Part 3

Memory Management

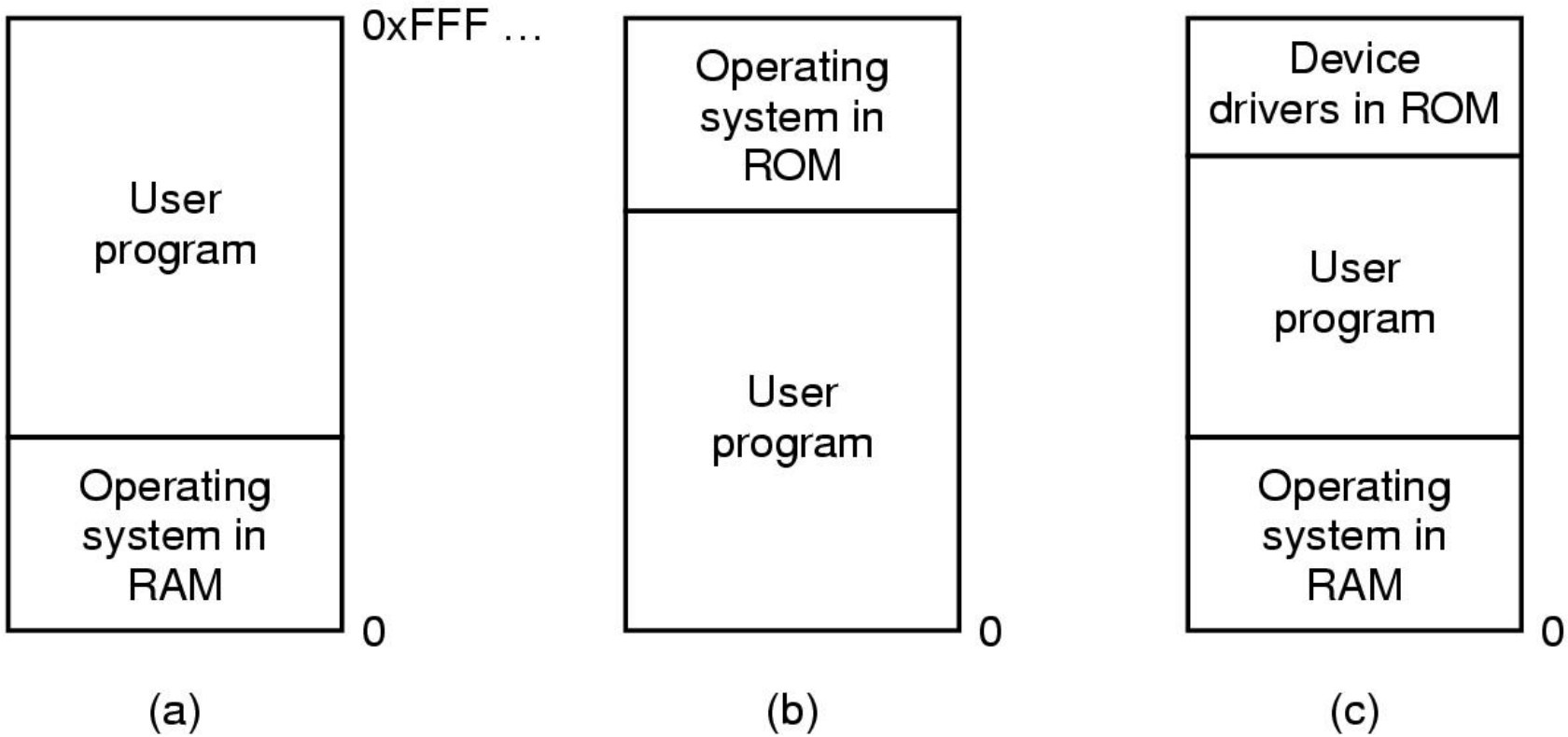
Memory Management

- 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

3.1 No Memory Abstraction

No Memory Abstraction

Monoprogramming without Swapping or Paging

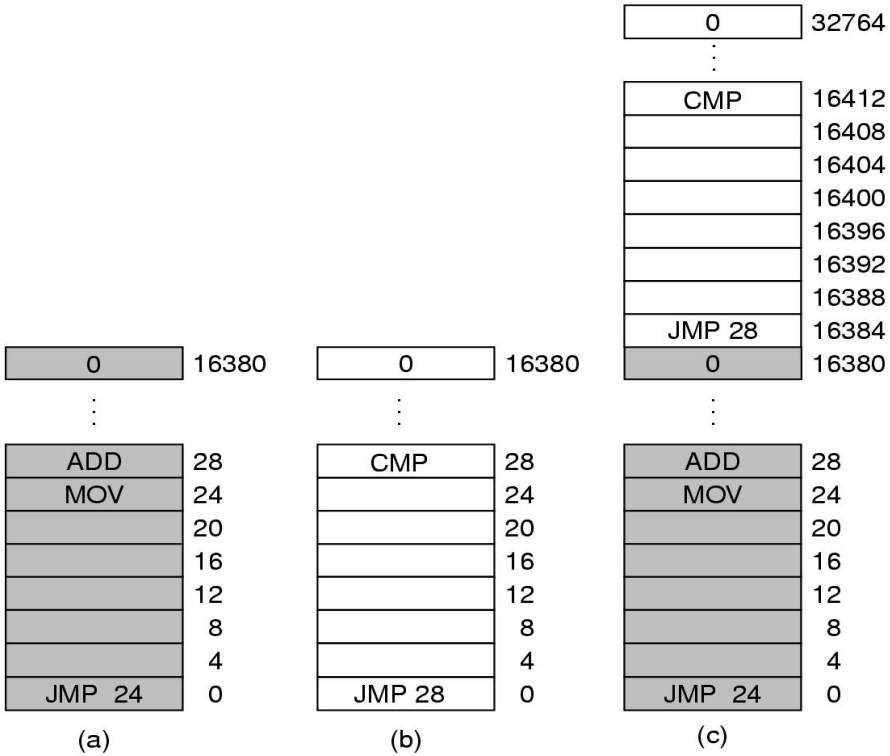


Three simple ways of organizing memory

- an operating system with one user process

No Memory Abstraction

Multiple Programs Without Memory Abstraction

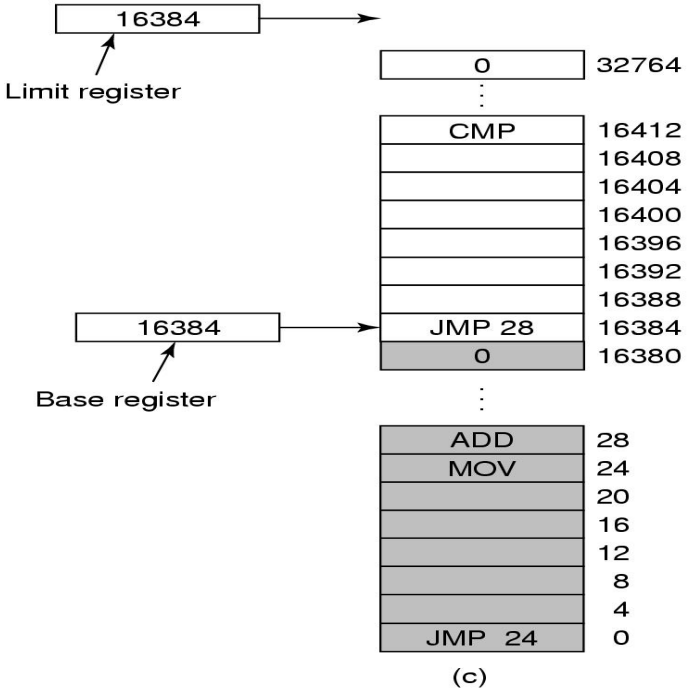


- Illustration of the relocation problem.
- **Static Relocation**: When program is loaded at address n, the constant n is added to every program address during the load process

3.2 Memory Abstraction

Memory Abstraction

Base and Limit Registers

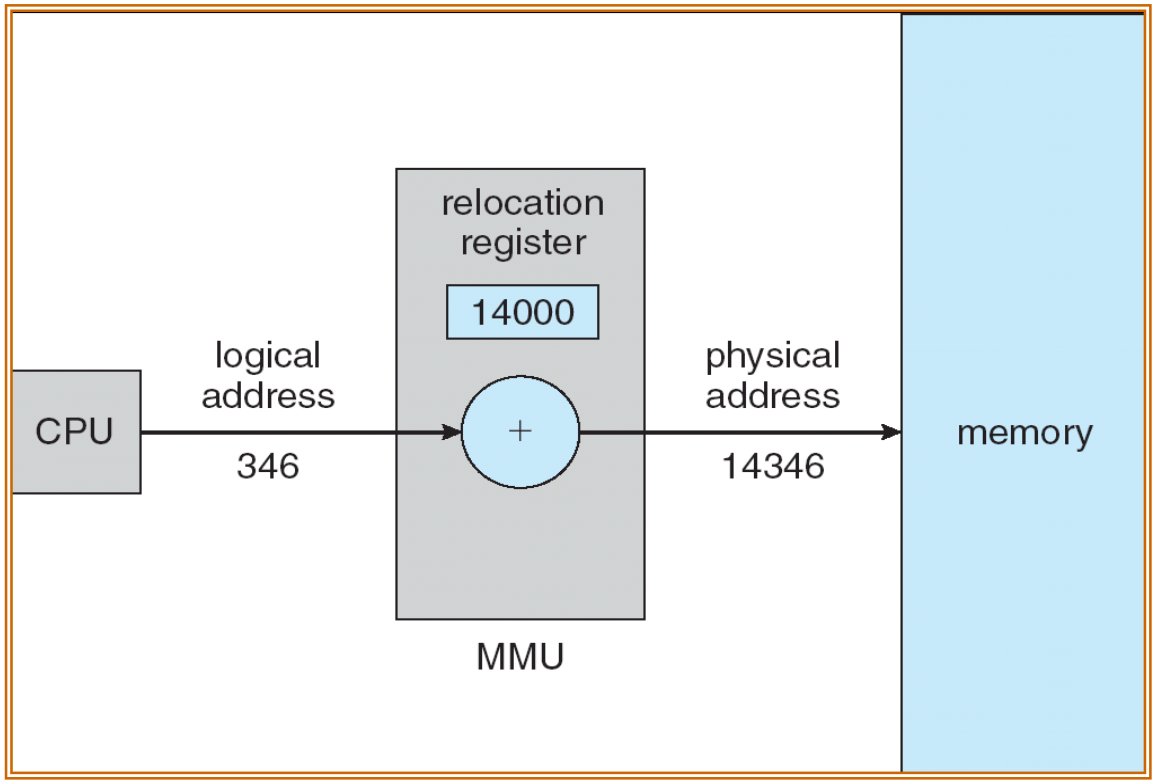


- Base and limit registers can be used to give each process a separate address space.

- **Address space** is the set of addresses that a process can use to address memory

Memory Abstraction

Dynamic relocation using a relocation register



Memory Abstraction
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

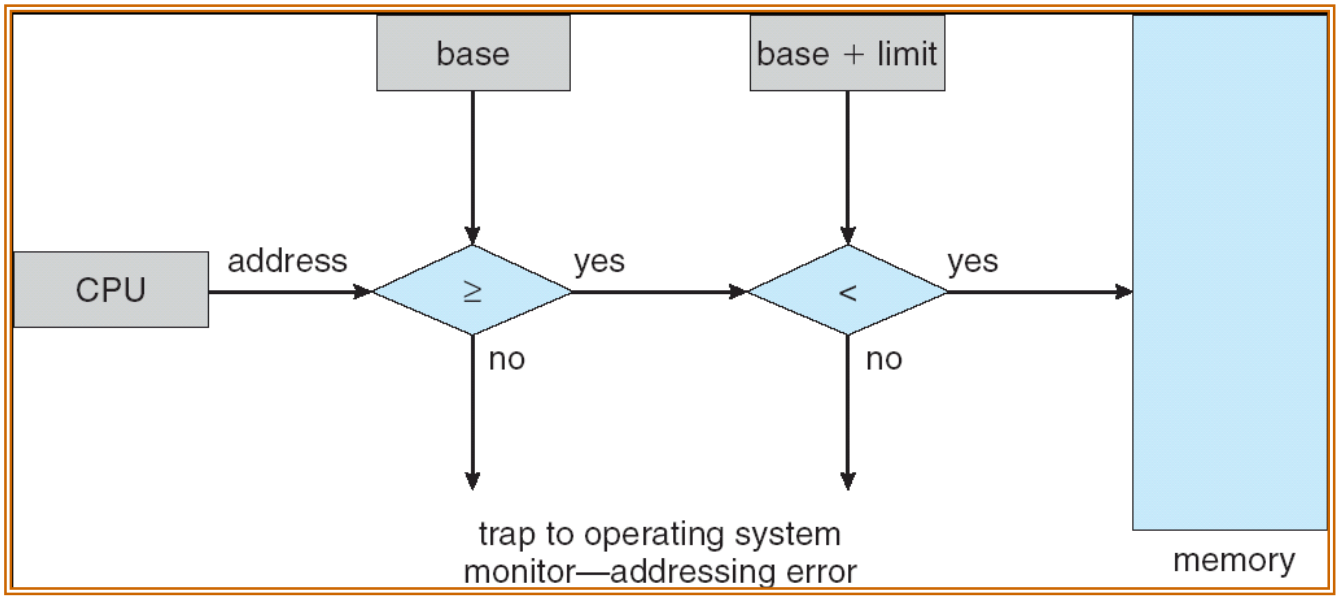
Memory Abstraction

Relocation and Protection

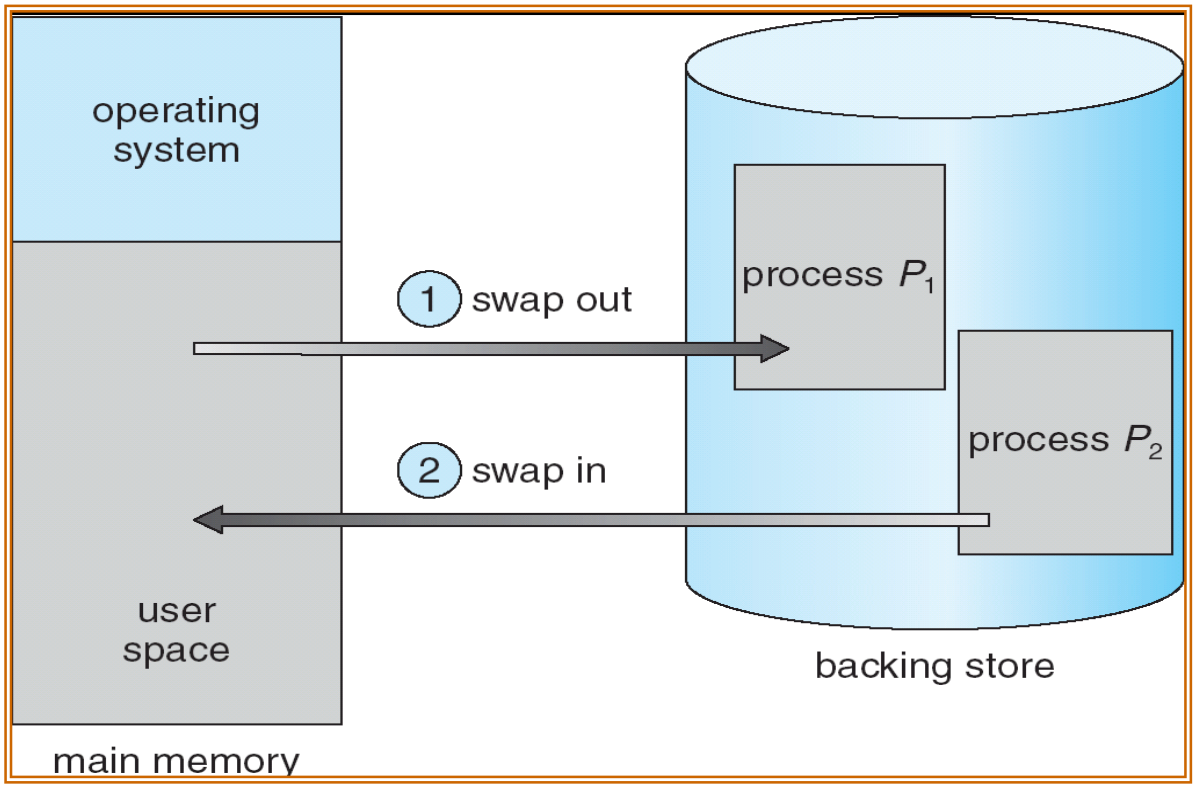
- Relocation registers used to protect user processes from each other, and from changing operating-system code and data
  - Base register contains value of smallest physical address
  - Limit register contains range of logical addresses – each logical address must be less than the limit register
  - MMU maps logical address *dynamically*

Memory Abstraction

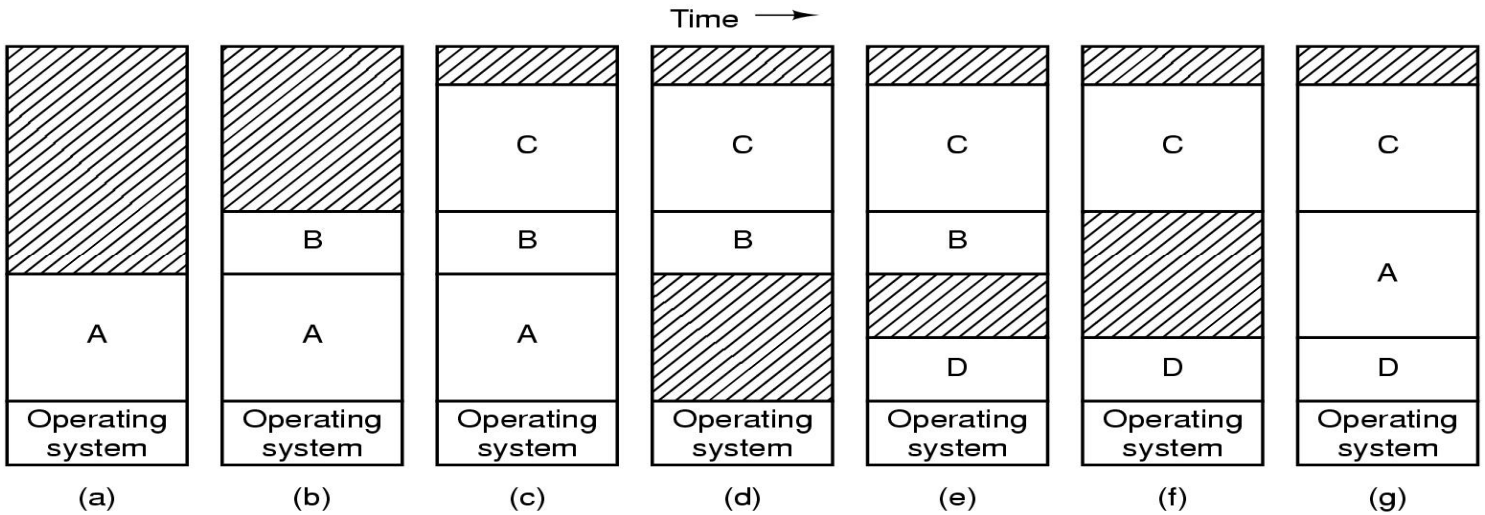
HW address protection with base and limit registers



Memory Abstraction
Swapping (1)
Schematic View of Swapping

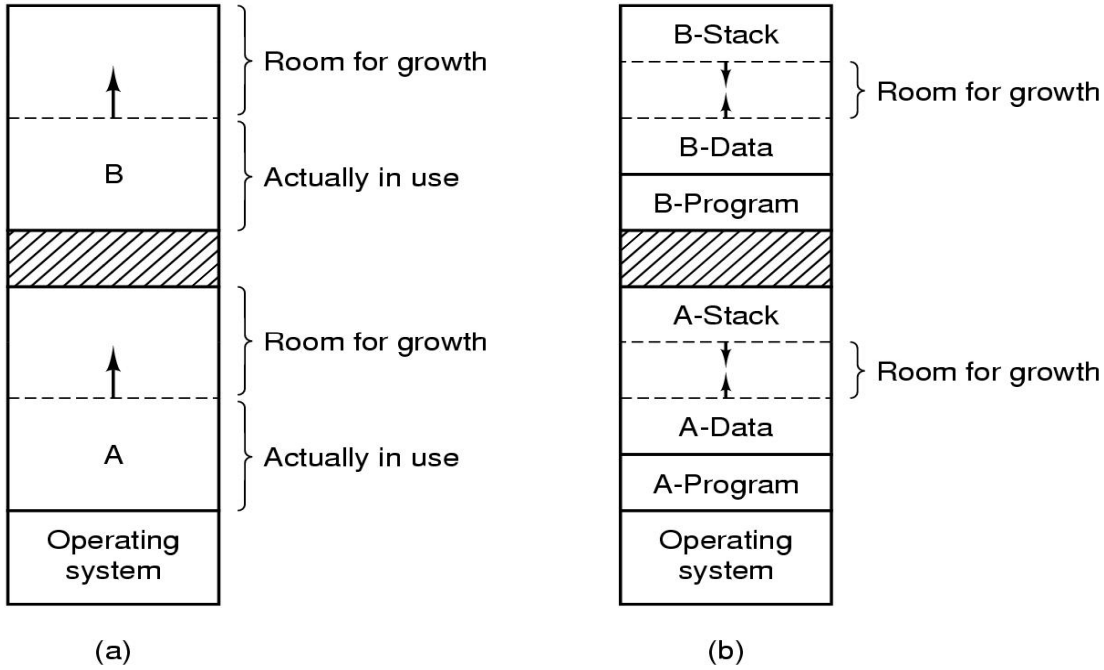


Memory Abstraction
Swapping (2)



- Memory allocation changes as
  - processes come into memory
  - leave memory
- Shaded regions are unused memory
- **External Fragmentation** – total memory space exists to satisfy a request, but it is not contiguous
- **Internal Fragmentation** – allocated memory may be slightly larger than requested memory; this size difference is memory internal to a partition, but not being used

Memory Abstraction
Swapping (3)

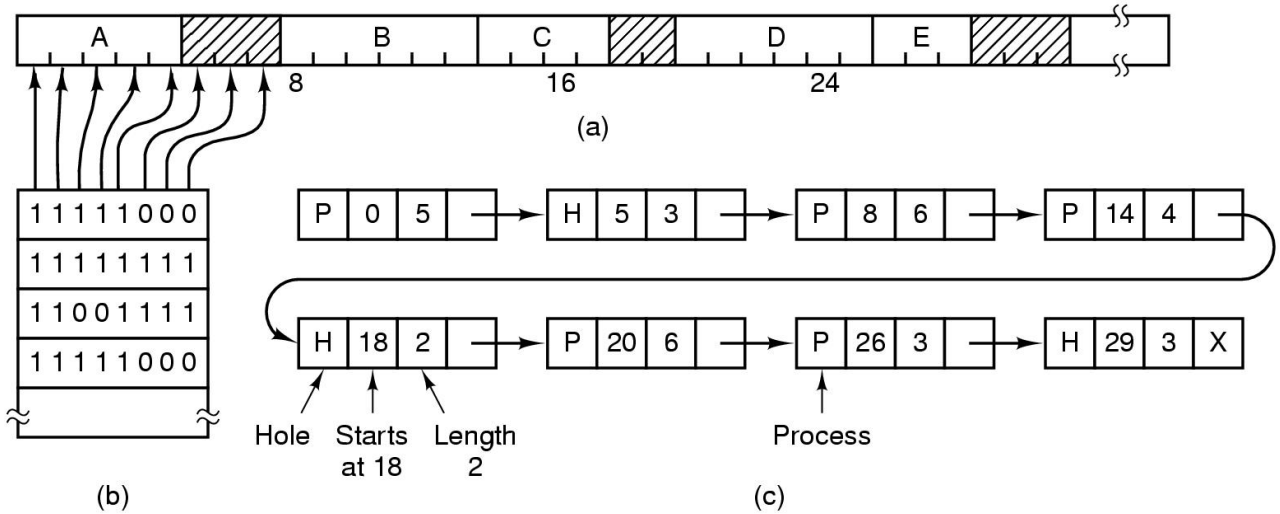


- (a) Allocating space for growing data segment
- (b) Allocating space for growing stack & data segment

Memory Abstraction
Managing free memory (1)

- Multiple-partition allocation
  - Hole – block of available memory; holes of various size are scattered throughout memory
  - When a process arrives, it is allocated memory from a hole large enough to accommodate it
  - Operating system maintains information about:
    a) allocated partitions b) free partitions (hole)
  - There are two ways to keep track of memory usages
    - Memory Management with Bit Maps
    - Memory Management with Linked Lists

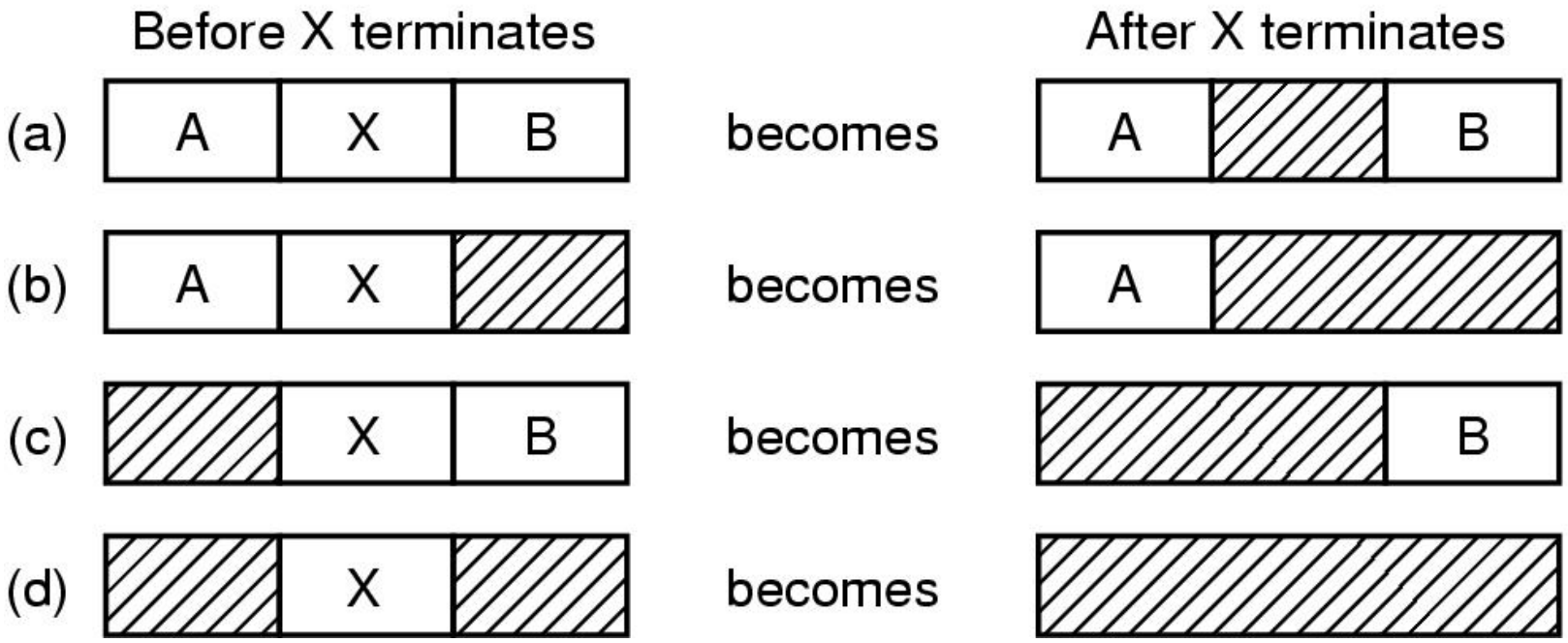
Memory Abstraction
Managing free memory (2)



Memory Management with Bit Maps

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

Memory Abstraction
Managing free memory (3)



Four neighbor combinations
for the terminating process, X.

Memory Abstraction
Managing free memory (4)

How to satisfy a request of size *n* from a list of free holes

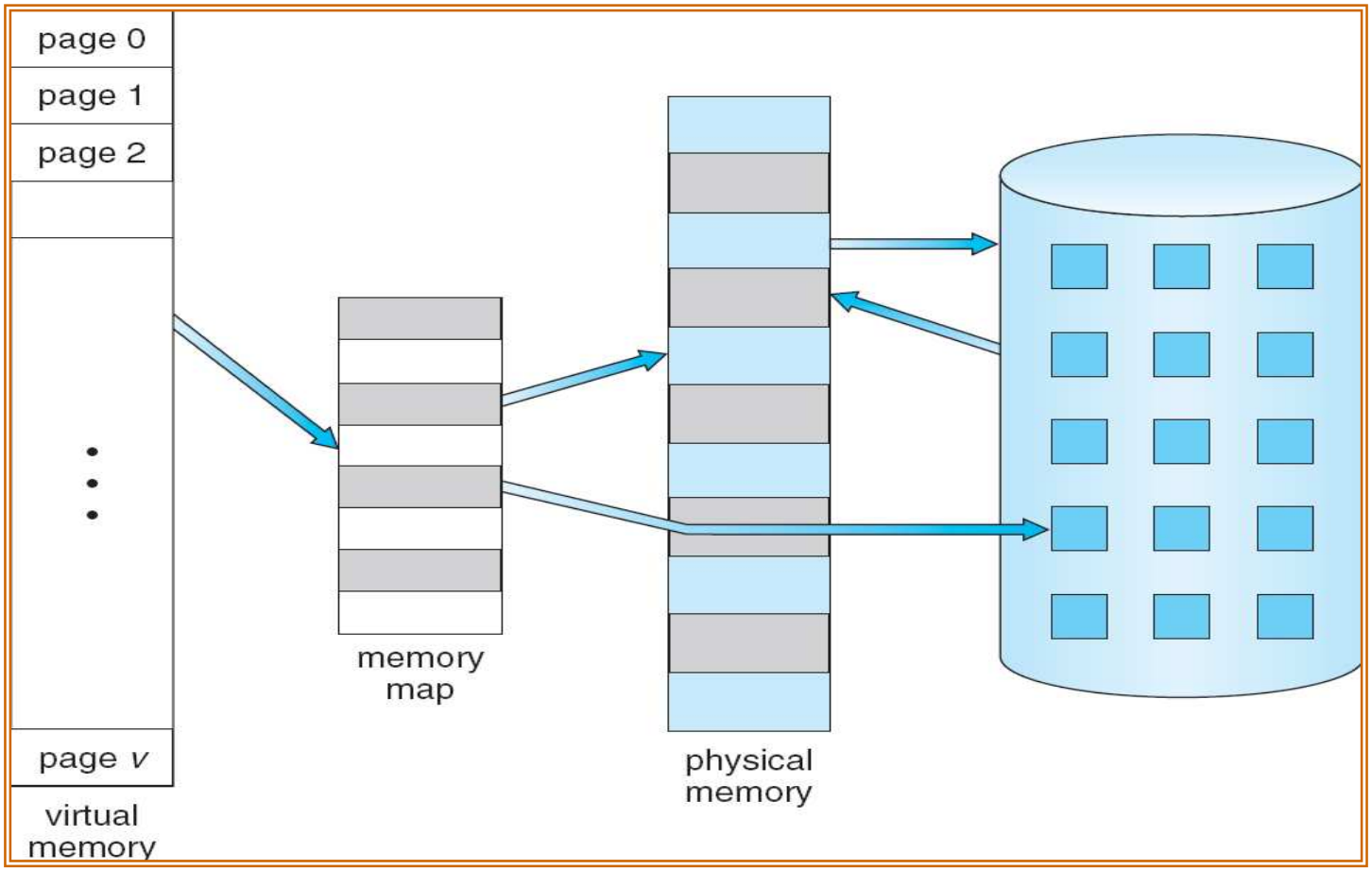
- **First-fit**: Allocate the *first* hole that is big enough
- **Next fit**: Start searching the list from the place where it left off last time
- **Best-fit**: Allocate the *smallest* hole that is big enough; must search entire list, unless ordered by size
  - Produces the smallest leftover hole
- **Worst-fit**: Allocate the *largest* hole; must also search entire list
  - Produces the largest leftover hole
- First-fit and best-fit better than worst-fit in terms of speed and storage utilization

3.3 Virtual Memory

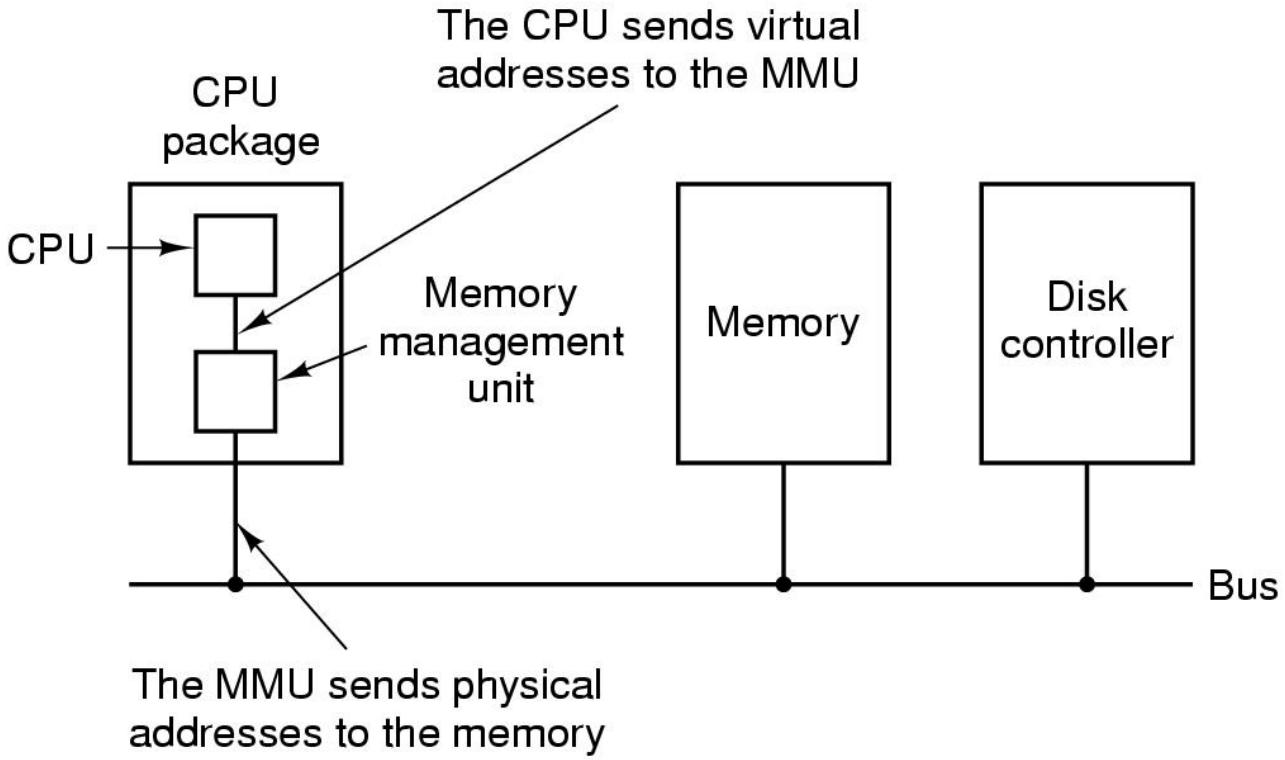
Virtual Memory

- **Virtual memory** – separation of user logical memory from physical memory.
  - Only part of the program needs to be in memory for execution
  - Logical address space can therefore be much larger than physical address space
  - Allows address spaces to be shared by several processes
  - Allows for more efficient process creation
- Virtual memory can be implemented via:
  - Demand **paging**
  - Demand **segmentation**

Virtual Memory
Paging



Virtual Memory
Paging



The position and function of the MMU

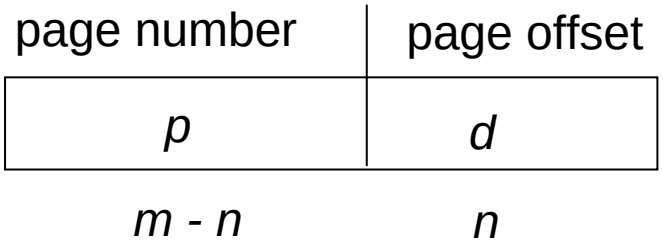
Virtual Memory
Paging

- Virtual address space of a process can be noncontiguous; process is allocated physical memory whenever the latter is available
- Divide physical memory into fixed-sized blocks called **Page frames** (size is power of 2, between 512 bytes and 8,192 bytes)
- Divide logical memory into blocks of same size called **pages**
- Keep track of all free frames
- To run a program of size ***n*** pages, need to find *n* free frames and load program
- Set up a **page table** to translate logical to physical addresses
- Internal fragmentation

Virtual Memory

Address Translation Scheme

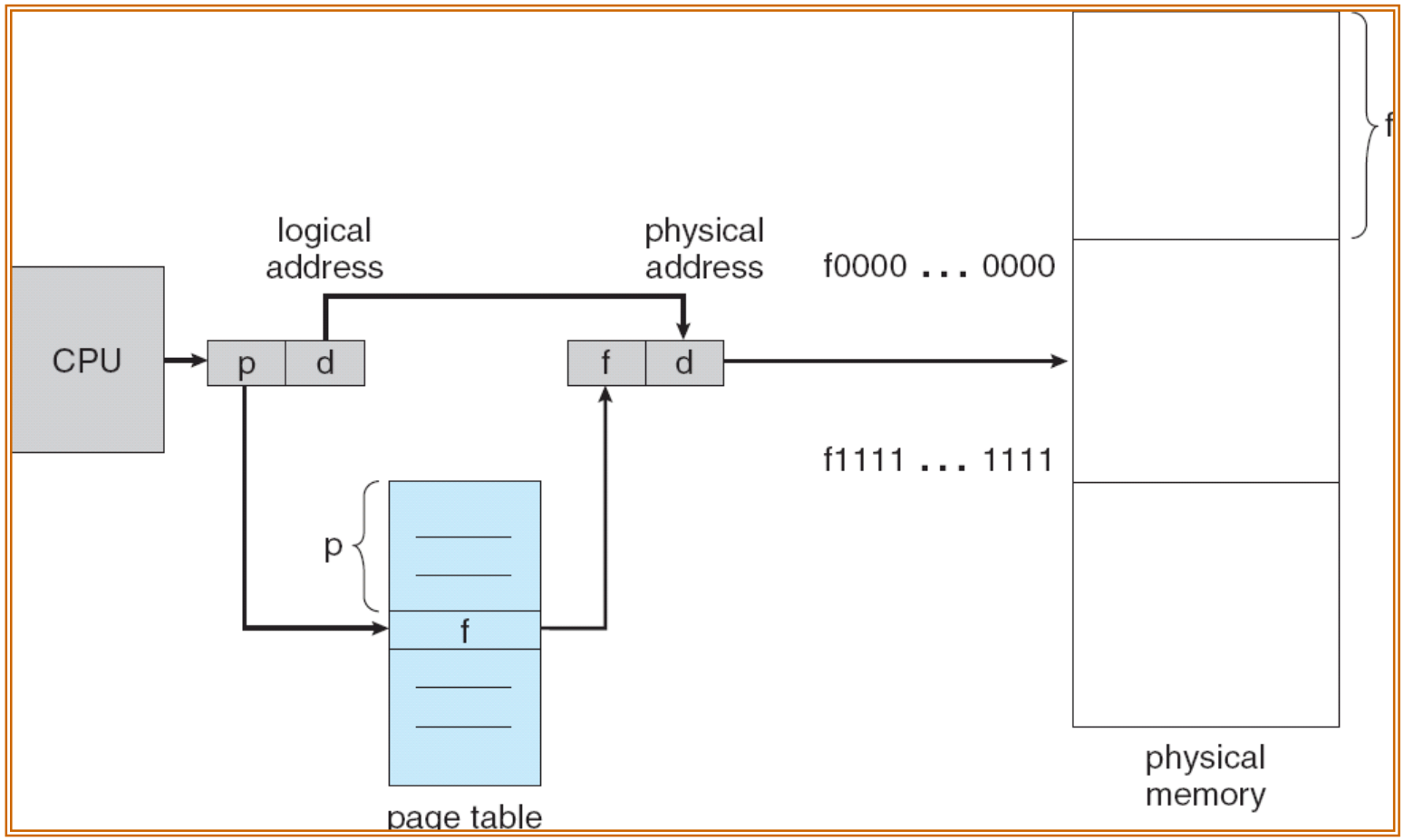
- Address generated by CPU is divided into:
  - **Page number (*p*)** – used as an index into a *page table* which contains base address of each page in physical memory



  - **Page offset (d)** – combined with base address to define the physical memory address that is sent to the memory unit
  - For given logical address space 2^m *and page size* 2^n

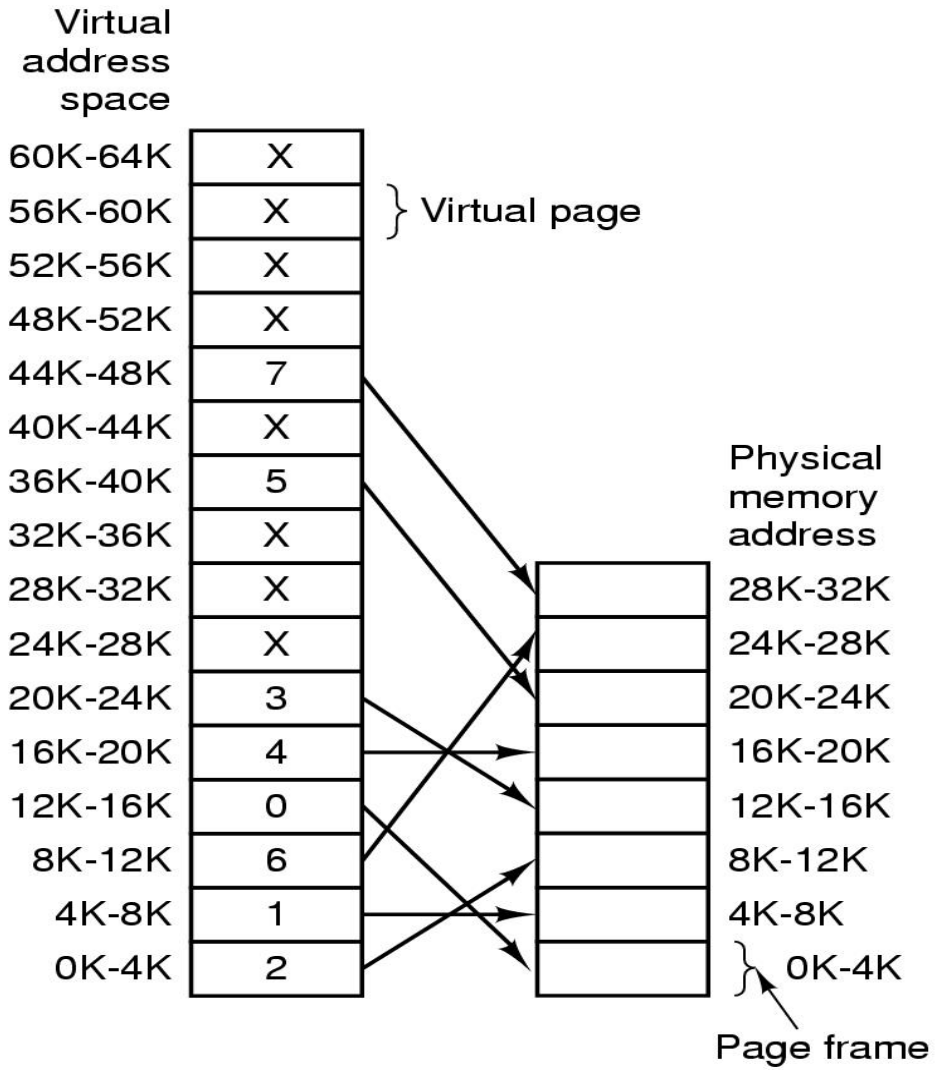
Virtual Memory

Paging Hardware



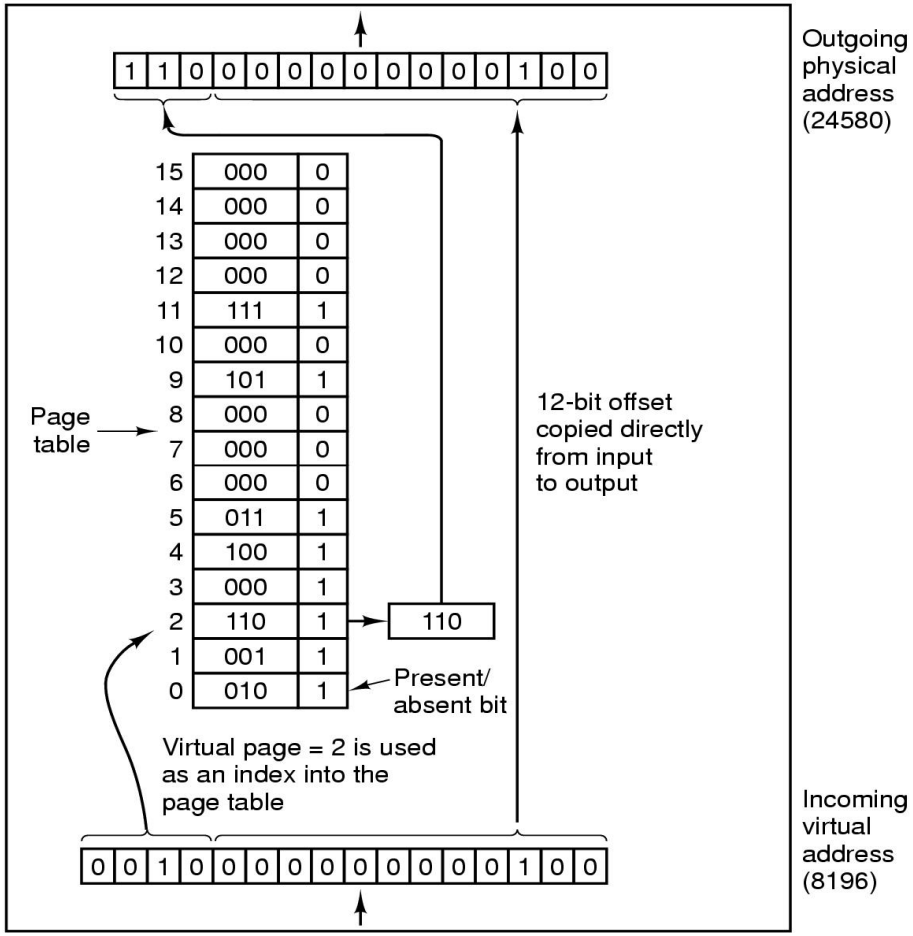
Virtual Memory
Paging: Example

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



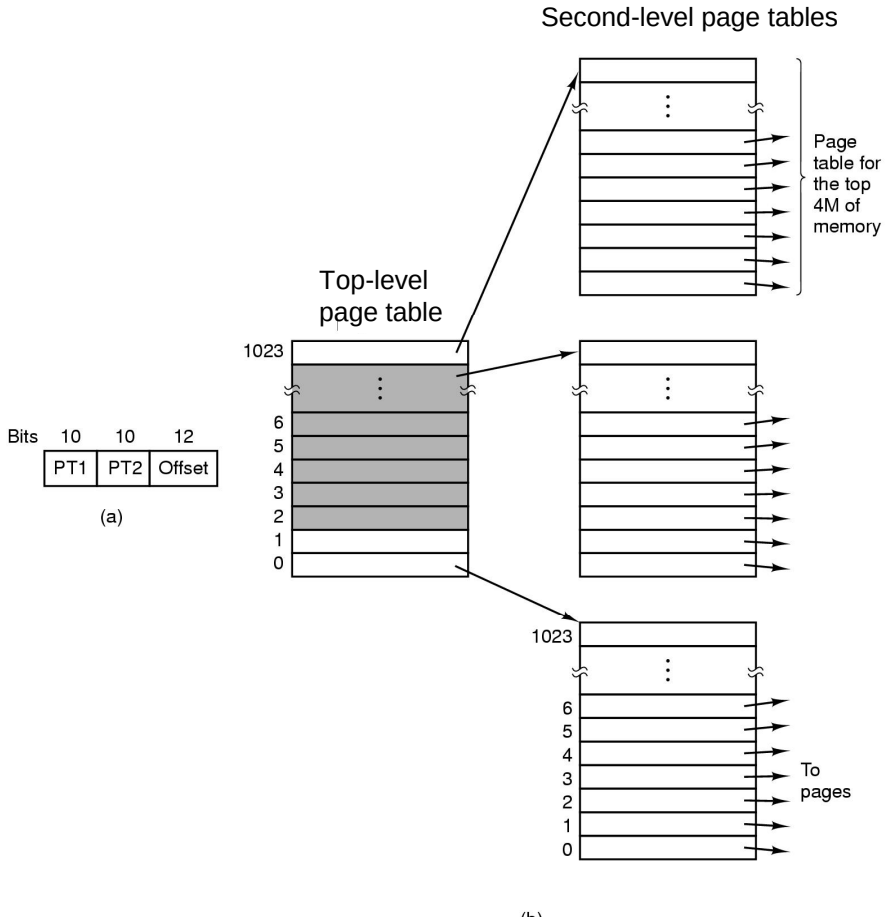
Virtual Memory

Page Tables: Example



Internal operation of MMU with 16 4 KB pages

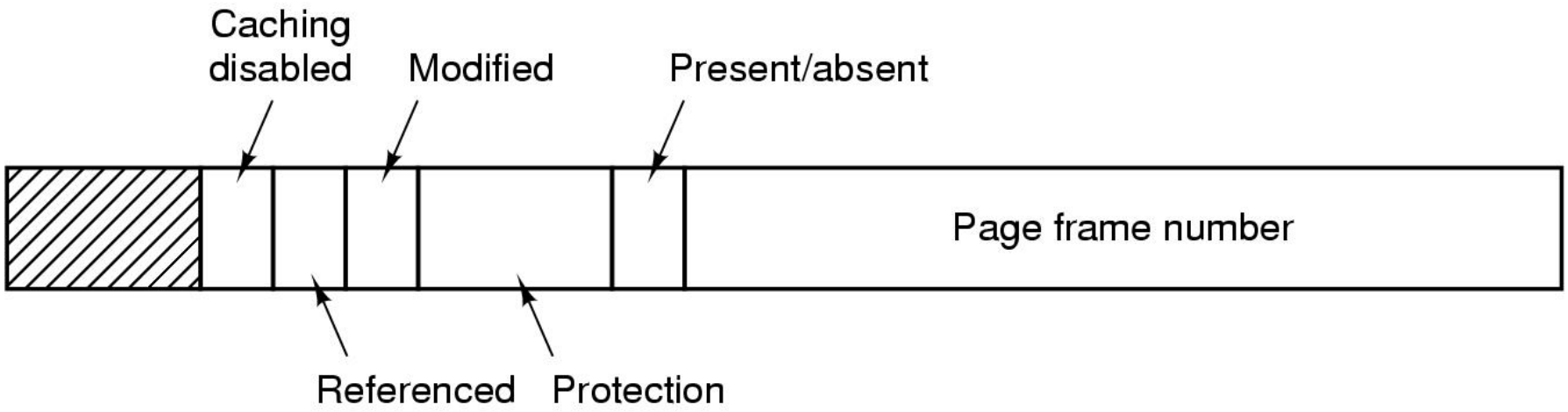
Virtual Memory
Two-level page tables



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

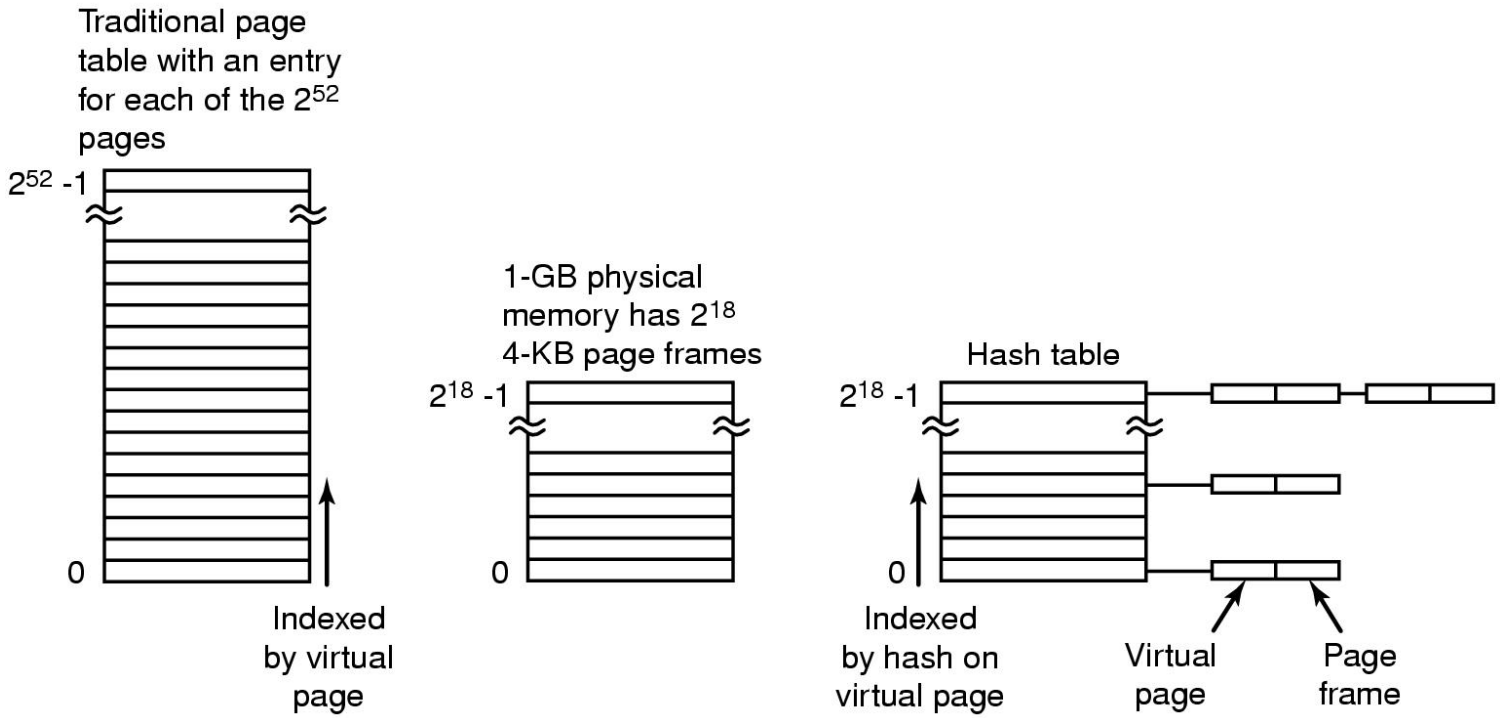
Virtual Memory

Typical page table entry



Typical page table entry

Virtual Memory
Inverted Page Tables



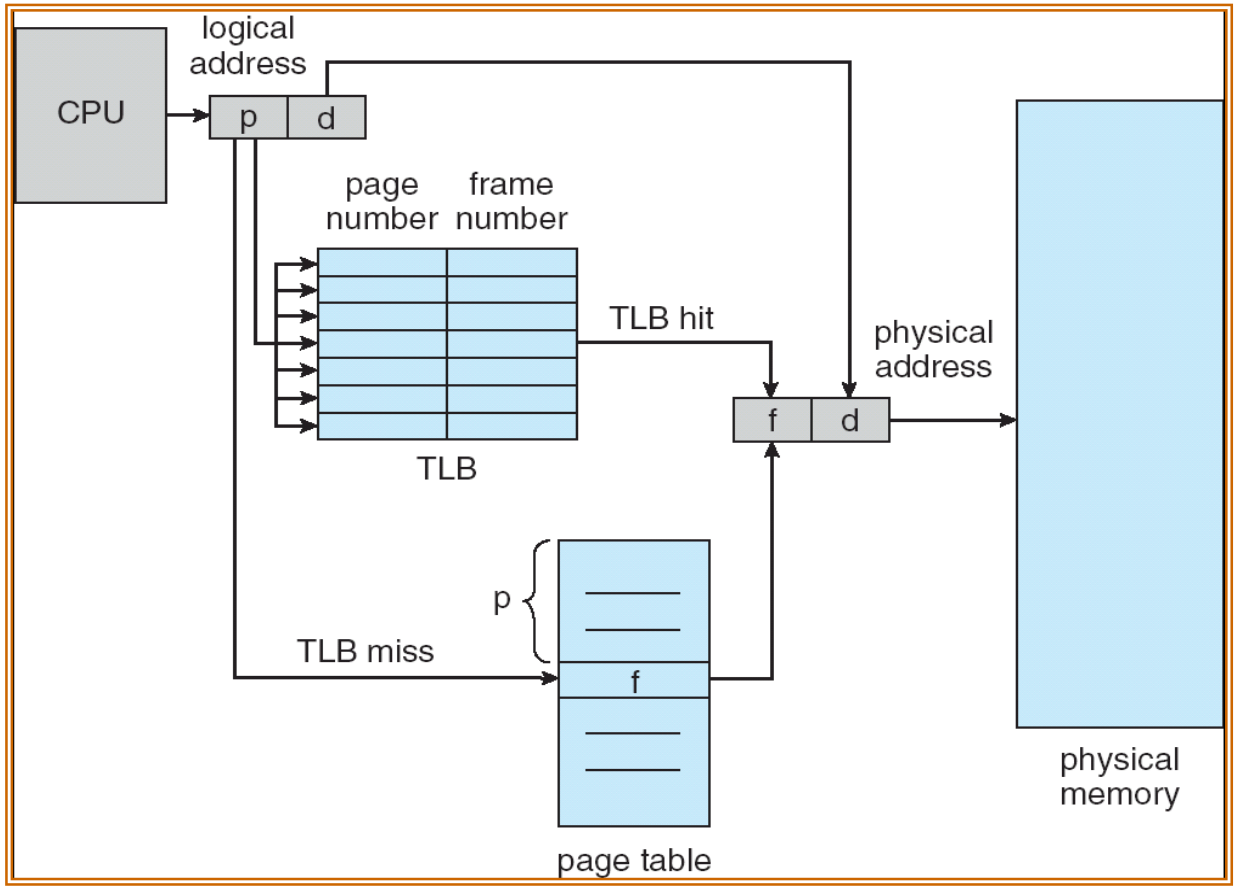
The Entry of The inverted page table keeps track of which (Process, virtual page) is located in the page frame.

Virtual Memory
Implementation of Page Table

- Page table is kept in main memory
- **Page-table base register (PTBR)** points to the page table
- **Page-table length register (PRLR)** indicates size of the page table
- In this scheme every data/instruction access requires two memory accesses. One for the page table and one for the data/instruction.
- The two memory access problem can be solved by the use of a special fast-lookup hardware cache called **associative memory** or **translation look-aside buffers (TLBs)**

Virtual Memory

Paging Hardware With TLB



Virtual Memory

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

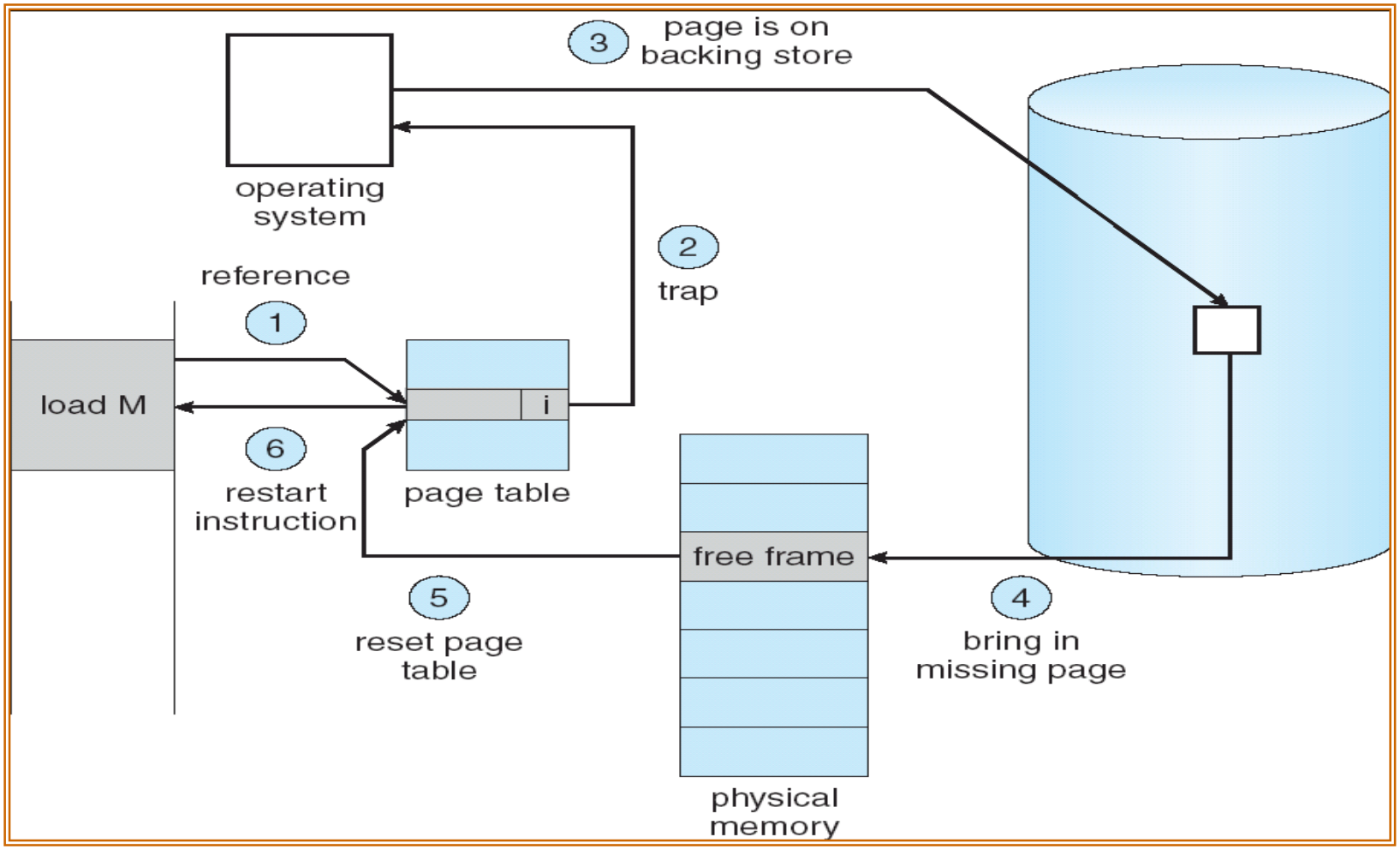
A TLB to speed up paging

Virtual Memory
Page Fault

1. If there is a reference to a page, Just not in memory: **page fault,**
2. Trap to operating system:
3. Get empty page frame, Determine page on backing store
4. Swap page from disk into page frame in memory
5. Modifies page tables, Set validation bit = **v**
6. Restart the instruction that caused the page fault

Virtual Memory

Steps in Handling a Page Fault



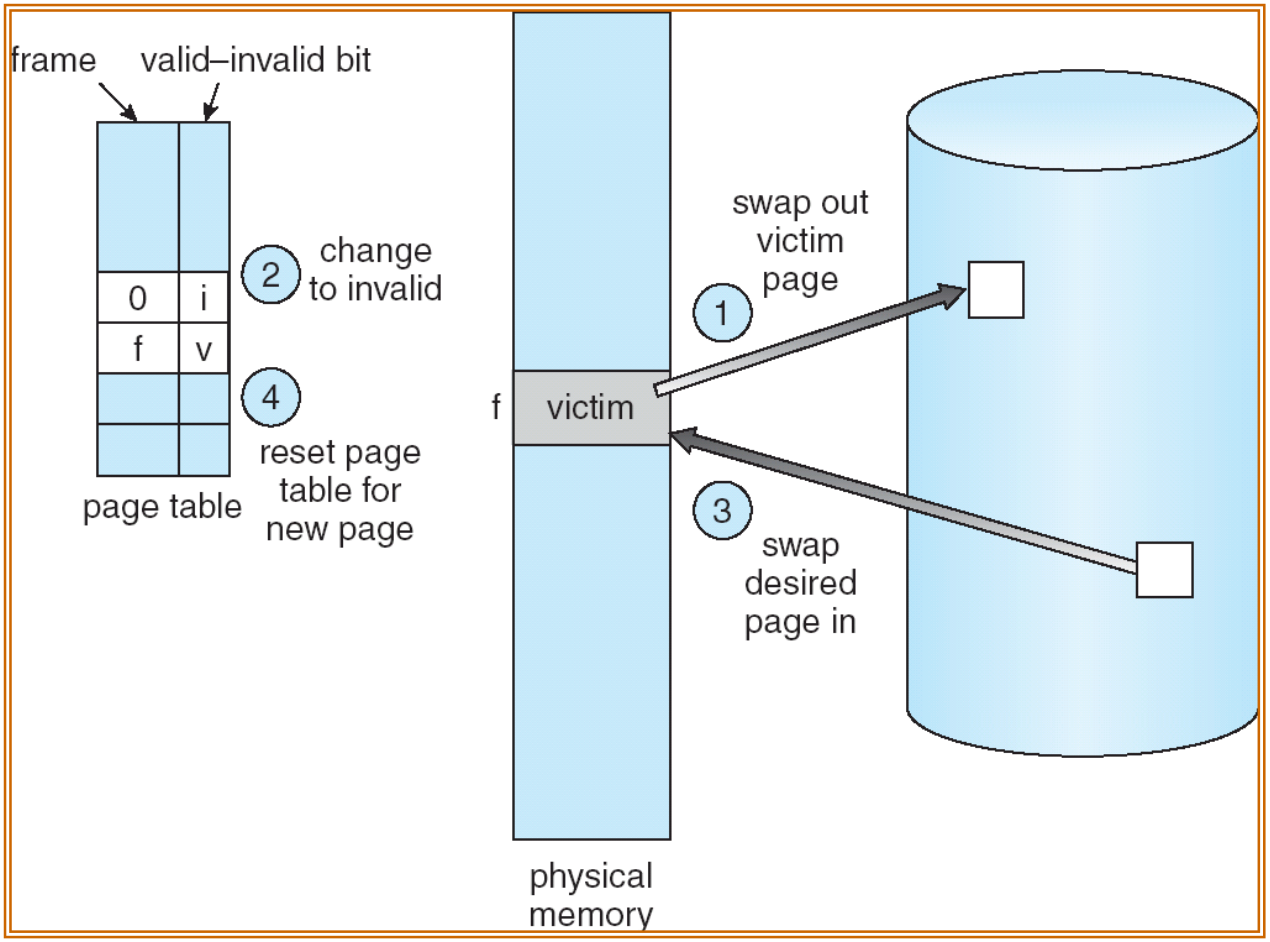
Virtual Memory
Page Replacement

- What happens if there is no free frame?
- Page replacement – find some page in memory, but not really in use, swap it out
  - algorithm
  - performance – want an algorithm which will result in minimum number of page faults
- Same page may be brought into memory several times

Virtual Memory
Basic Page Replacement

1. Find the location of the desired page on disk
2. Find a free frame:
   - If there is a free frame, use it
   - If there is no free frame, use a page replacement algorithm to select a **victim** frame
3. Bring the desired page into the (newly) free frame; update the page and frame tables
4. Restart the process

Virtual Memory
Page Replacement



3.4 Page Replacement Algorithms

Page Replacement Algorithms

- Optimal page replacement algorithm
- Not recently used page replacement
- First-In, First-Out page replacement
- Second chance page replacement
- Clock page replacement
- Least recently used page replacement
- Working set page replacement
- WSClock page replacement

Page Replacement Algorithms

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

Page Replacement Algorithms

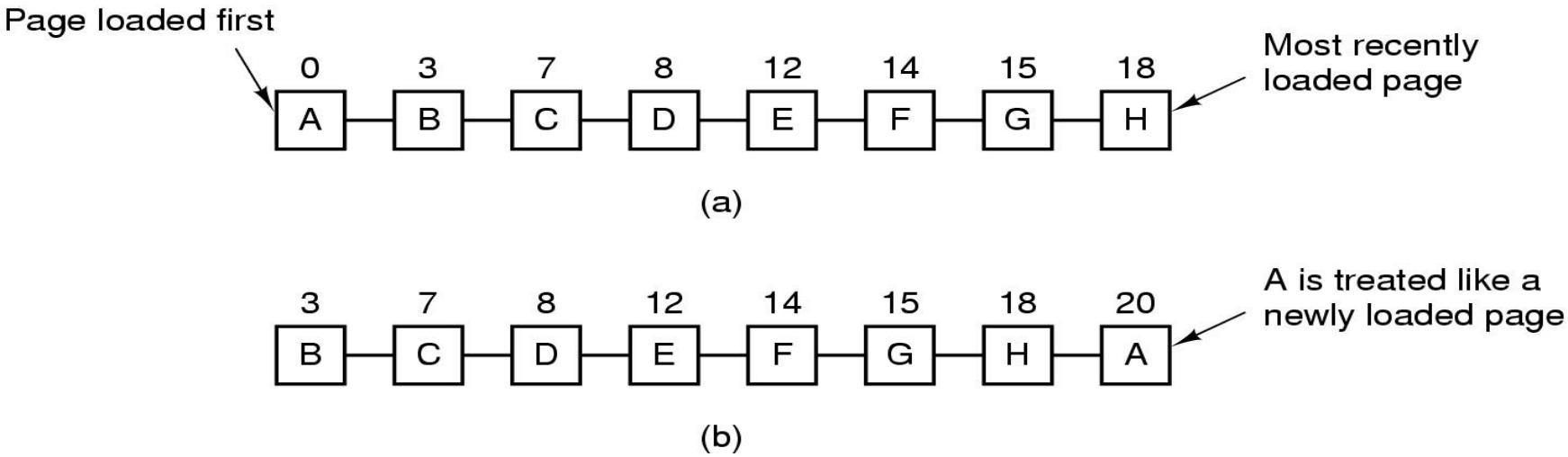
Not Recently Used Page Replacement Algorithm

- 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

Page Replacement Algorithms
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

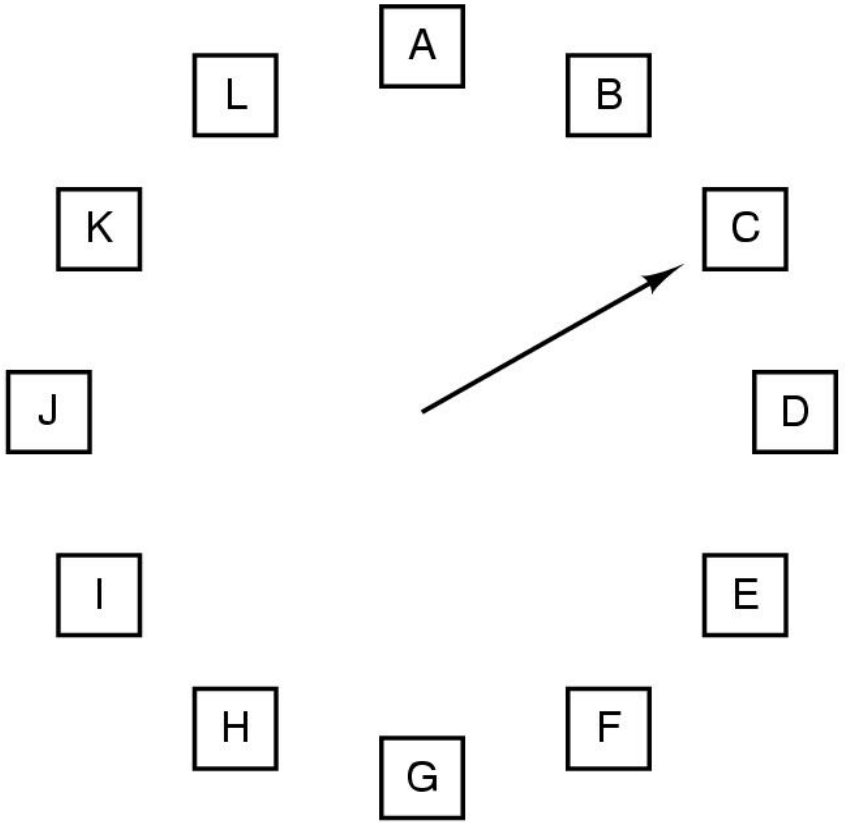
Page Replacement Algorithms
Second Chance Page Replacement Algorithm



- 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)

Page Replacement Algorithms

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

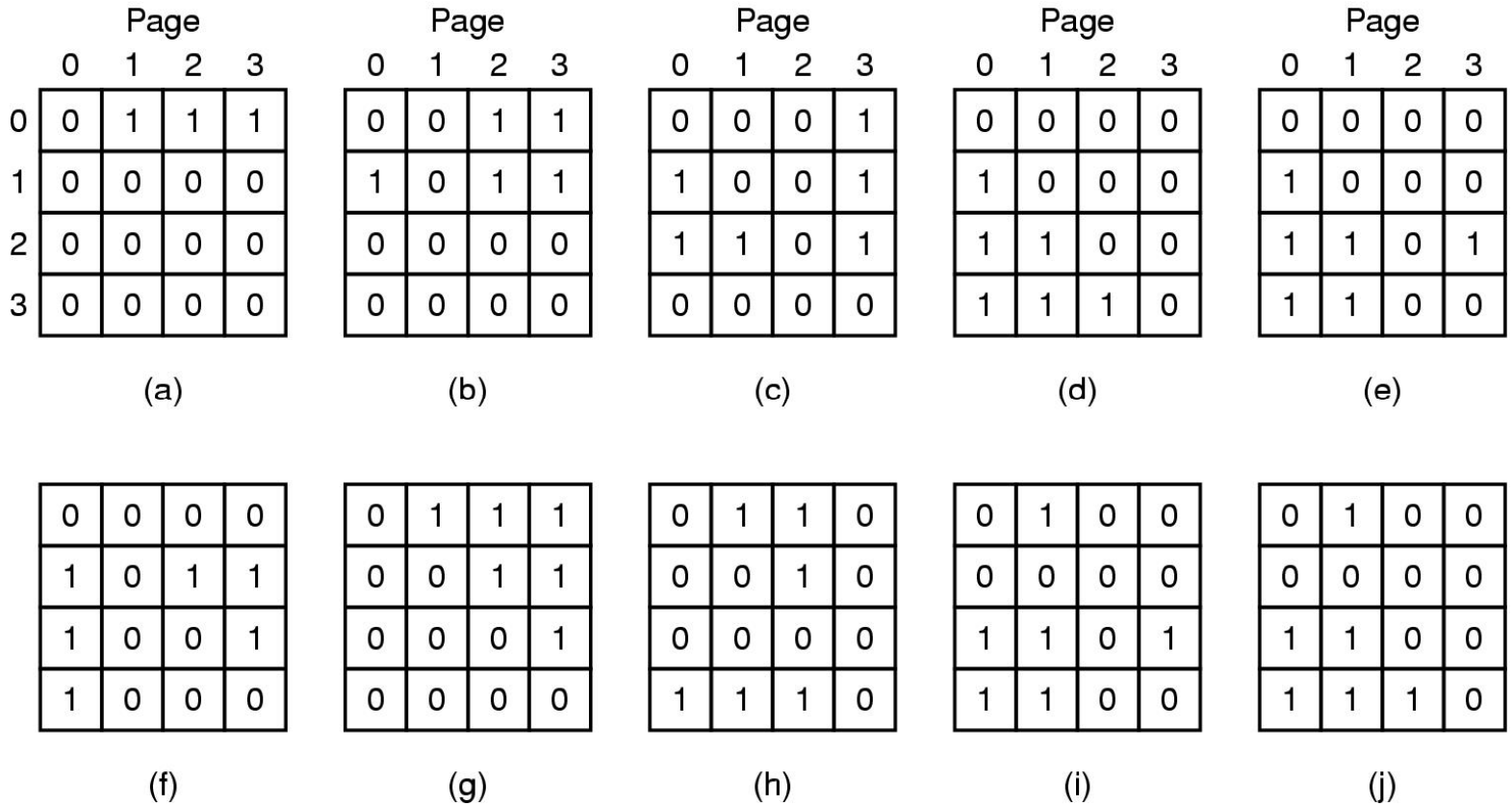
R = 1: Clear R and advance hand

Page Replacement Algorithms
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 every memory reference !!
- Alternatively keep counter in each page table entry
  - choose page with lowest value counter
  - periodically zero the counter

Page Replacement Algorithms

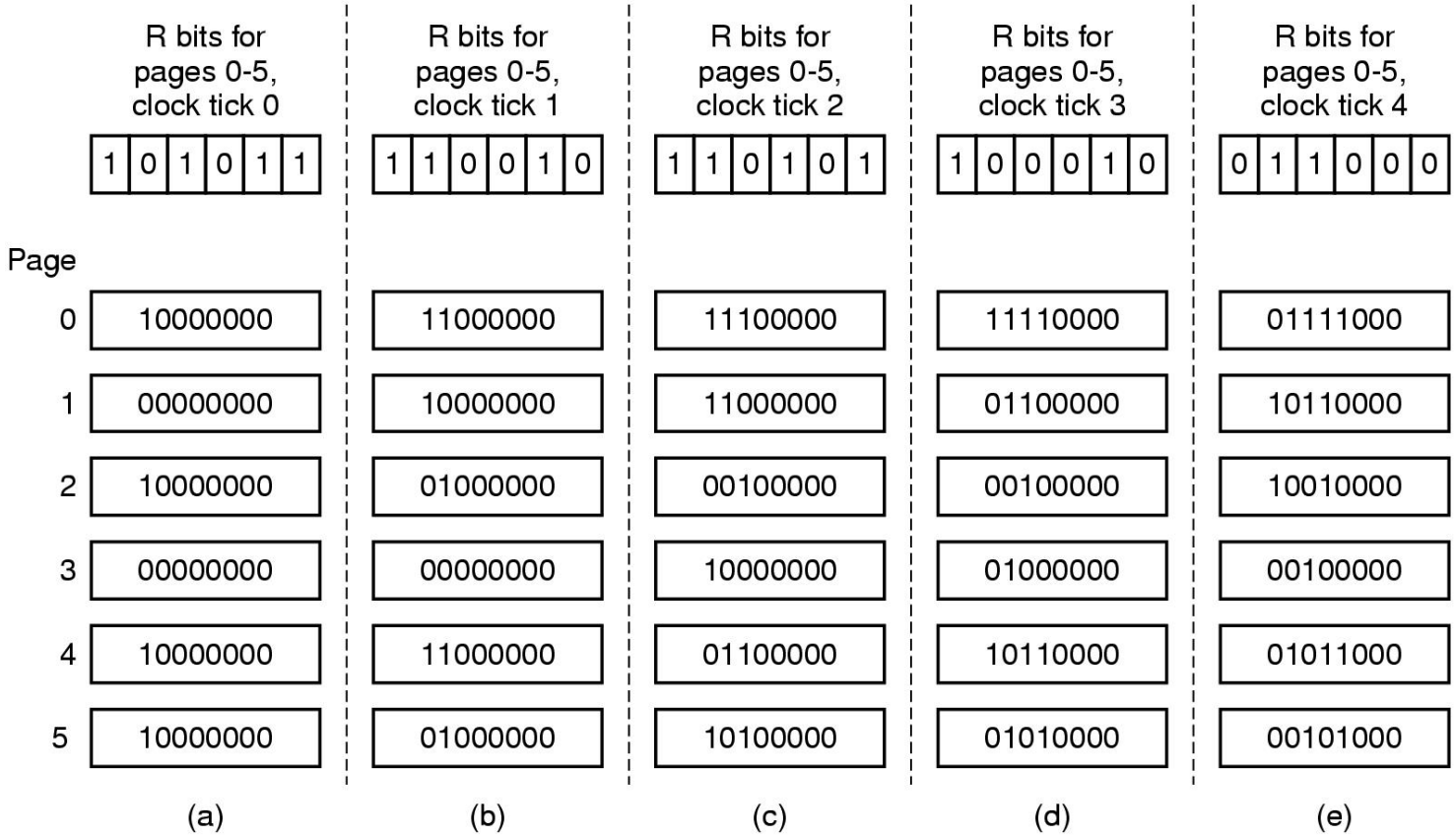
LRU



LRU using a matrix when pages are referenced in the order 0, 1, 2, 3, 2, 1, 0, 3, 2, 3.

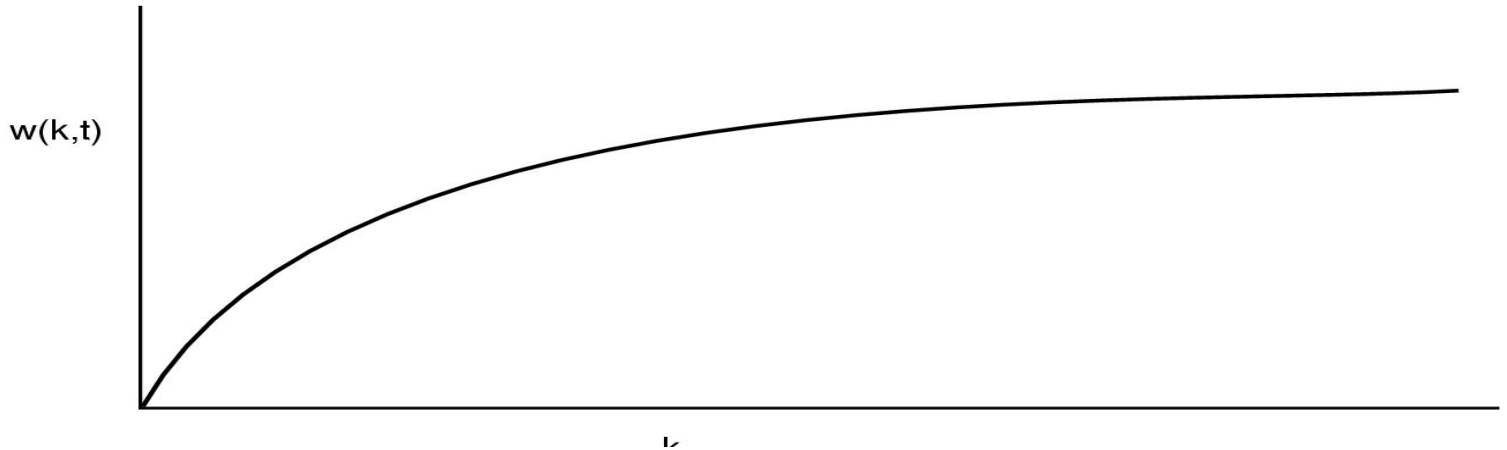
Page Replacement Algorithms

Simulating LRU in Software



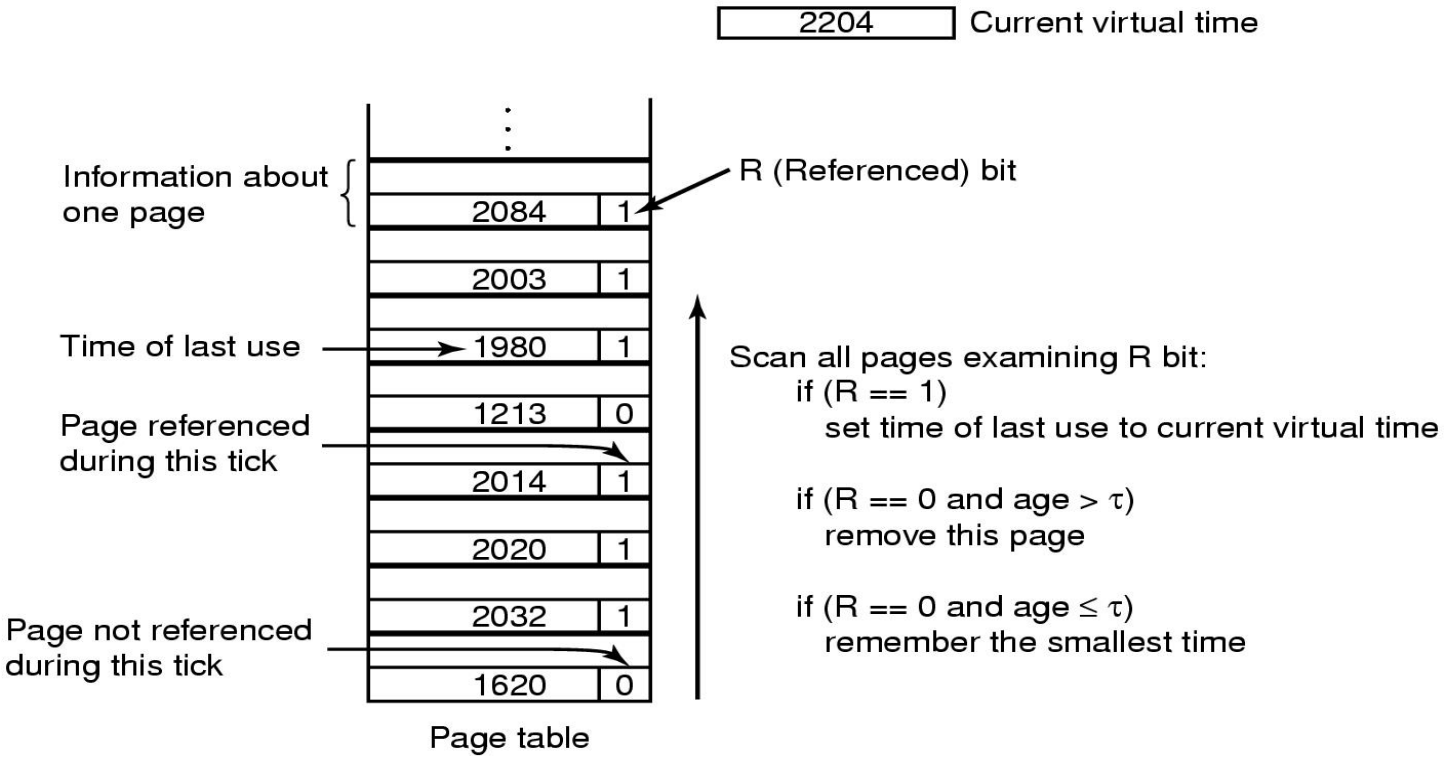
The aging algorithm simulates LRU in software. Shown are six page for five clock ticks. The five clock ticks are represented by (a) to (e).

Page Replacement Algorithms
Working Set Page Replacement (1)



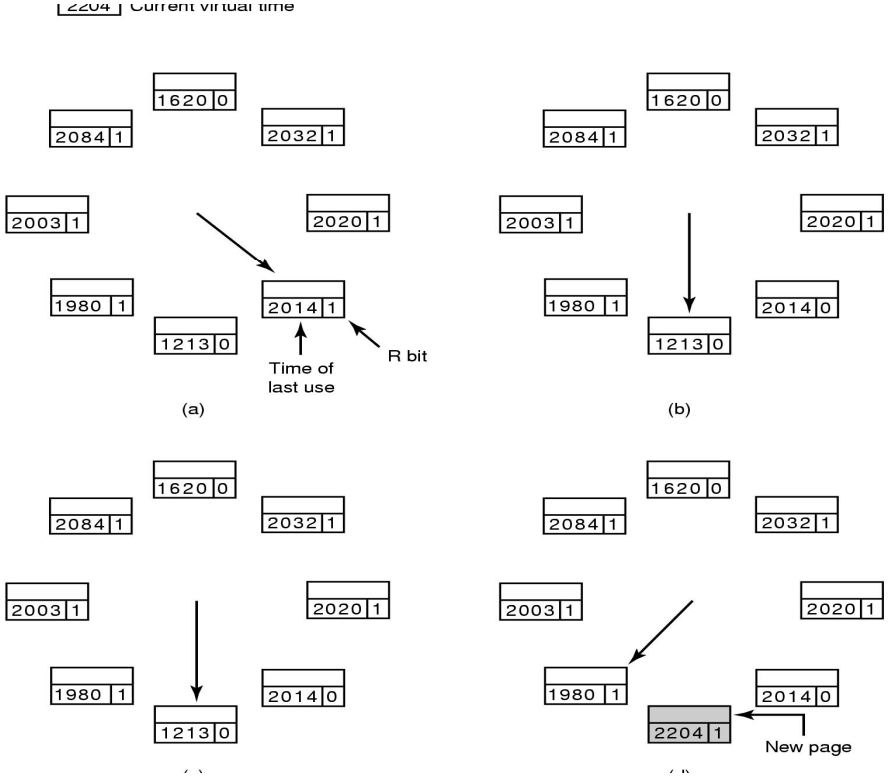
- The working set is the set of pages used by the k most recent memory references.
- The function w(k, t) is the size of the working set at time t.
- If Working set is in memory -> few page fault
- A program causing page faults every few instruction is said to be **Thrashing**

Page Replacement Algorithms
Working Set Page Replacement (2)



The working set algorithm.

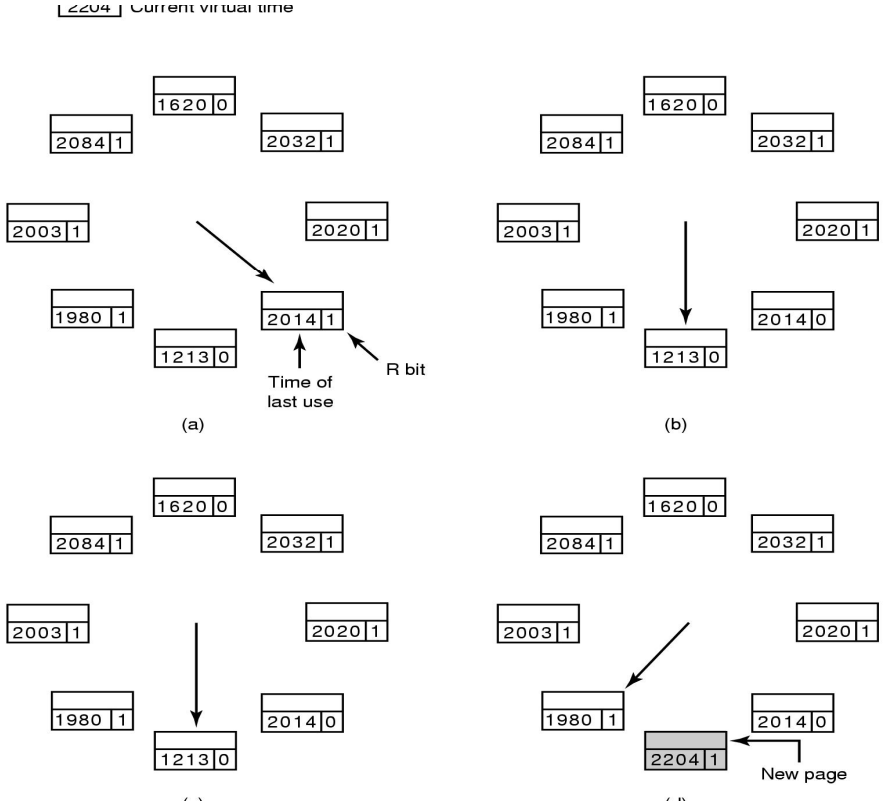
Page Replacement Algorithms
The WSClock Page Replacement Algorithm (3)



Operation of the WSClock algorithm. (a) and (b) give an example of what happens when R = 1.

Page Replacement Algorithms

The WSClock Page Replacement Algorithm (4)



Operation of the WSClock algorithm.
(c) and (d) give an example of R = 0.

Page Replacement Algorithms

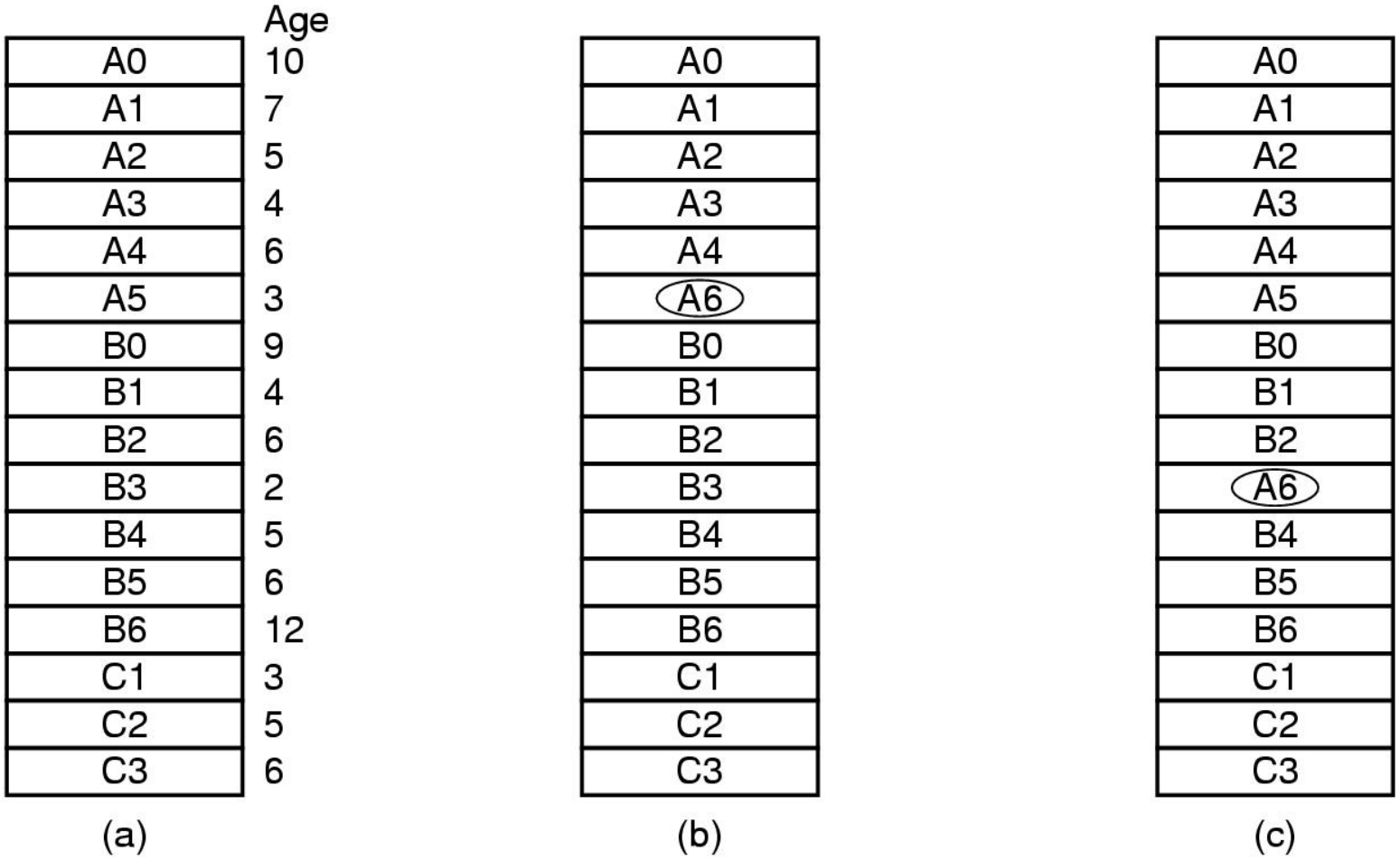
Summary

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	Somewhat expensive to implement
WSClock	Good efficient algorithm

3.5 Design Issues For Paging System

Design Issues For Paging System

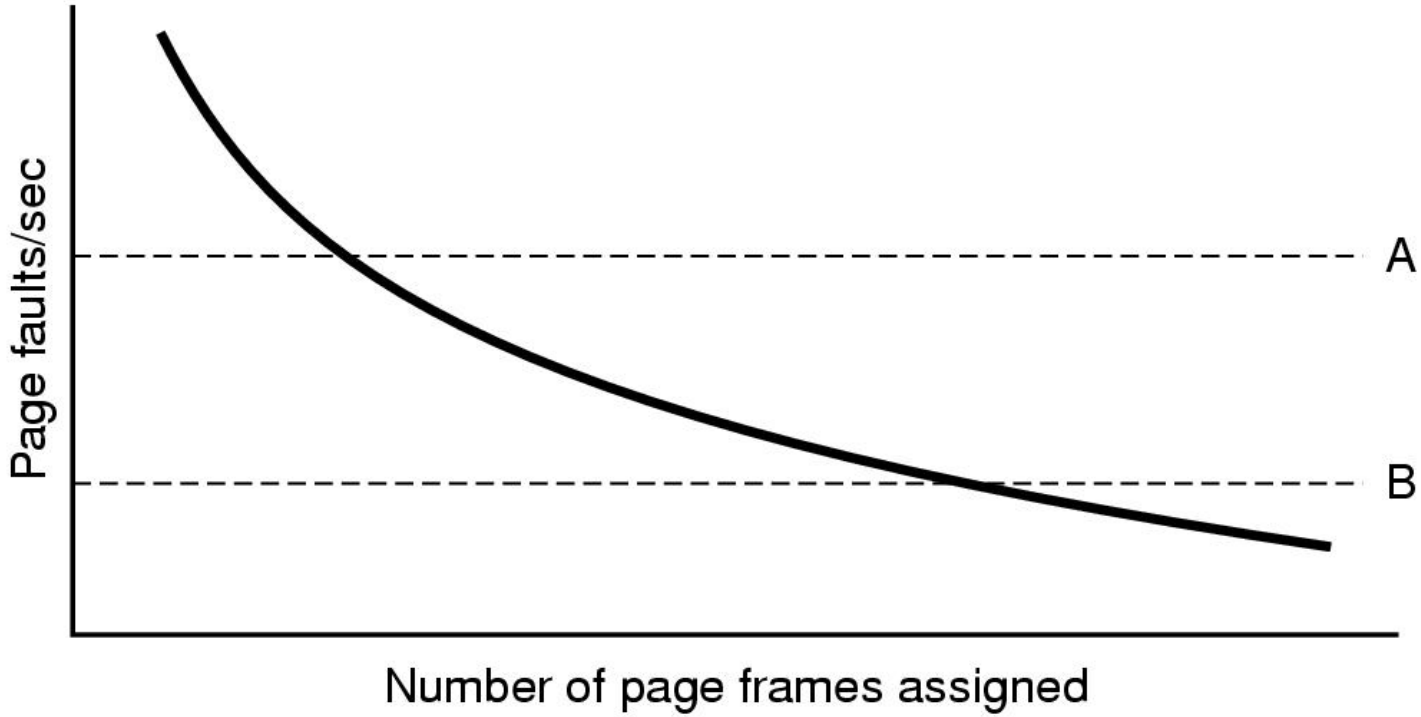
Local versus Global Allocation Policies (1)



(a) Original configuration. (b) Local page replacement. (c) Global page replacement.

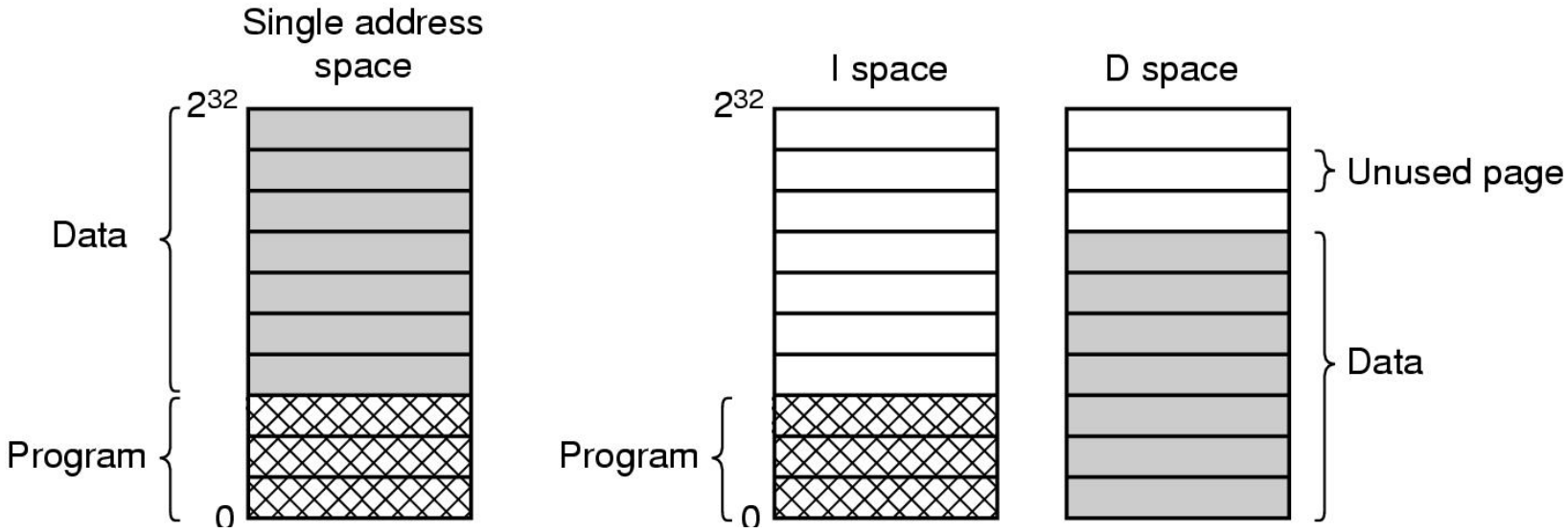
Design Issues For Paging System

Local versus Global Allocation Policies (2)



Page fault rate as a function
of the number of page frames assigned.

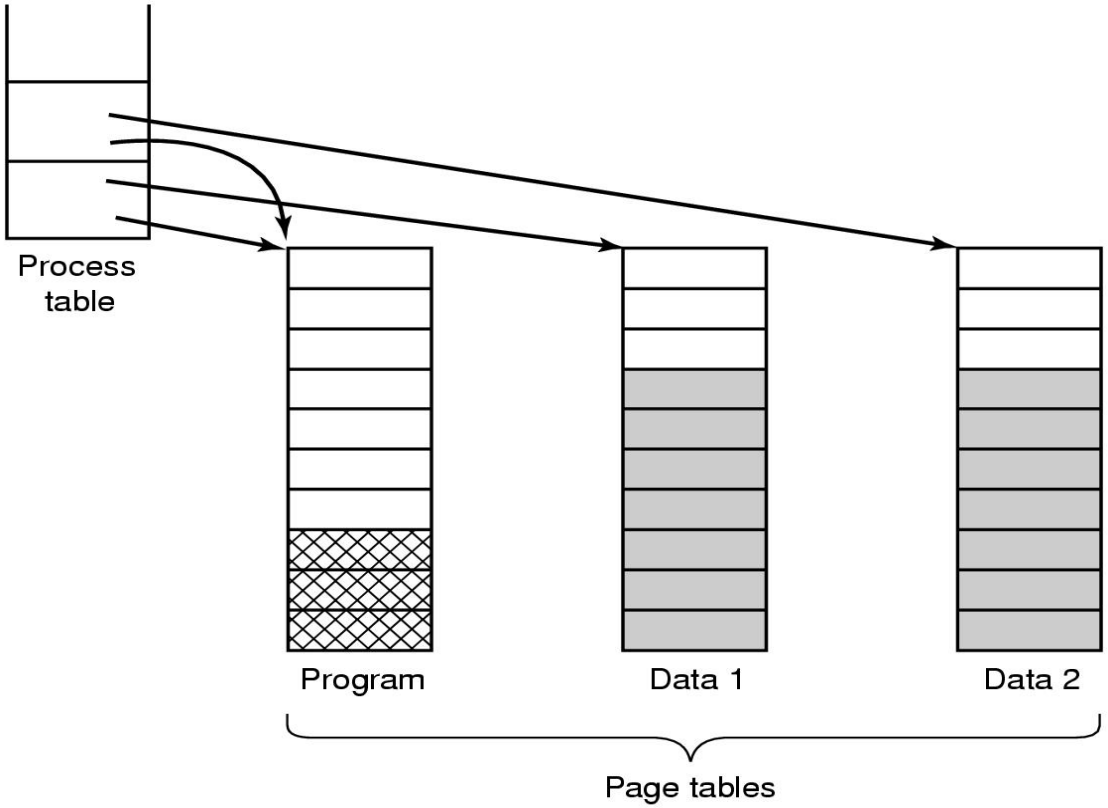
Design Issues For Paging System
Separate Instruction and Data Spaces



(a) One address space.
(b) Separate I and D spaces.

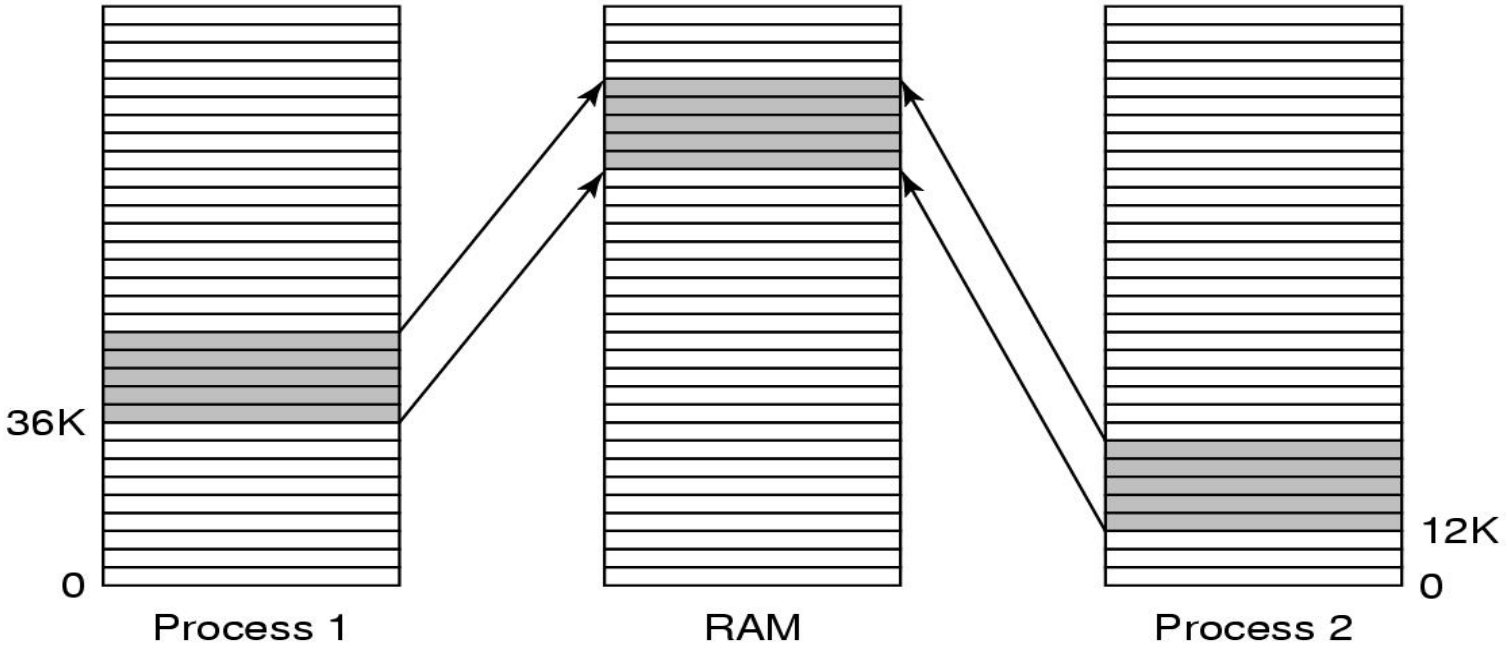
Design Issues For Paging System

Shared Pages



- Two processes sharing the same program sharing its page table.
- A problem occurs with sharing pages, when one process is terminated

Design Issues For Paging System
Shared Libraries



- A shared library being used by two processes.
- Avoiding absolute addresses: code uses relative offsets is called **Position-independent code**

3.6 Implementation issues

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

Implementation Issues
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

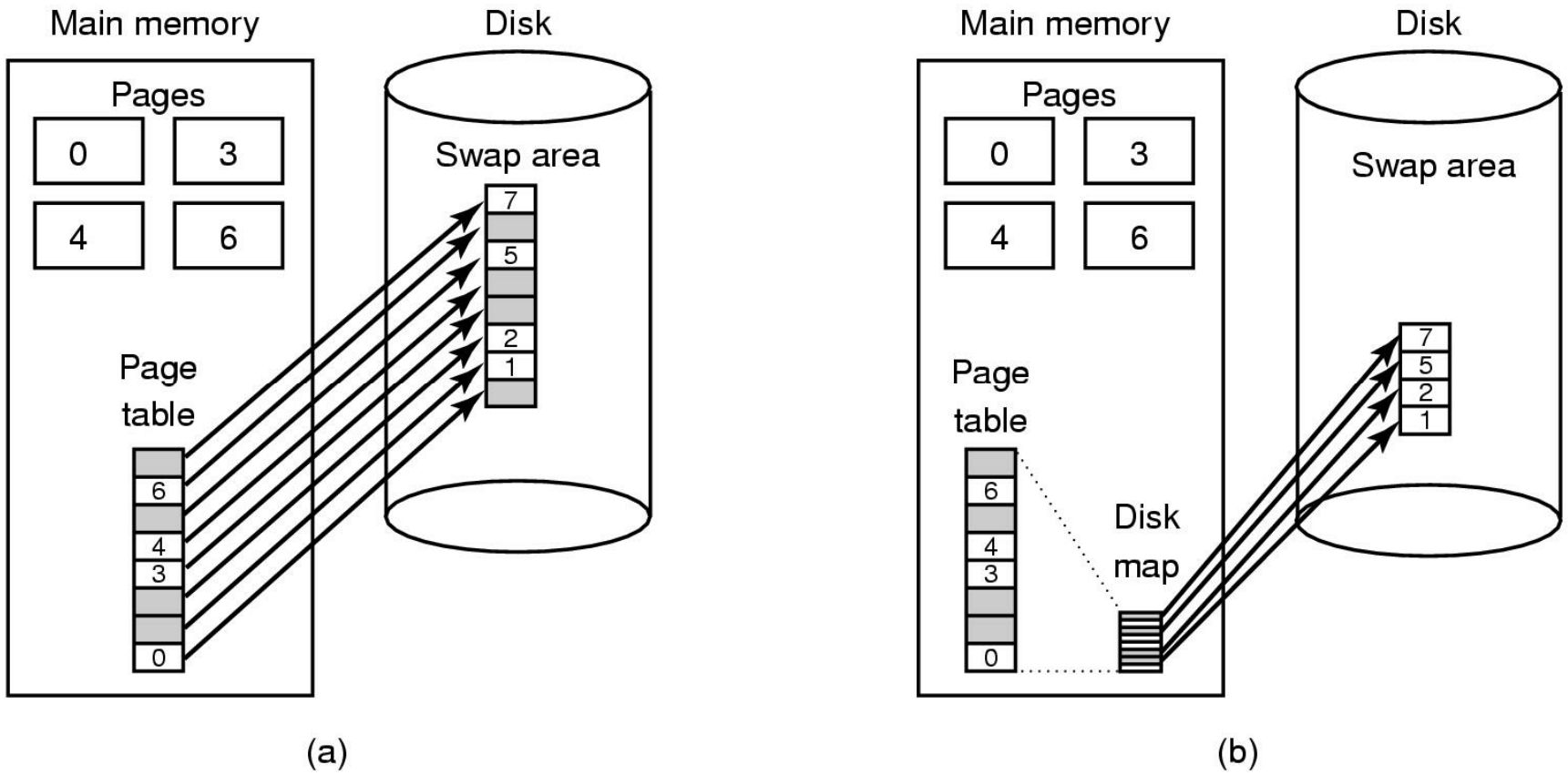
Implementation Issues
Page Fault Handling (2)

6. OS brings schedules new page in from disk
7. Page tables updated
8. Faulting instruction backed up to when it began
9. Faulting process scheduled
10. Registers restored, Program continues

Implementation Issues
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

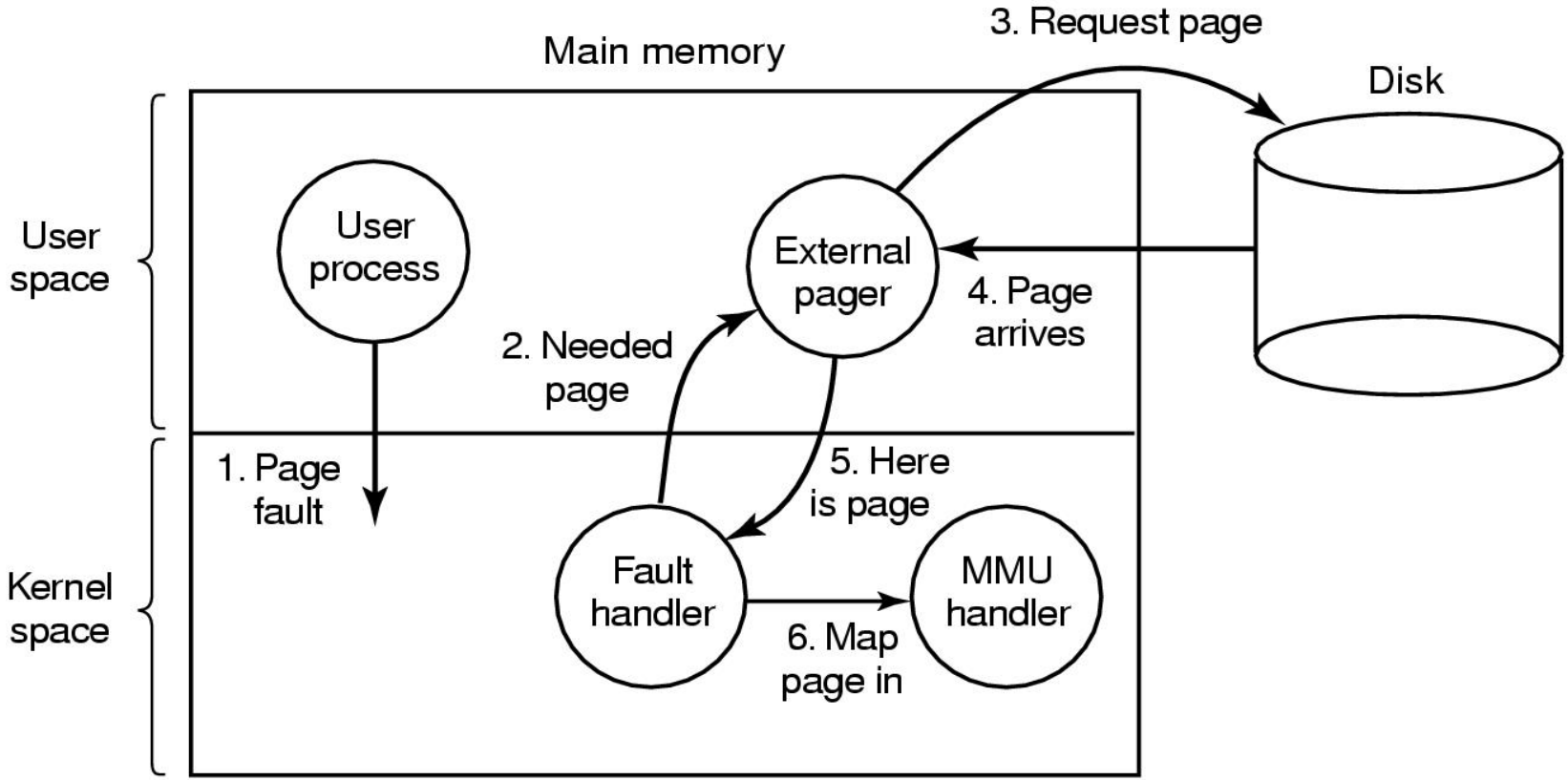
Implementation Issues
Backing Store



(a) Paging to static swap area

(b) Backing up pages dynamically

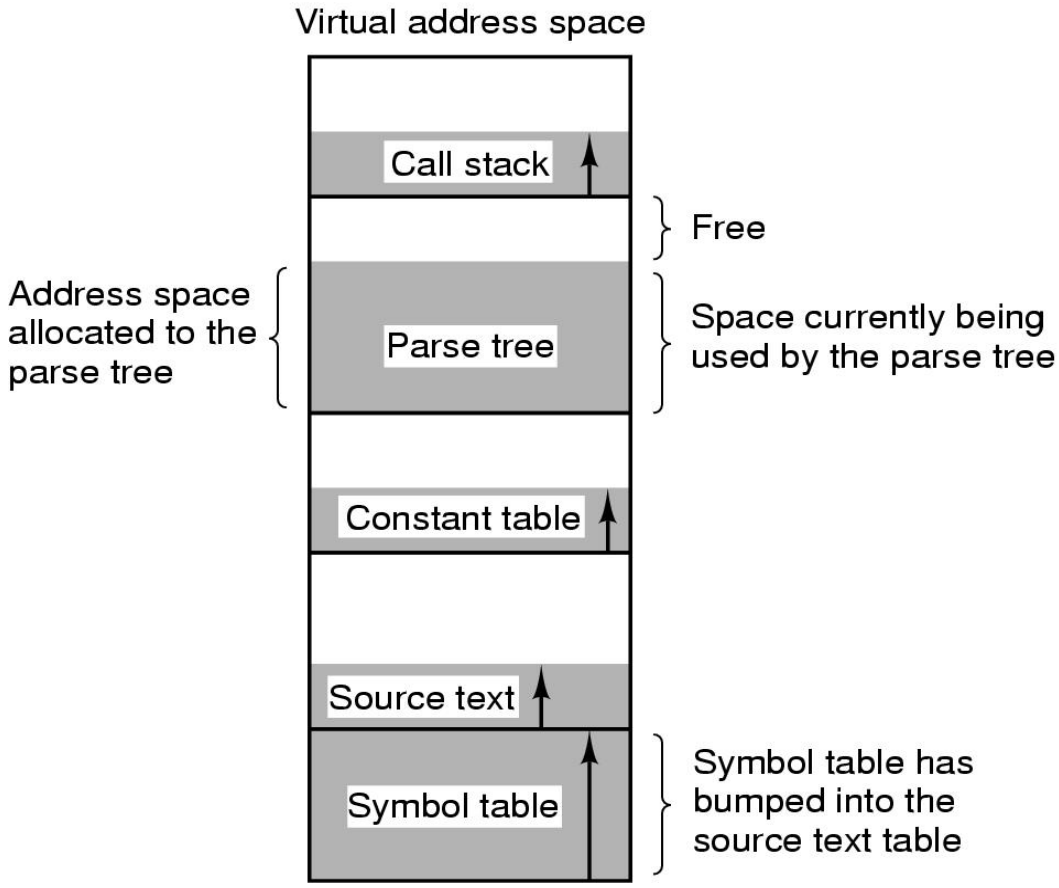
Implementation Issues
Separation of Policy and Mechanism



Page fault handling with an external pager

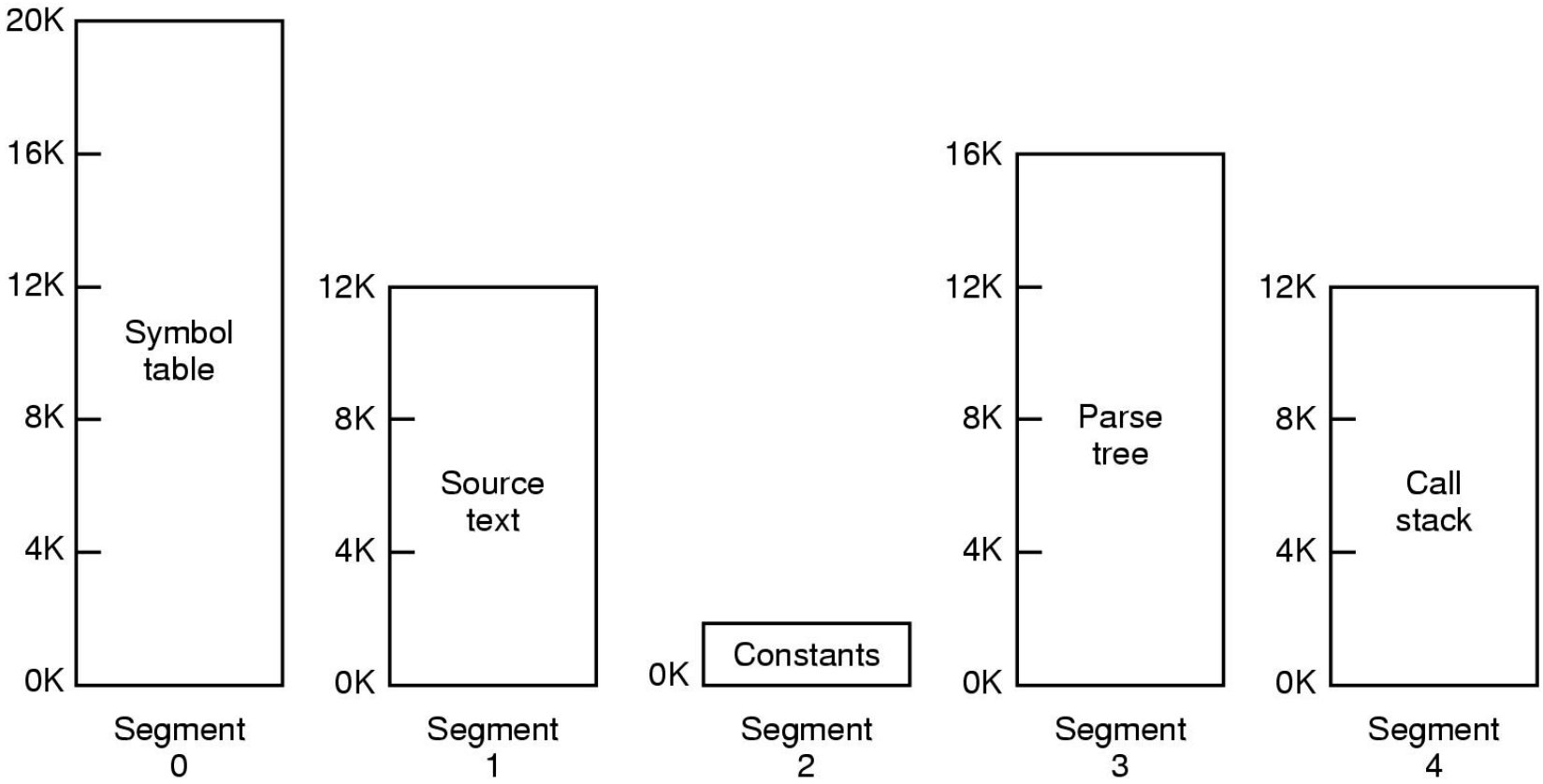
3.7 Virtual Memory Segmentation

Segmentation (1)



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

Segmentation (2)



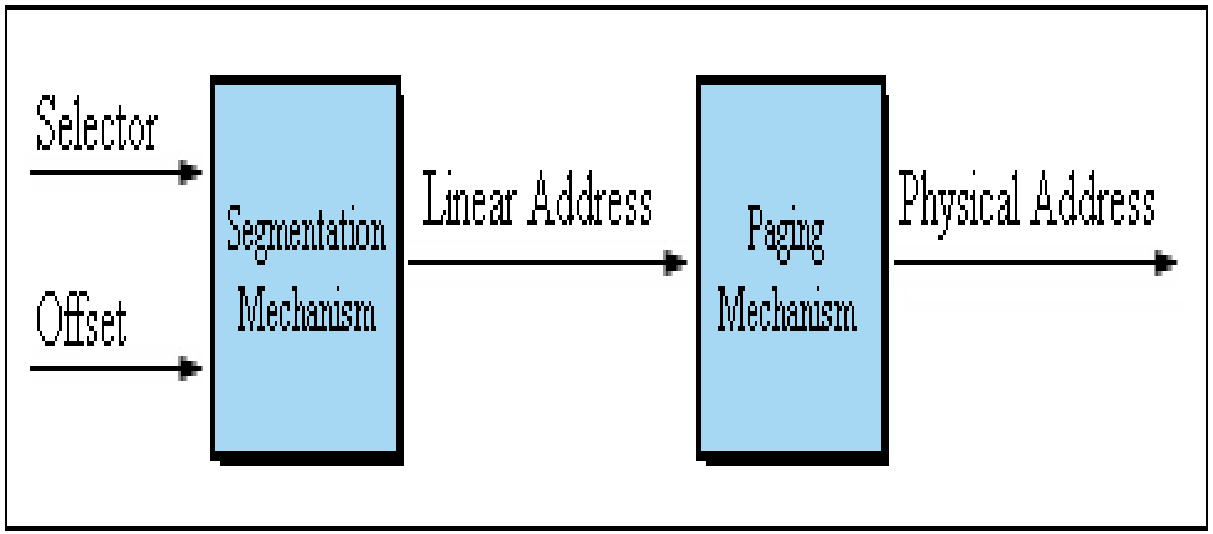
Allows each table to grow or shrink, independently

Segmentation (3)

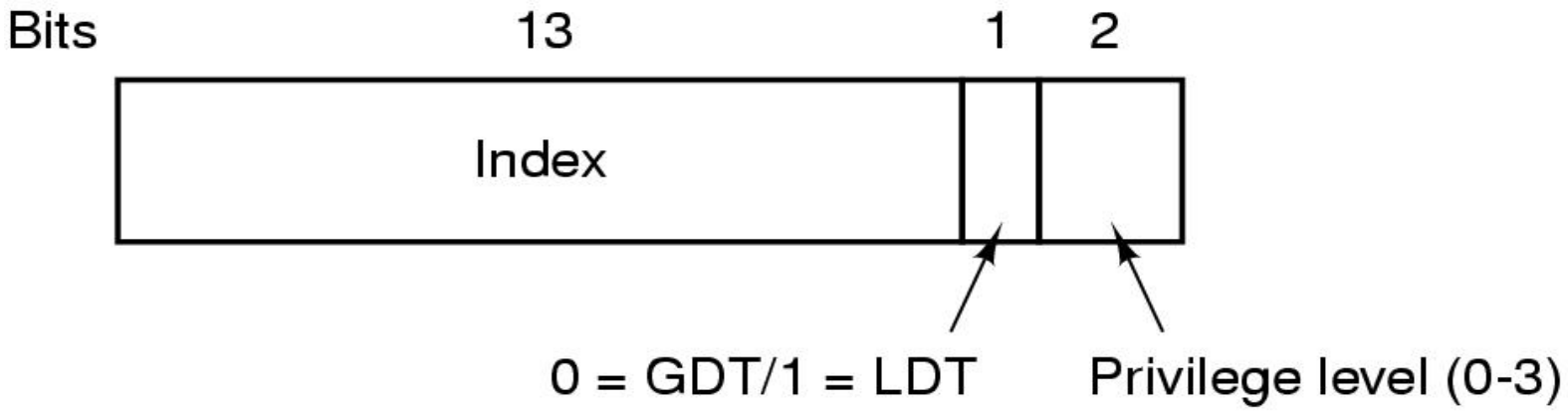
Consideration	Paging	Segmentation
Need the programmer be aware that this technique is being used?	No	Yes
How many linear address spaces are there?	1	Many
Can the total address space exceed the size of physical memory?	Yes	Yes
Can procedures and data be distinguished and separately protected?	No	Yes
Can 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

Segmentation with Paging: Pentium (1)



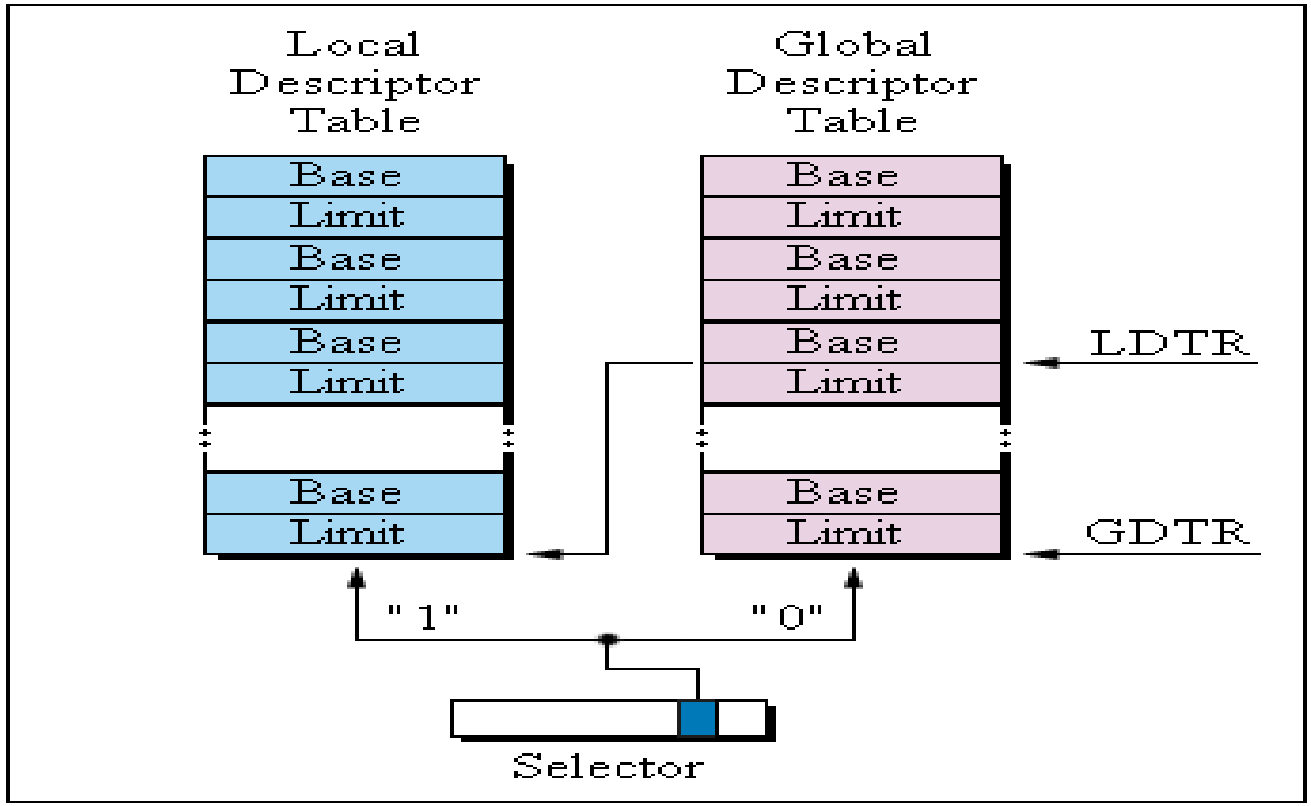
Segmentation with Paging: Pentium (2)



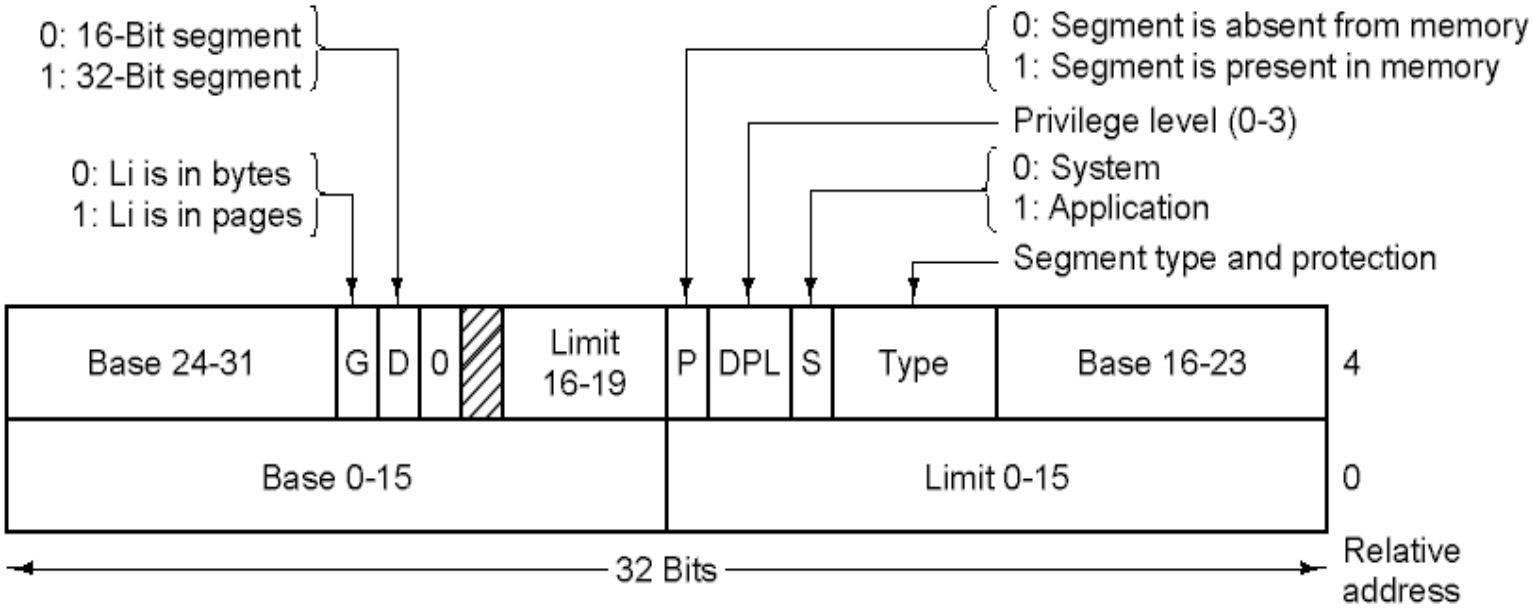
A Pentium selector

GDT (Global Descriptor Table), LDT (Local Descriptor Table)

Segmentation with Paging: Pentium (3)

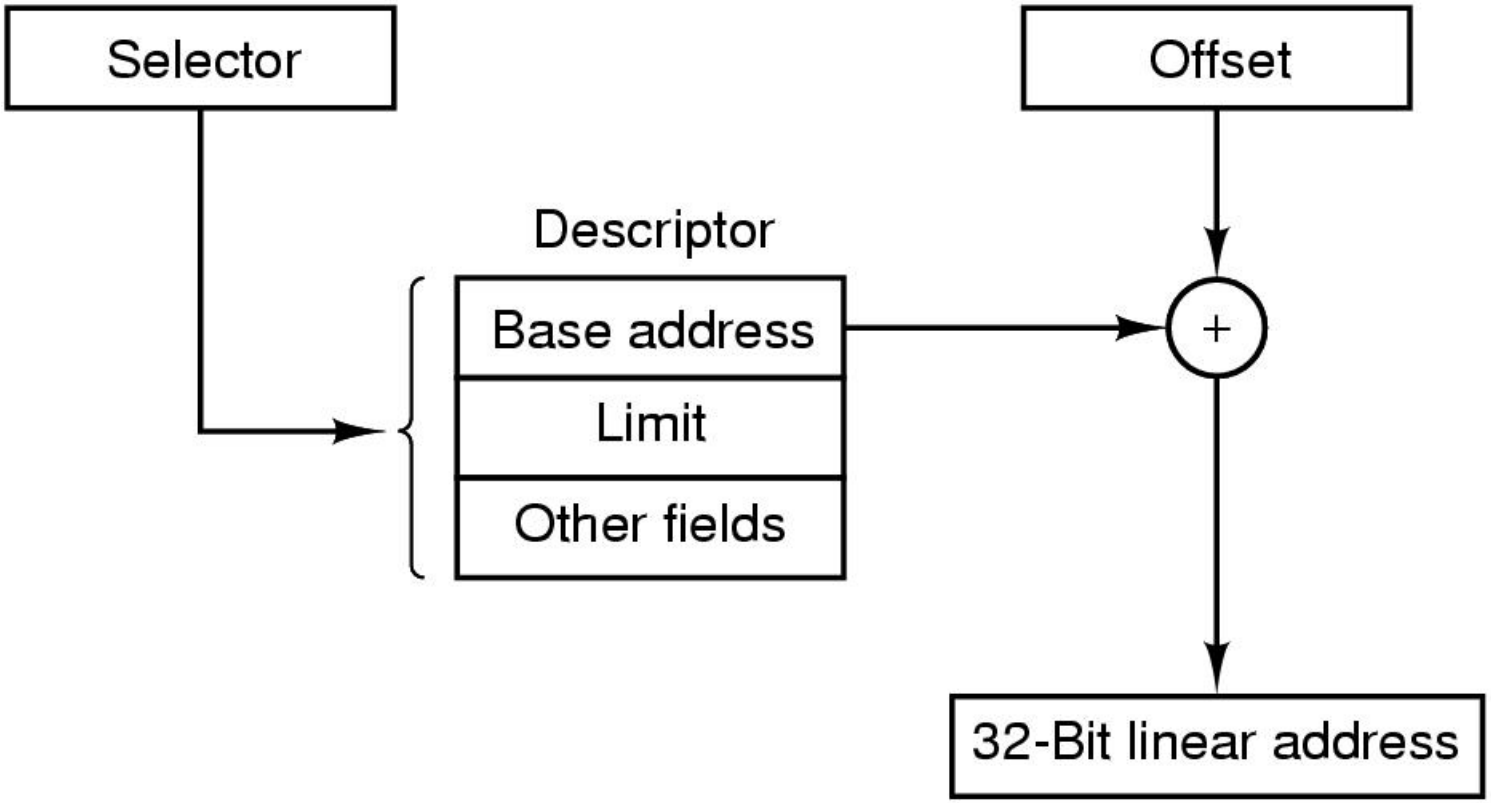


Segmentation with Paging: Pentium (4)



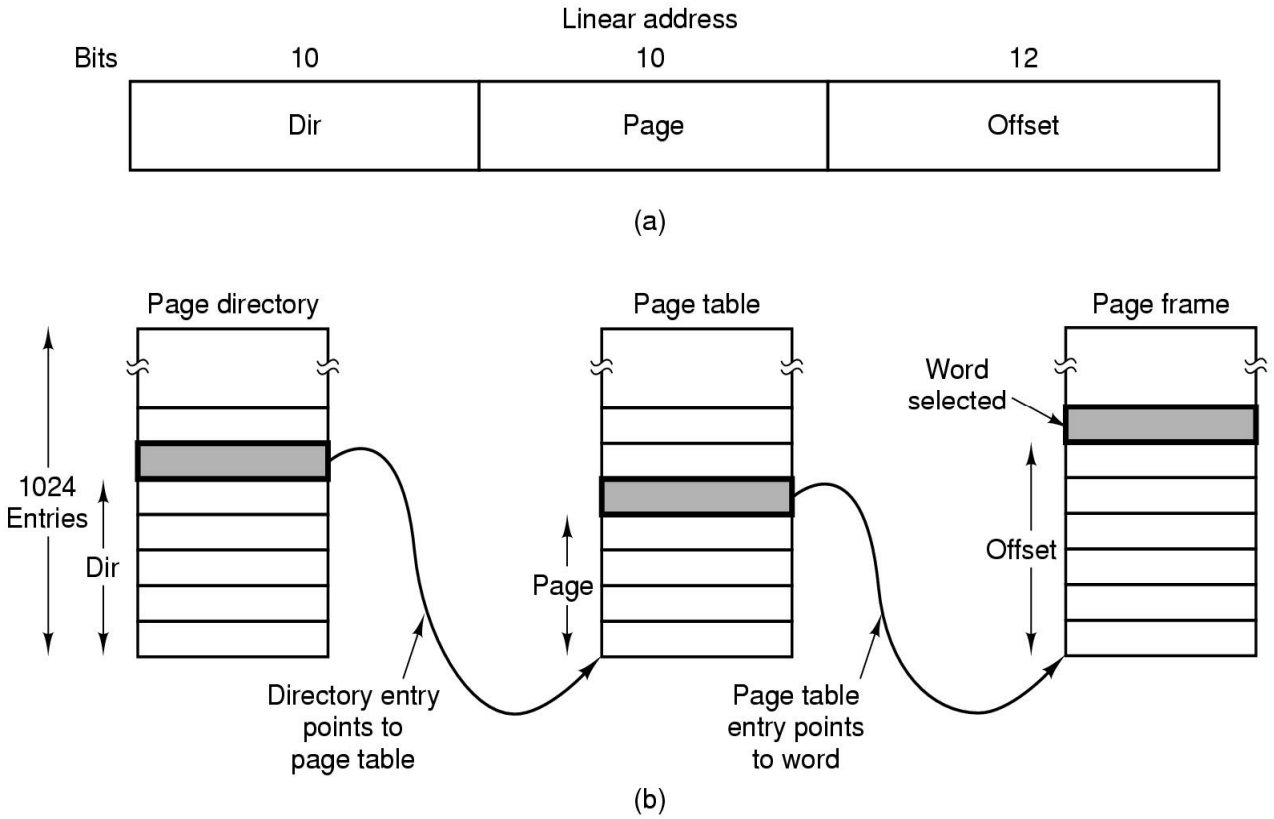
- Pentium code segment descriptor
- Data segments differ slightly

Segmentation with Paging: Pentium (5)



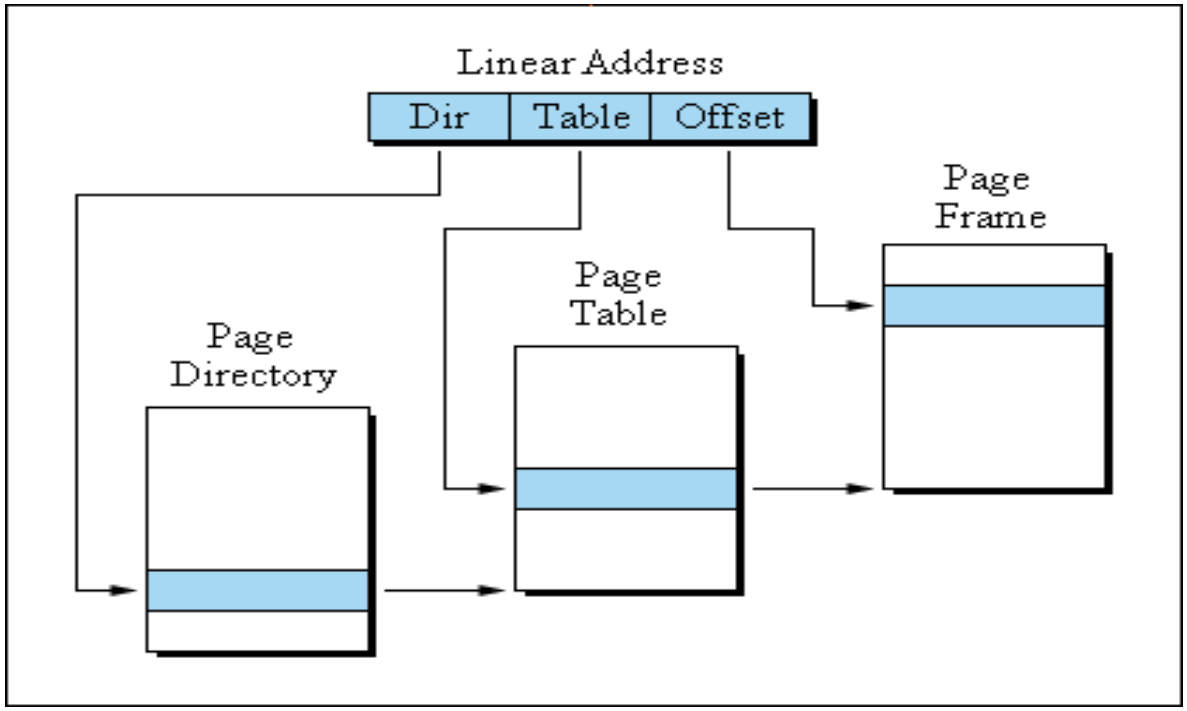
Conversion of a (selector, offset) pair to a linear address

Segmentation with Paging: Pentium (6)

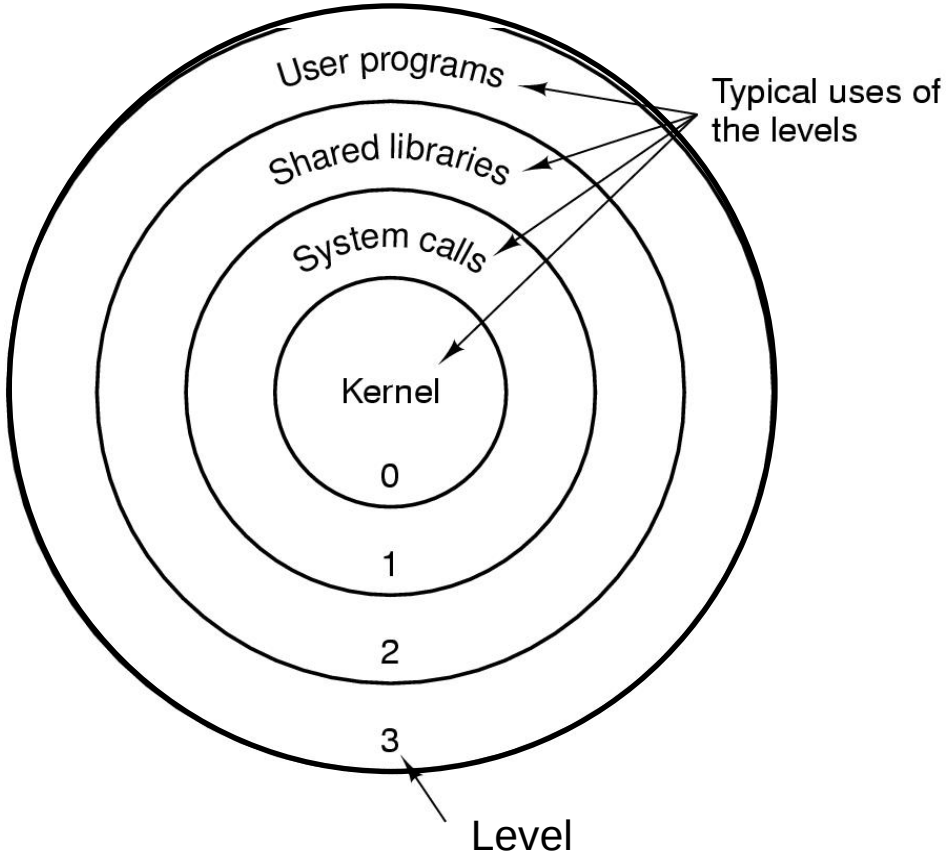


Mapping of a linear address onto a physical address

Segmentation with Paging: Pentium (7)



Segmentation with Paging: Pentium (8)



Protection on the Pentium