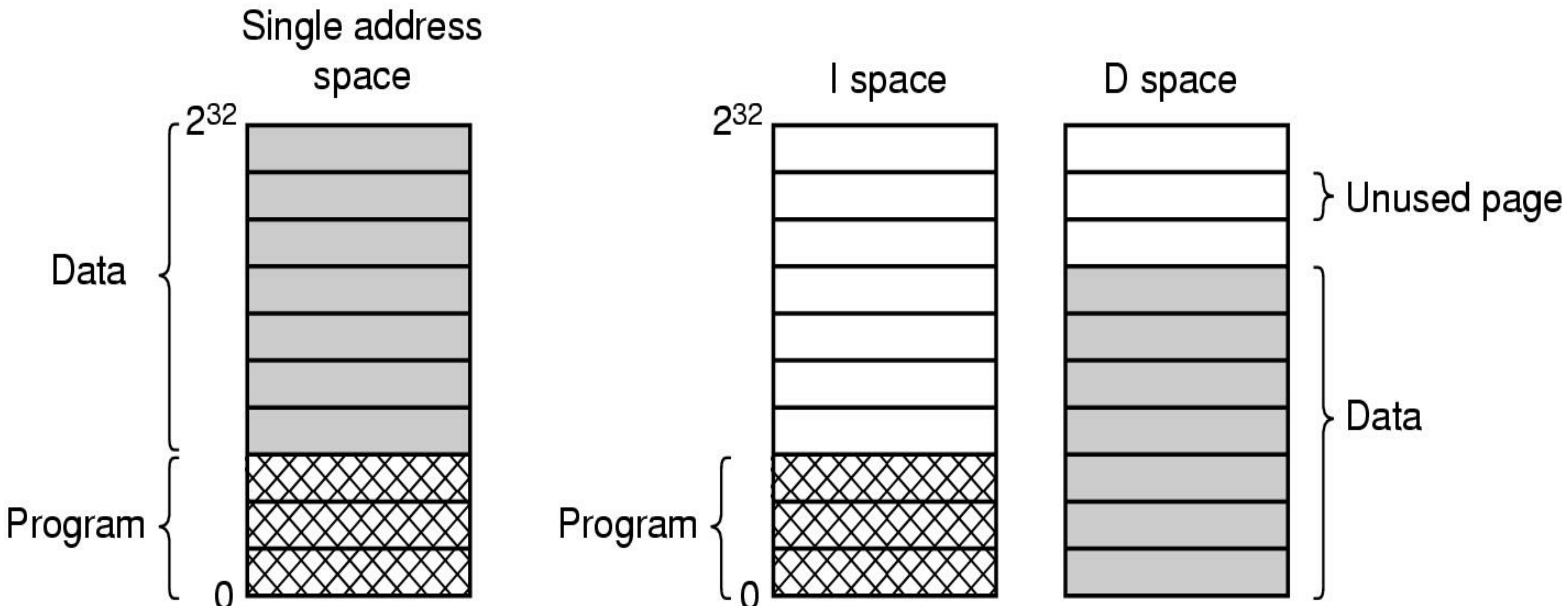
$$overhead = \overbrace{\frac{s \cdot e}{p}}^{\text{page table space}} + \underbrace{\frac{p}{2}}_{\text{internal fragmentation}}$$

- Where
  - s = average process size in bytes
  - p = page size in bytes
  - e = page entry

Optimized when

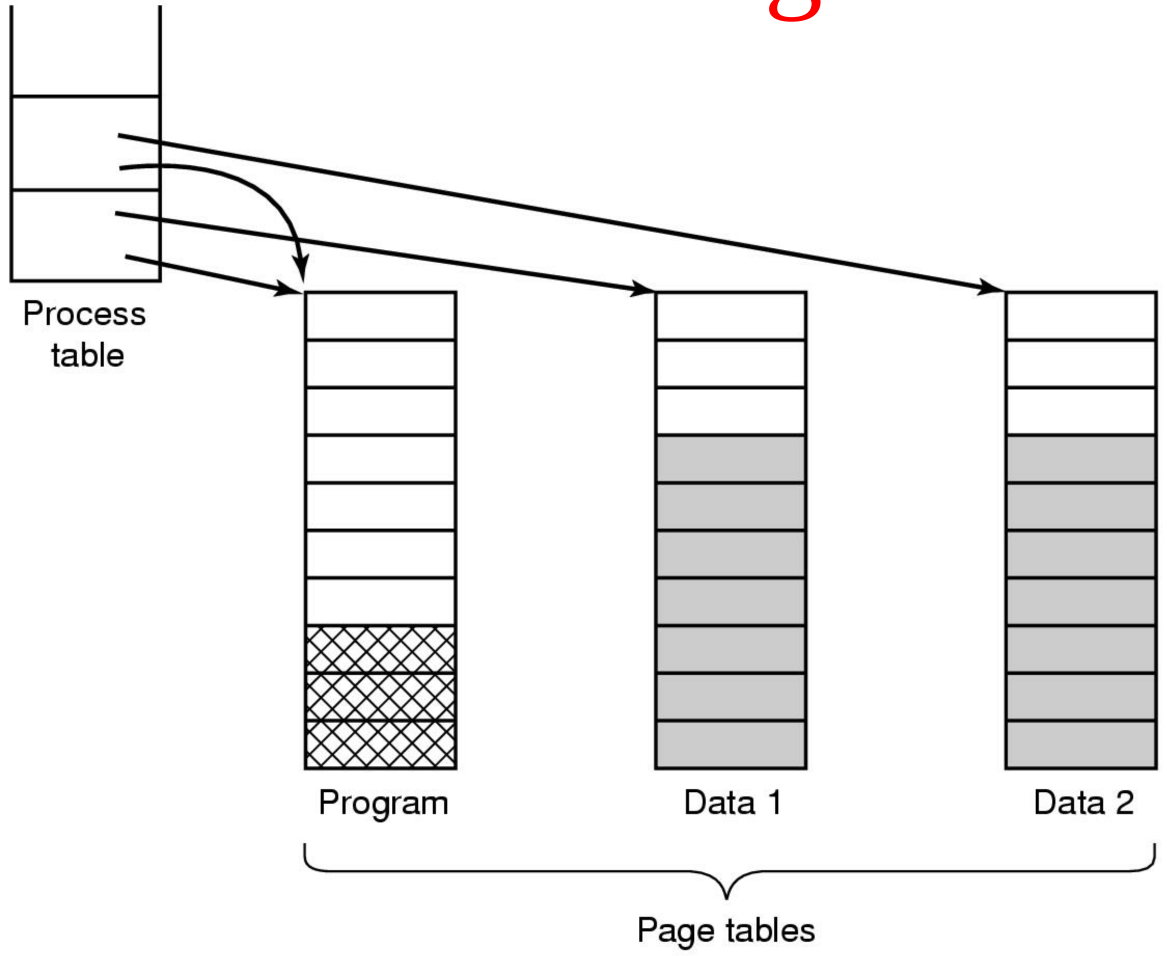
$$p = \sqrt{2se}$$

Separate Instruction and Data Spaces



- One address space
- Separate I and D spaces

Shared Pages



Two processes sharing same program sharing its page table

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 algorithmbut with diff ptr

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

Page Fault Handling (1)

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

Page Fault Handling (2)

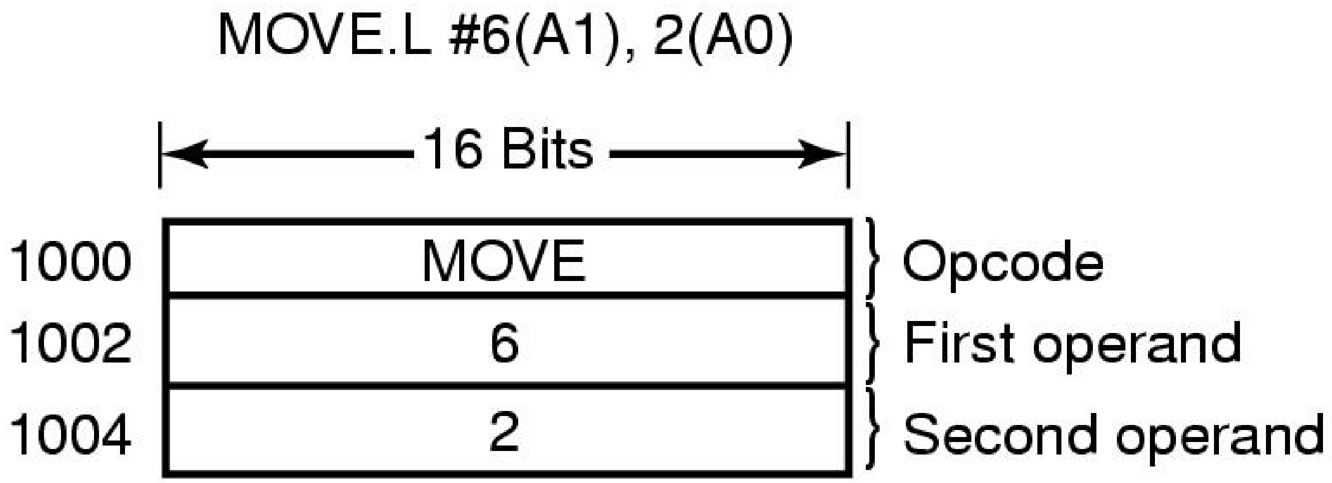
6. OS brings schedules new page in from disk
7. Page tables updated

- Faulting instruction backed up to when it began

6. Faulting process scheduled
7. Registers restored

- Program continues

Instruction Backup

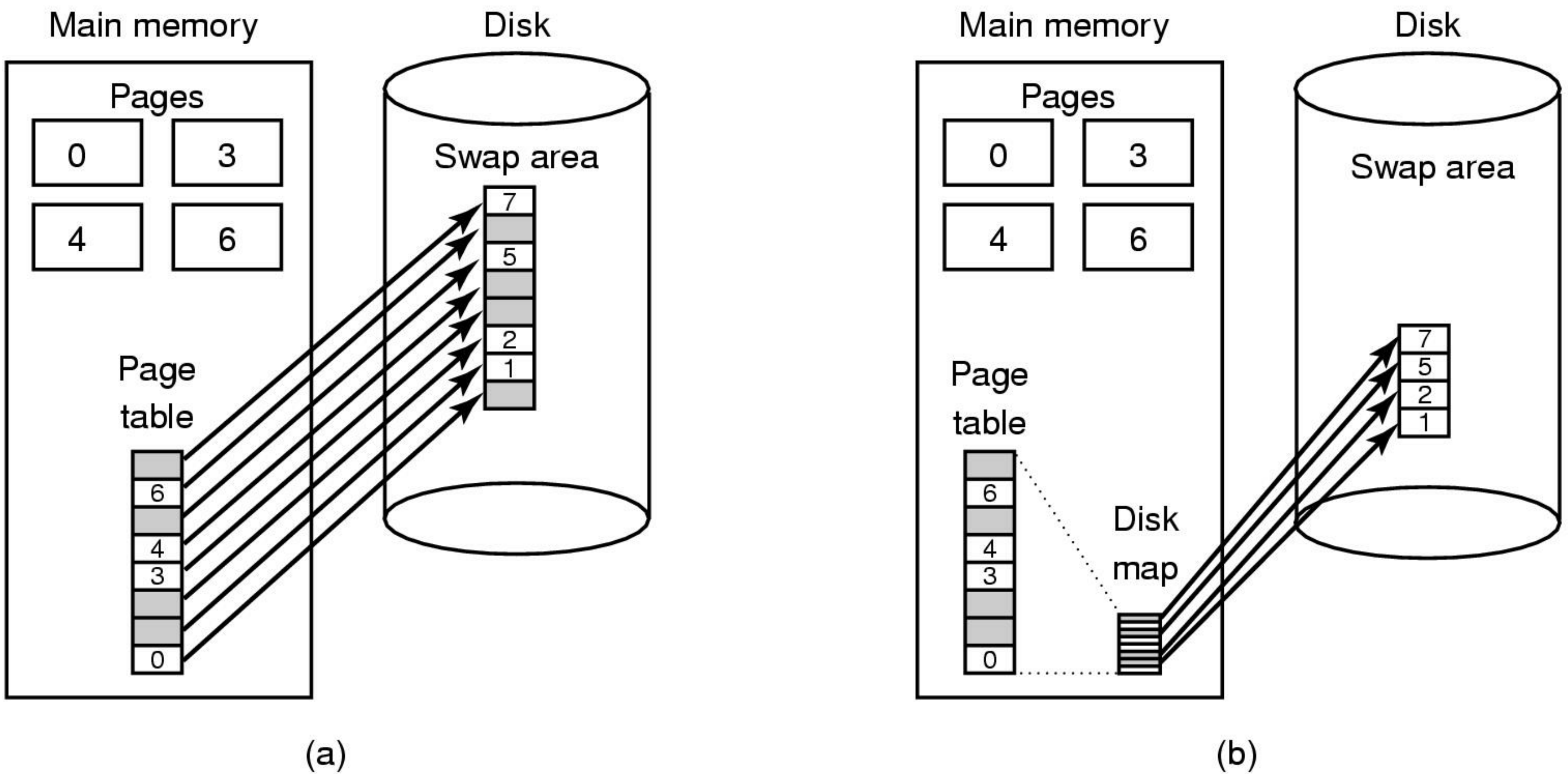


An instruction causing a page fault

Locking Pages in Memory

- Virtual memory and I/O occasionally interact
- Proc issues call for read from device into buffer
  - while waiting for I/O, another processes starts up
  - has a page fault
  - buffer for the first proc may be chosen to be paged out
- Need to specify some pages locked
  - exempted from being target pages

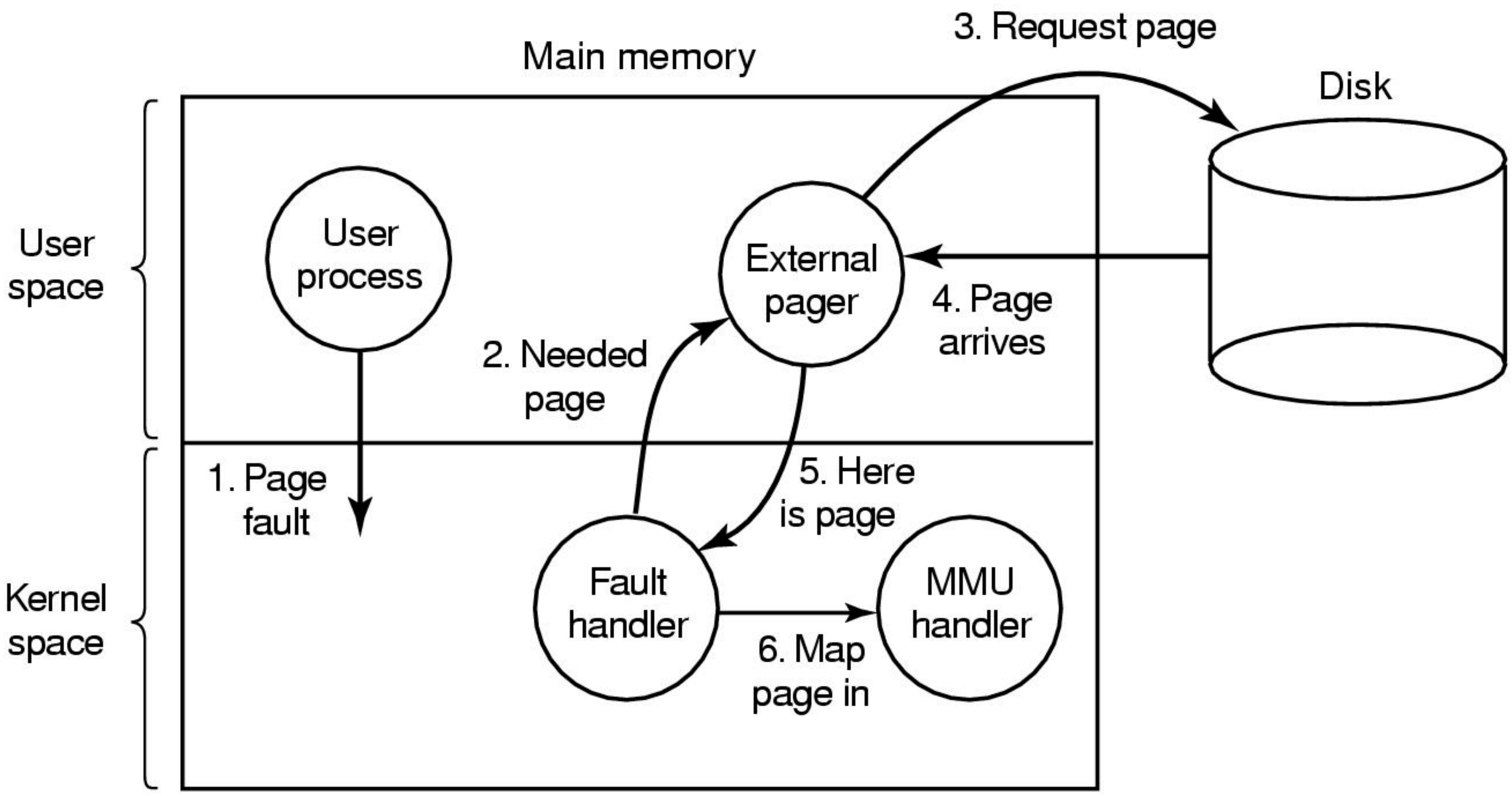
Backing Store



(a) Paging to static swap area

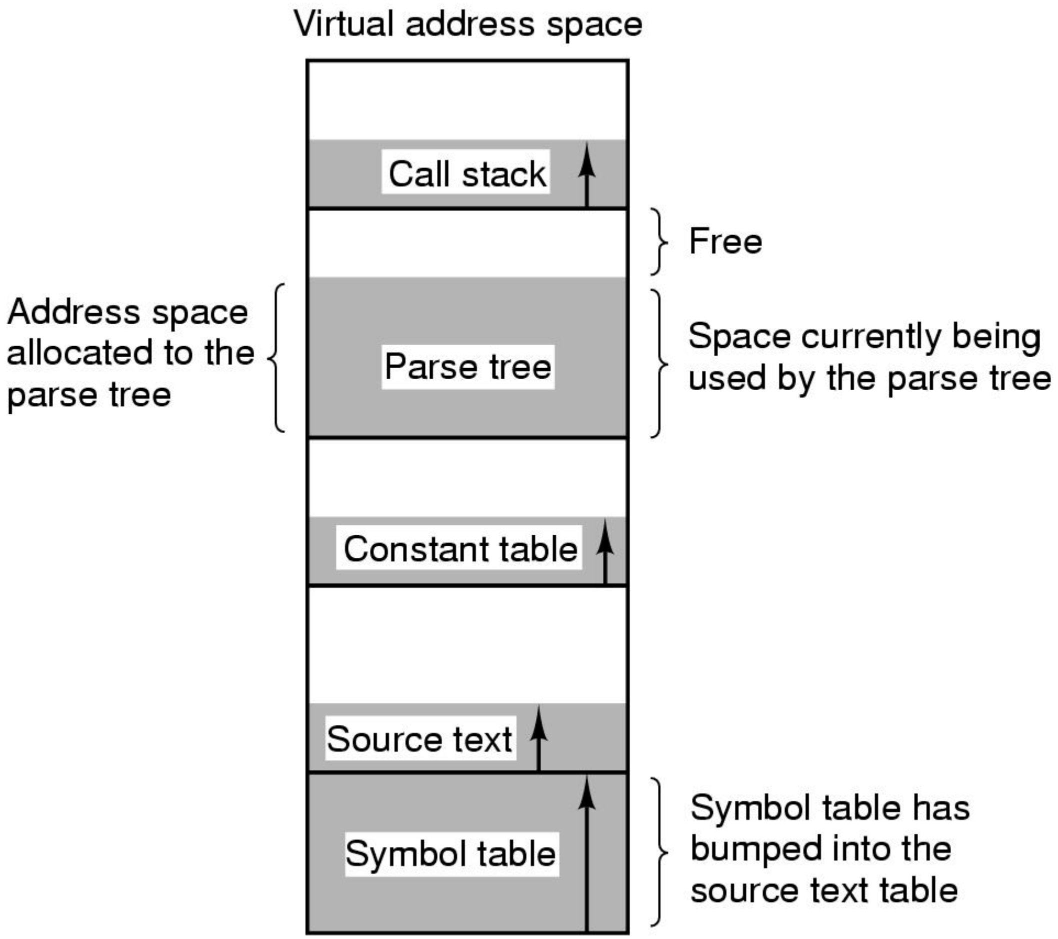
(b) Backing up pages dynamically

Separation of Policy and Mechanism



Page fault handling with an external pager

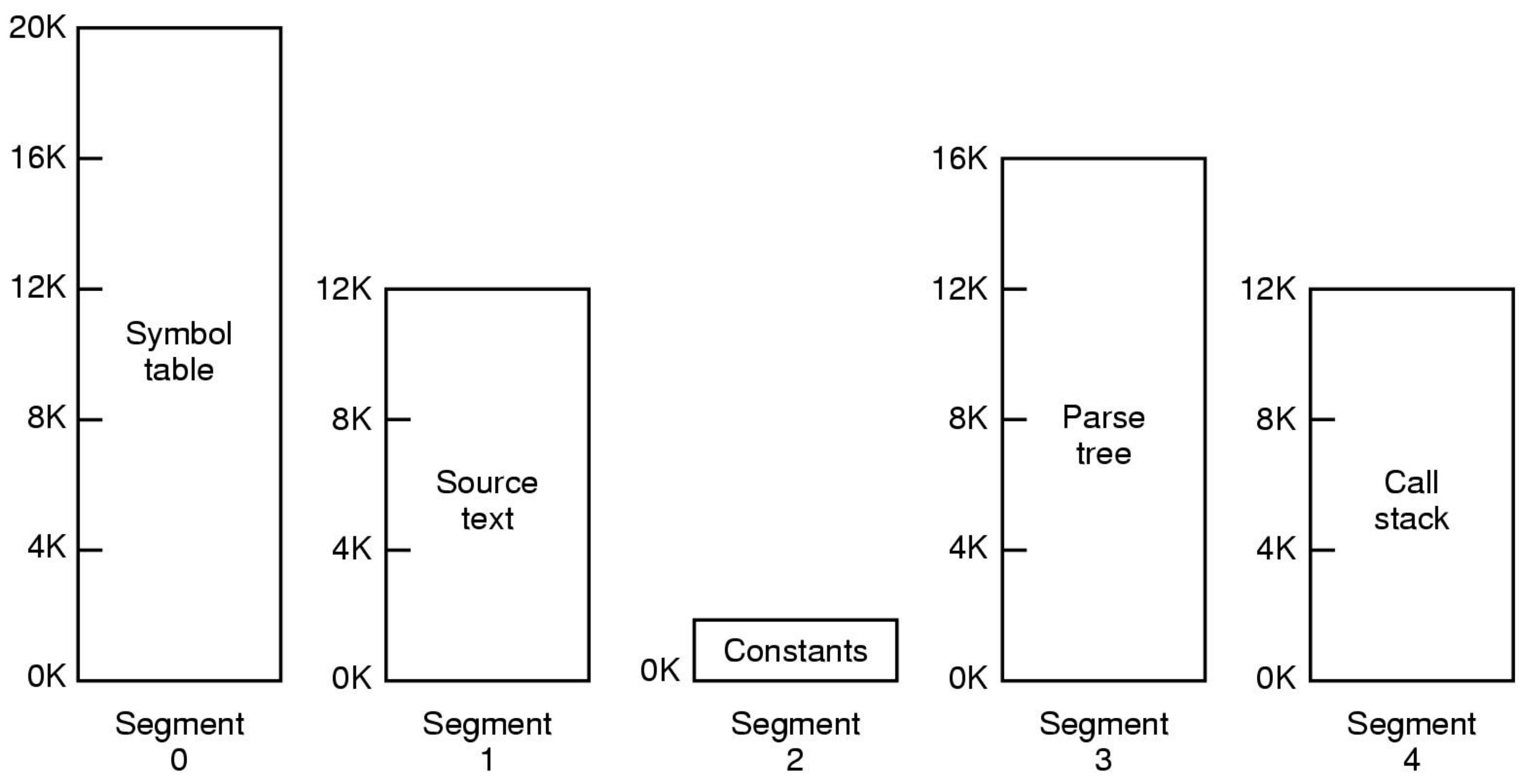
Segmentation (1)



Virtual address space
Call stack
Free
Address space allocated to the parse tree
Parse tree
Space currently being used by the parse tree
Constant table
Source text
Symbol table
Symbol table has bumped into the source text table

- One-dimensional address space with growing tables
- One table may bump into another

Segmentation (2)



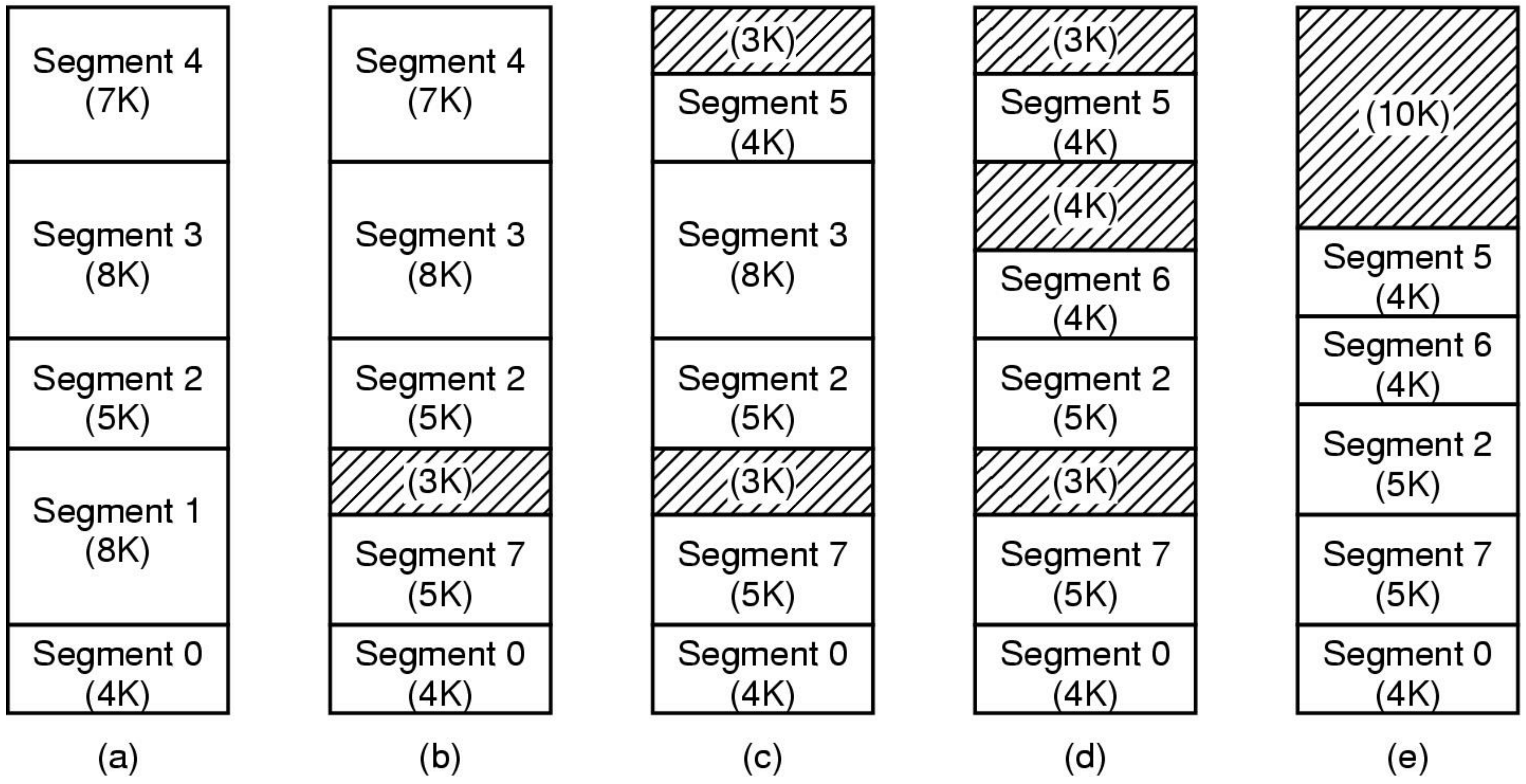
Allows each table to grow or shrink, independently

Segmentation (3)

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 fluctuates 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

Comparison of paging and segmentation

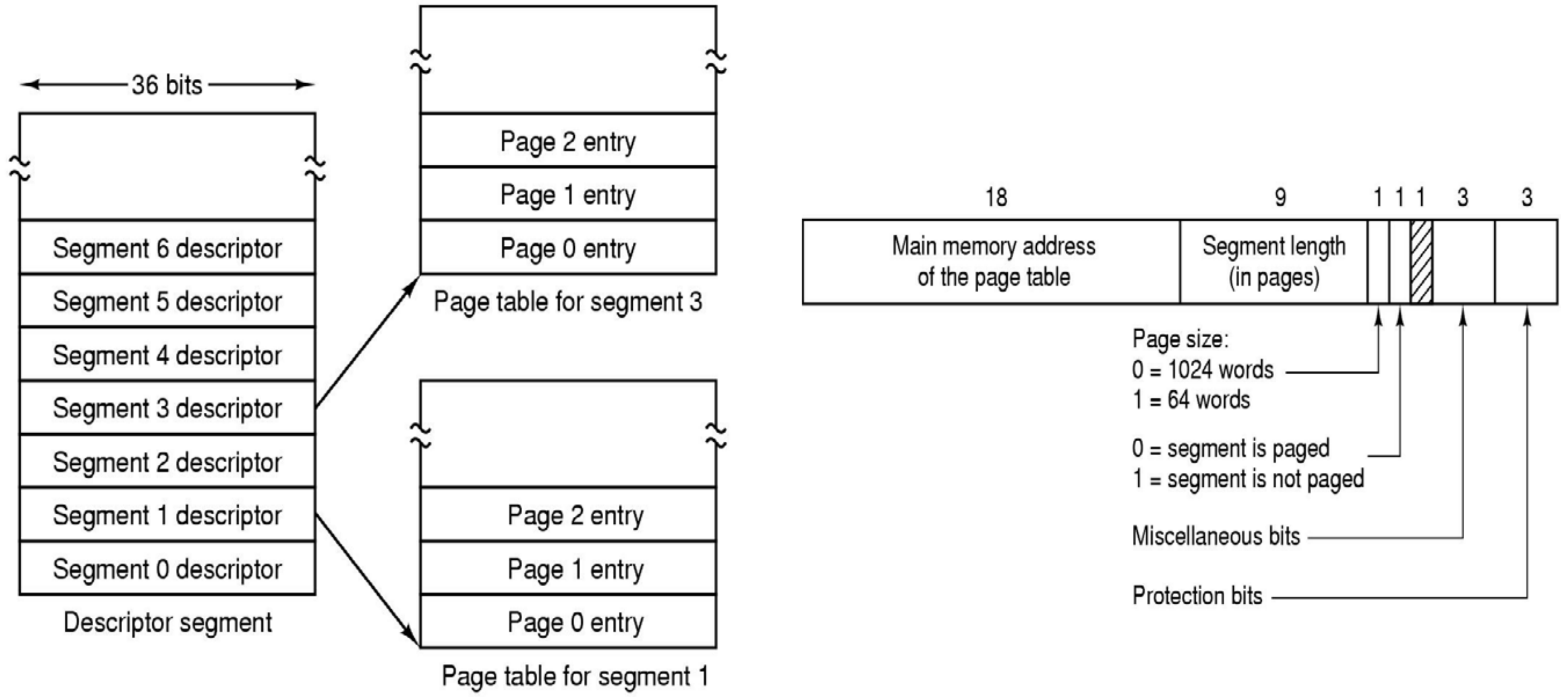
Implementation of Pure Segmentation



(a)-(d) Development of checkerboarding

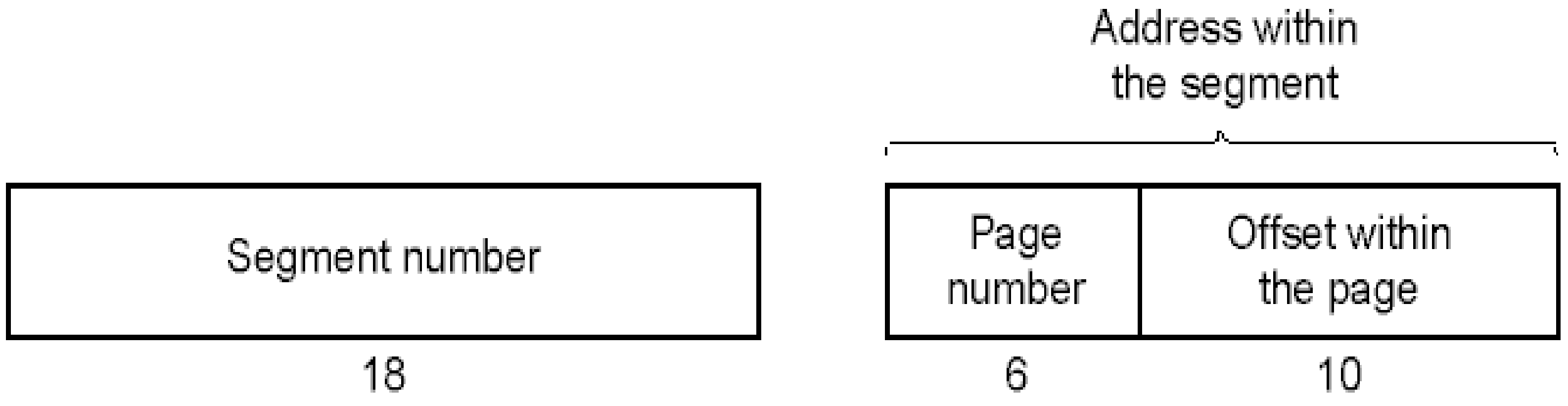
(e) Removal of the checkerboarding by compaction

Segmentation with Paging: MULTICS (1)



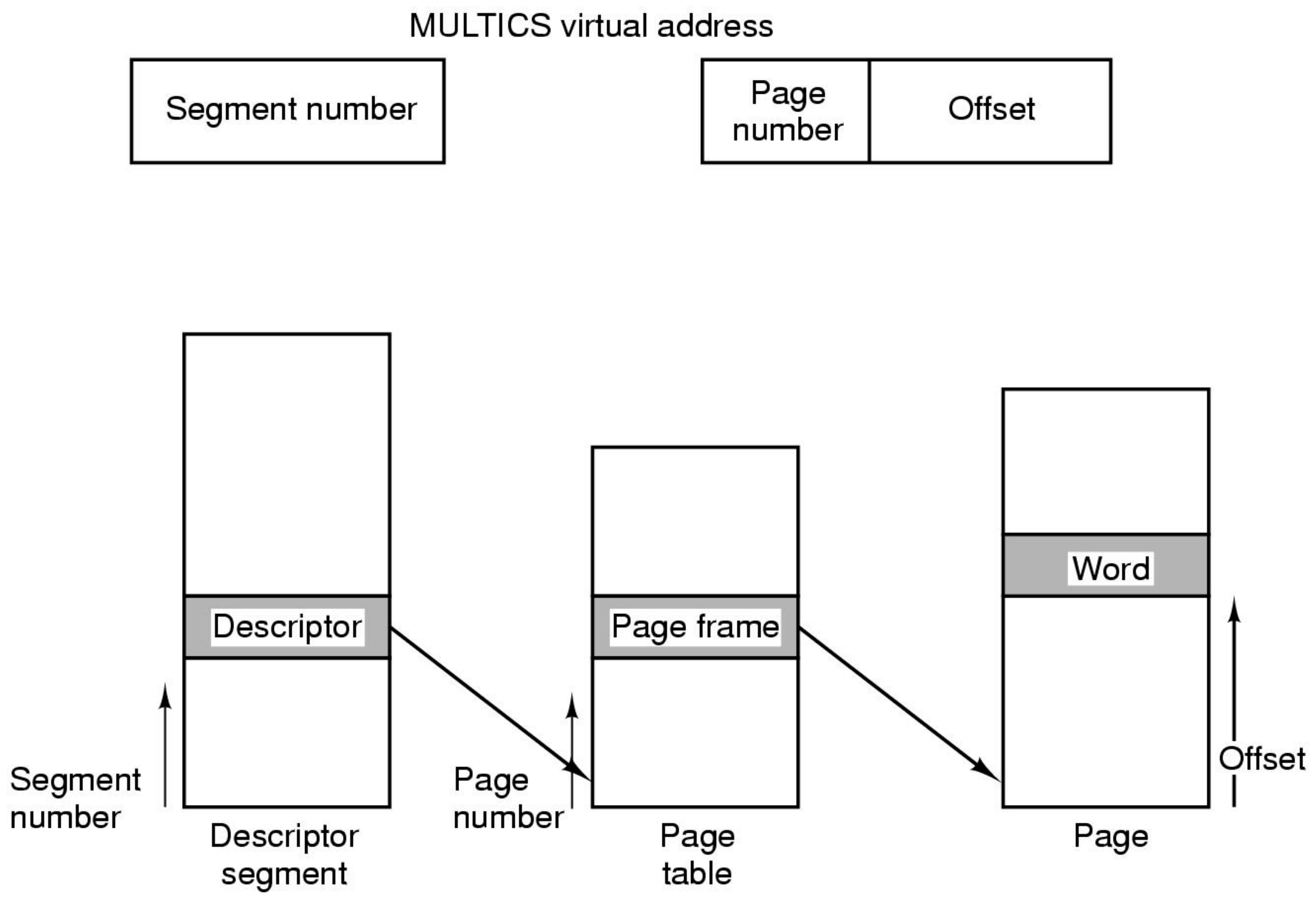
- Descriptor segment points to page tables
- Segment descriptor – numbers are field lengths

Segmentation with Paging: MULTICS (2)



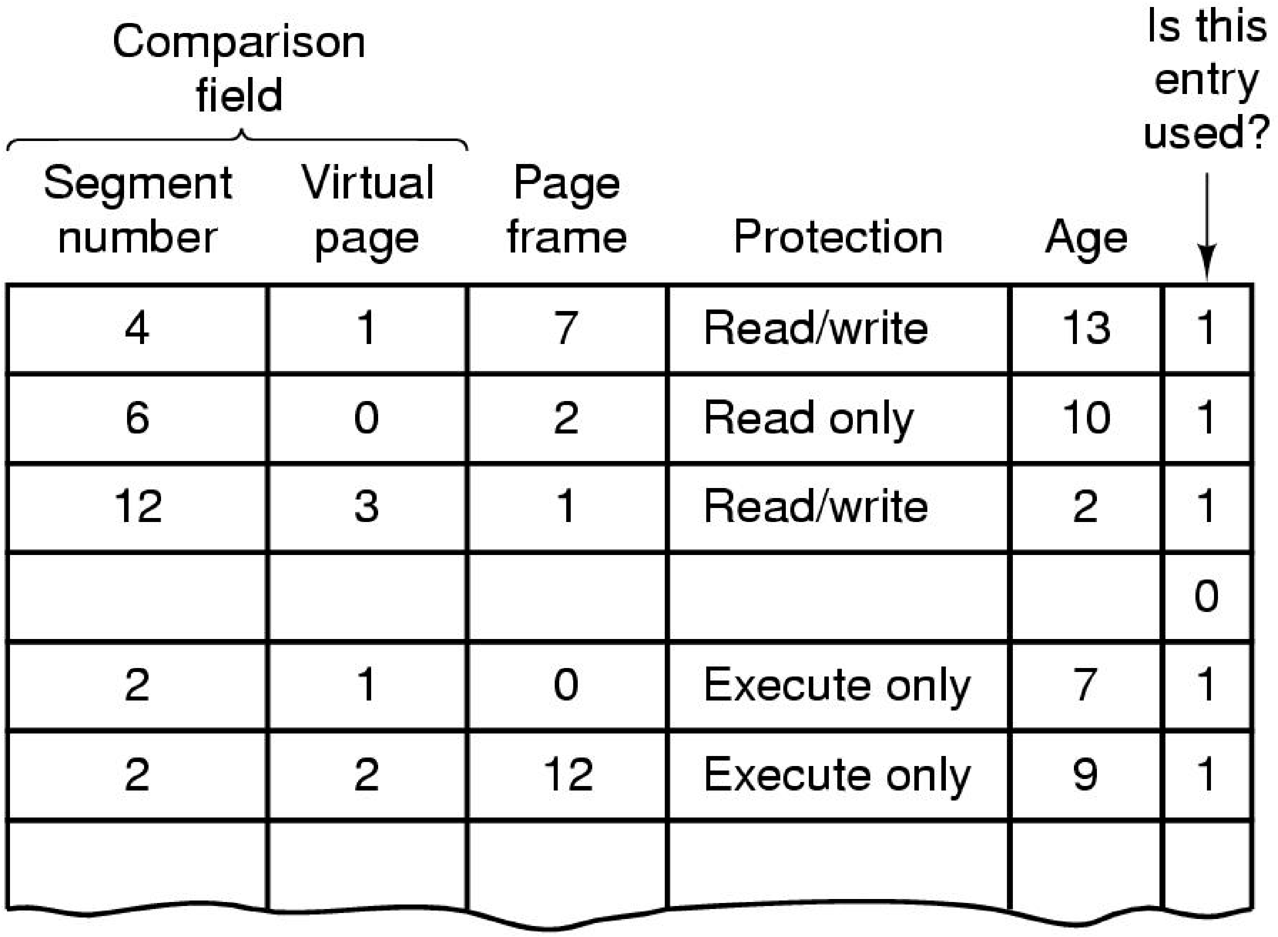
A 34-bit MULTICS virtual address

Segmentation with Paging: MULTICS (3)



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

Segmentation with Paging: MULTICS (4)

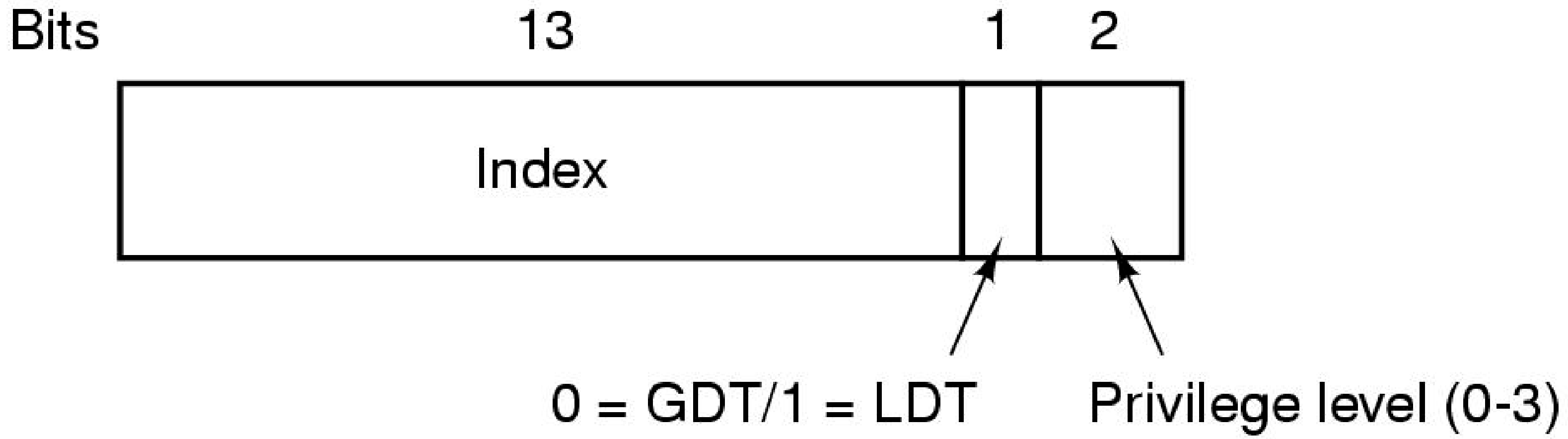


Segment number	Virtual page	Page frame	Protection	Age	Is this entry used?
4	1	7	Read/write	13	1
6	0	2	Read only	10	1
12	3	1	Read/write	2	1
					0
2	1	0	Execute only	7	1
2	2	12	Execute only	9	1

Comparison field

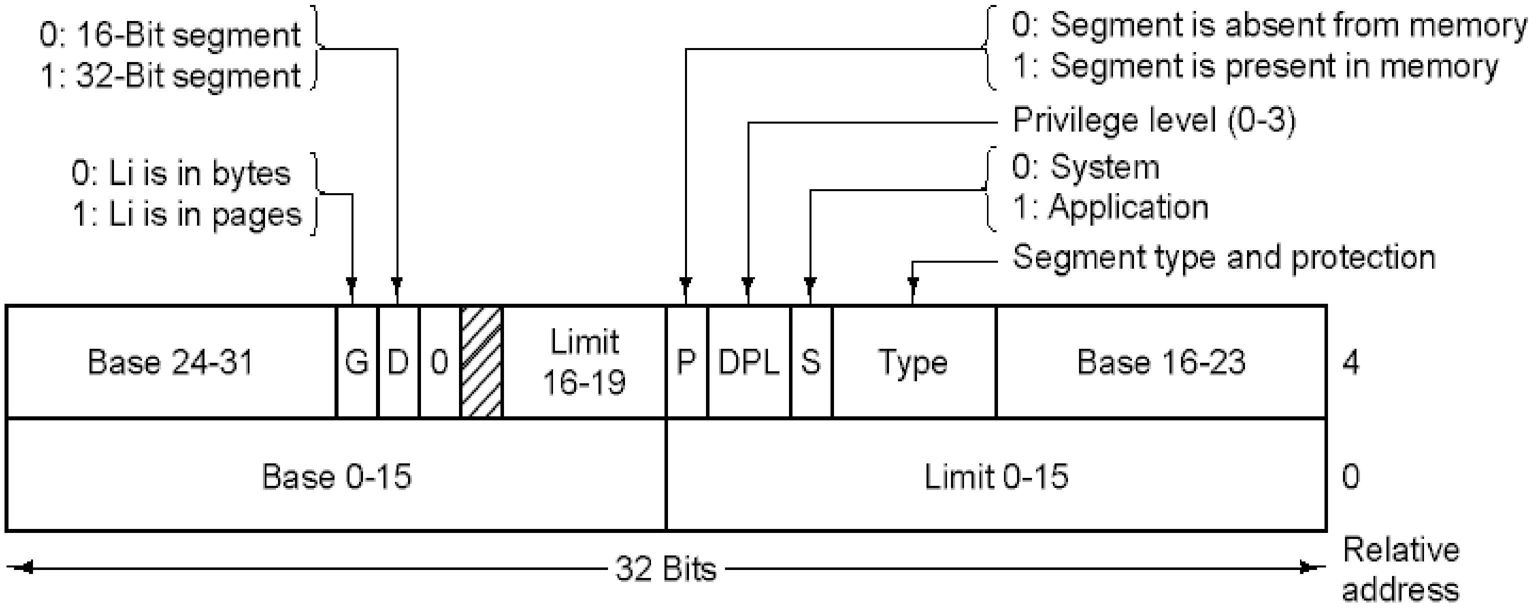
- Simplified version of the MULTICS TLB
- Existence of 2 page sizes makes actual TLB more complicated

Segmentation with Paging: Pentium (1)



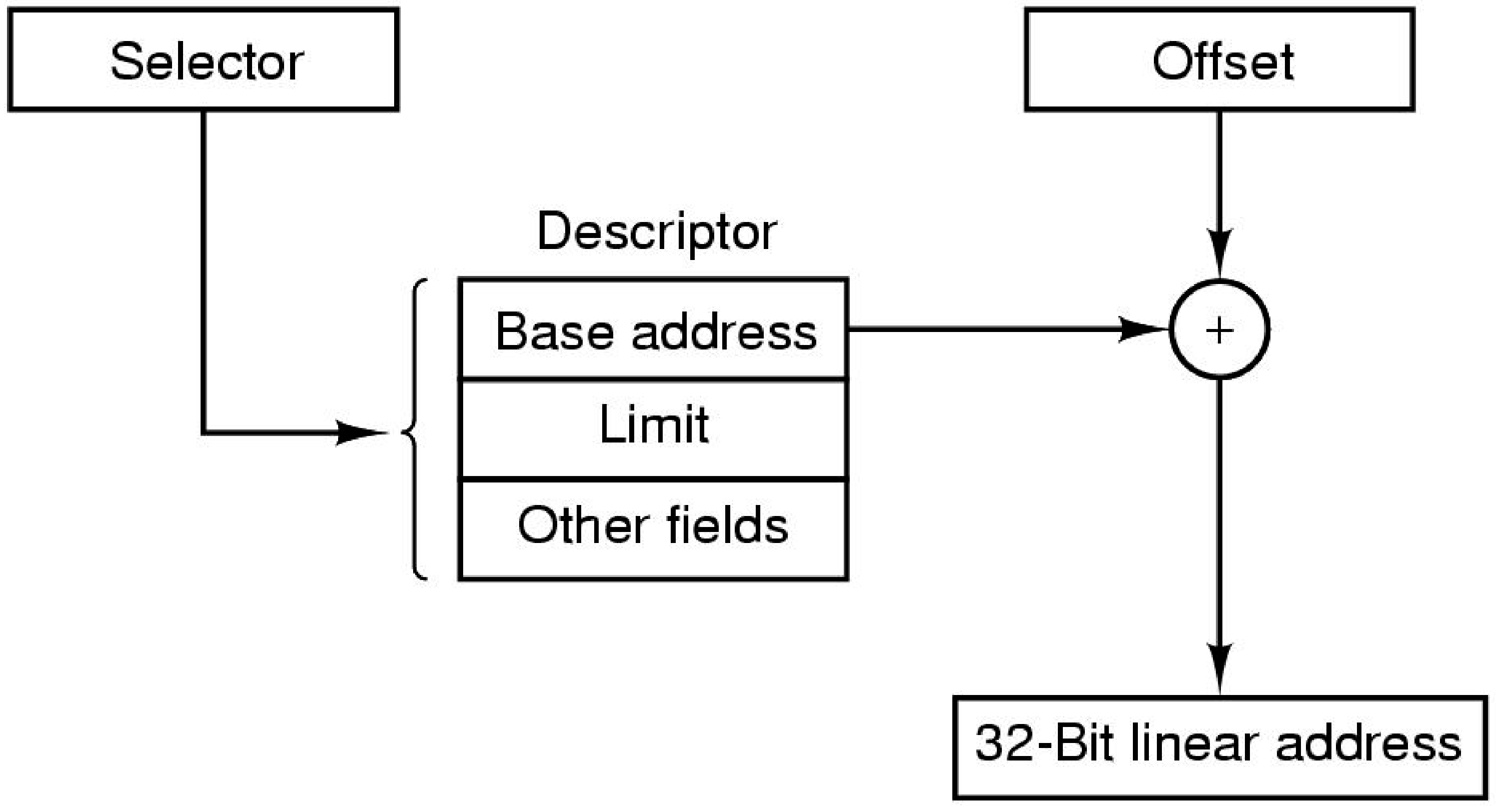
A Pentium selector

Segmentation with Paging: Pentium (2)



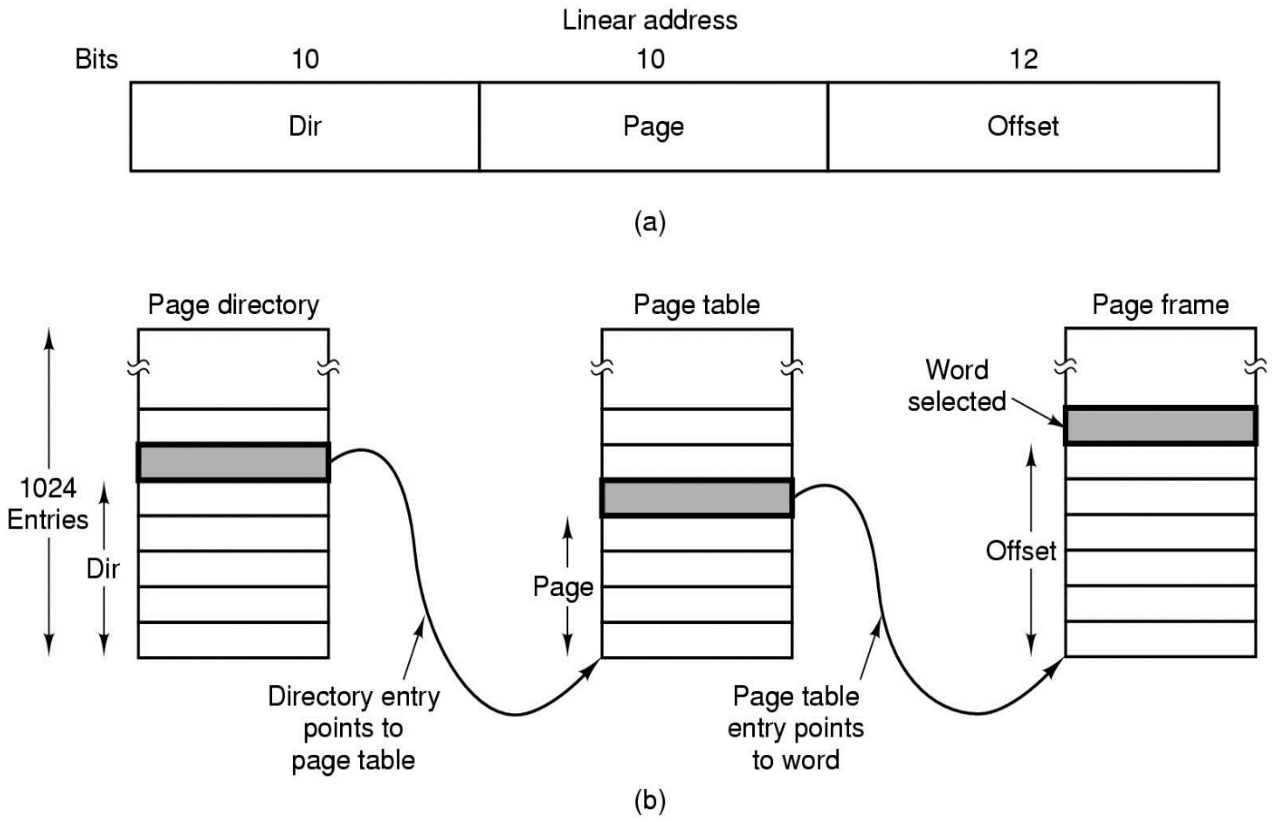
- Pentium code segment descriptor
- Data segments differ slightly

Segmentation with Paging: Pentium (3)



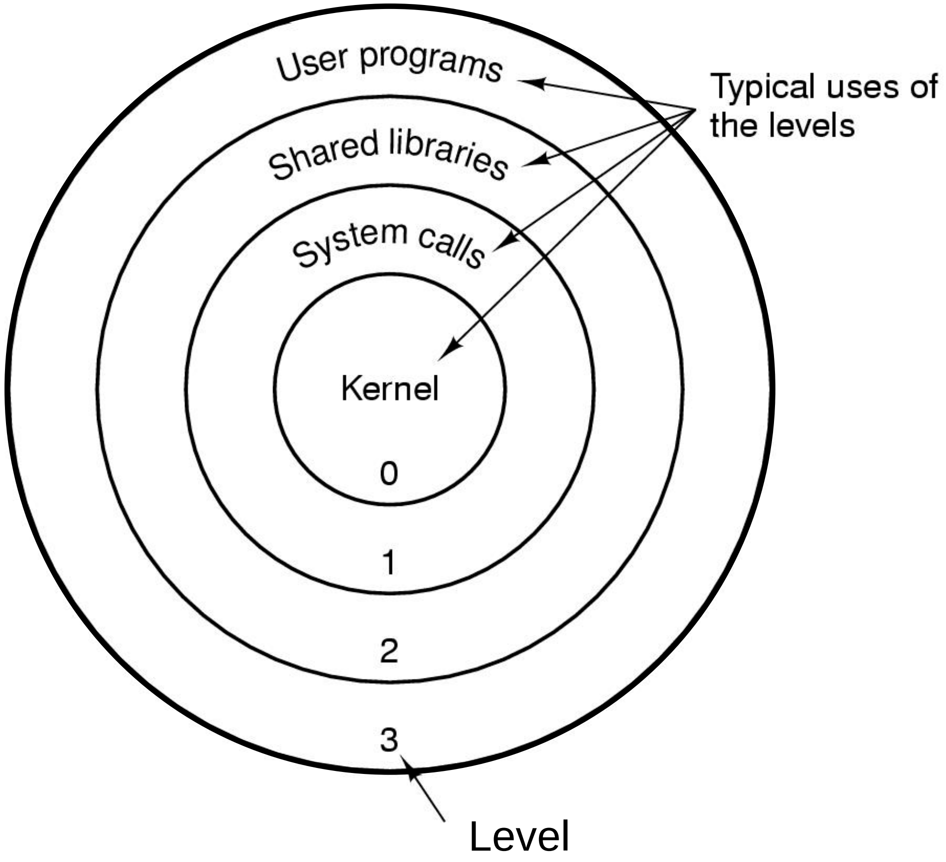
Conversion of a (selector, offset) pair to a linear address

Segmentation with Paging: Pentium (4)



Mapping of a linear address onto a physical address

Segmentation with Paging: Pentium (5)



Protection on the Pentium