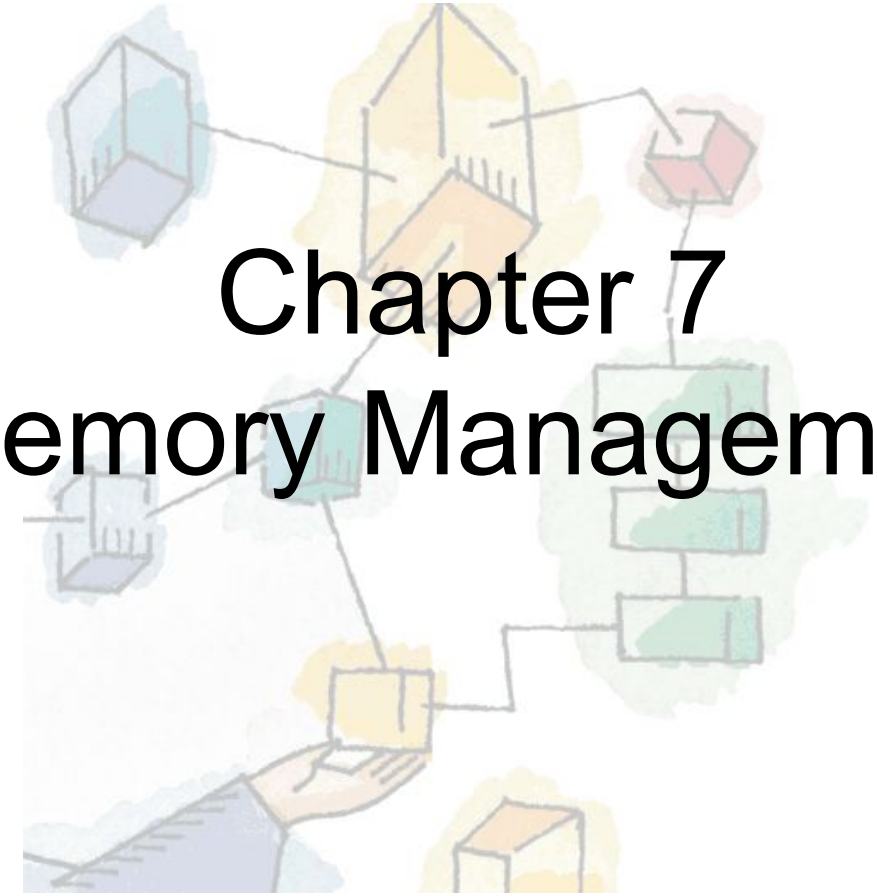
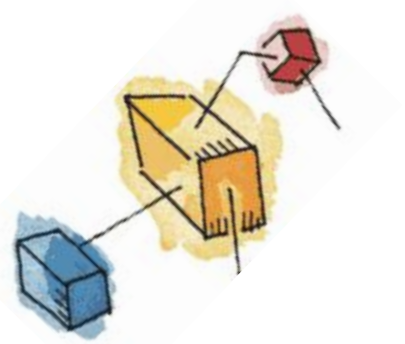
Operating Systems:
Internals and Design Principles, 6/E
William Stallings



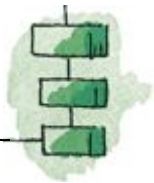
Chapter 7
Memory Management

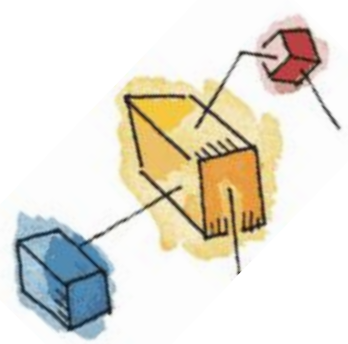
Patricia Roy
Manatee Community College, Venice, FL



Roadmap

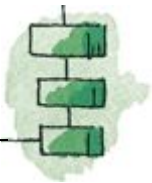
- Basic requirements of Memory Management
- Memory Partitioning
- Basic blocks of memory management
  - Paging
  - Segmentation





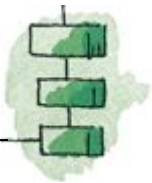
The need for memory management

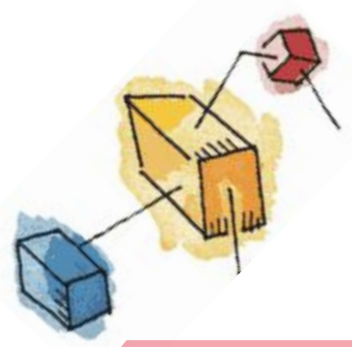
- Memory is cheap today, and getting cheaper
  - But applications are demanding more and more memory, there is never enough!
- Memory Management, involves swapping blocks of data from secondary storage.
- Memory I/O is slow compared to a CPU
  - The OS must cleverly time the swapping to maximise the CPU's efficiency



Memory Management

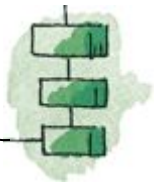
Memory needs to be allocated to ensure a reasonable supply of ready processes to consume available processor time

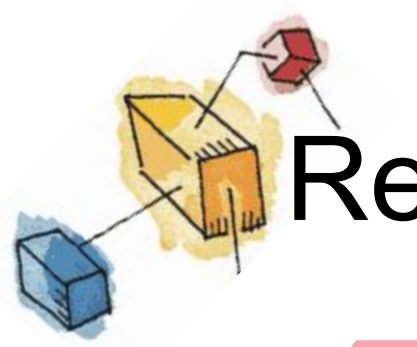




Memory Management Requirements

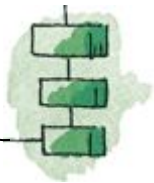
- Relocation
- Protection
- Sharing
- Logical organisation
- Physical organisation

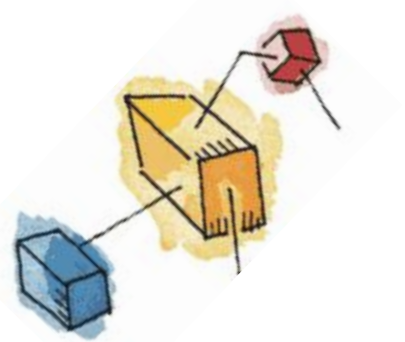




Requirements: Relocation

- The programmer does not know where the program will be placed in memory when it is executed,
  - it may be swapped to disk and return to main memory at a different location (relocated)
- Memory references must be translated to the actual physical memory address

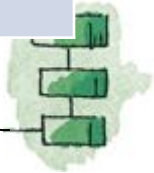




Memory Management Terms

Table 7.1 Memory Management Terms

Term	Description
Frame	***Fixed***-length block of main memory.
Page	***Fixed***-length block of data in secondary memory (e.g. on disk).
Segment	***Variable-length*** block of data that resides in secondary memory.



Addressing

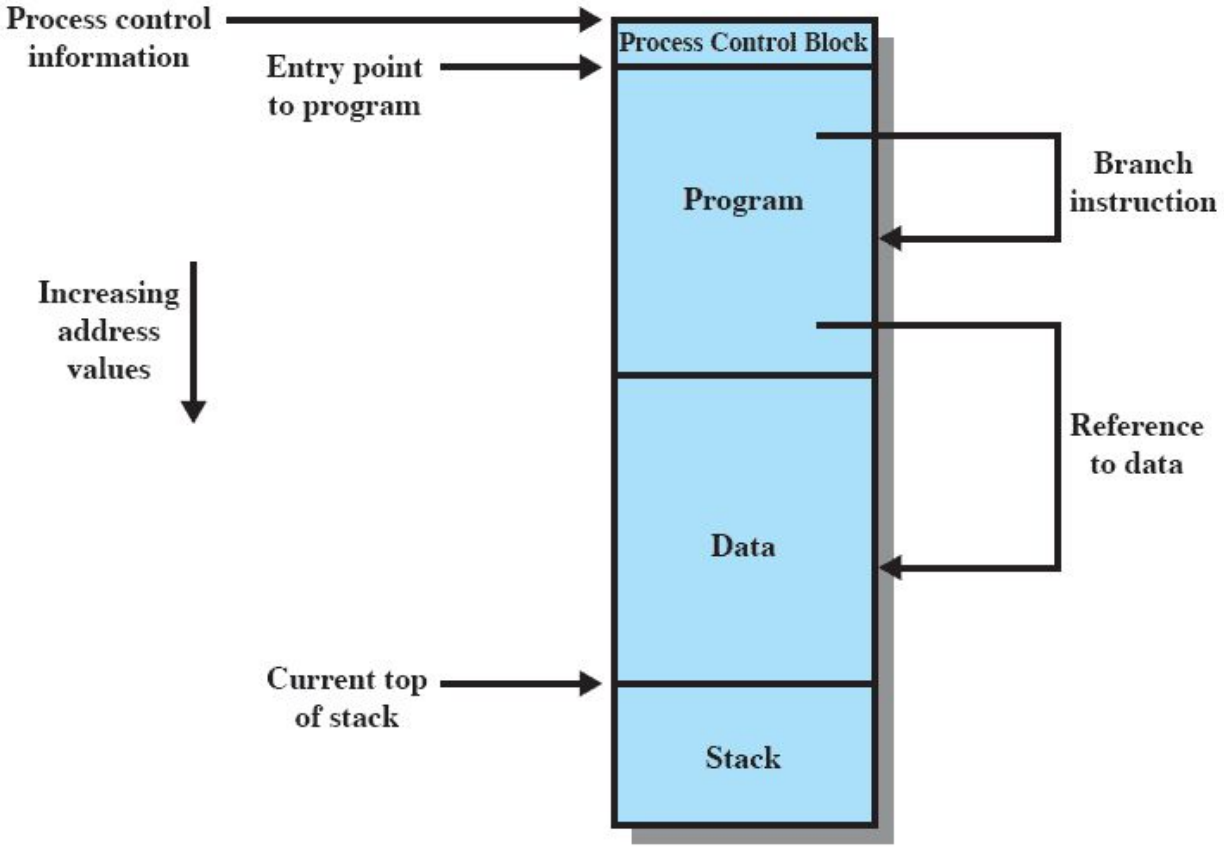
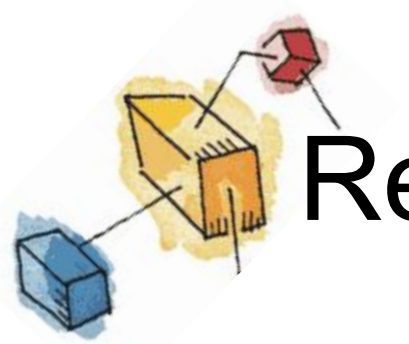
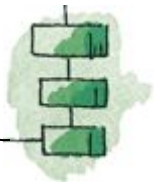


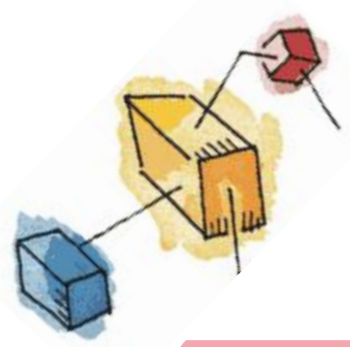
Figure 7.1 Addressing Requirements for a Process



Requirements: Protection

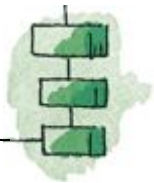
- Processes should not be able to reference memory locations in another process without permission
- Impossible to check absolute addresses at compile time
- Must be checked at run time

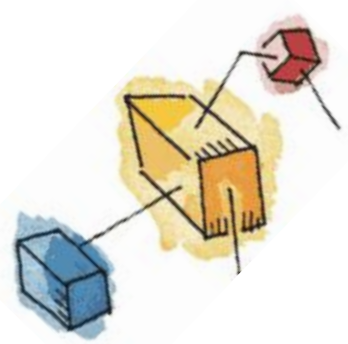




Requirements: Sharing

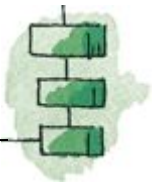
- Allow several processes to access the same portion of memory
- Better to allow each process access to the same copy of the program rather than have their own separate copy

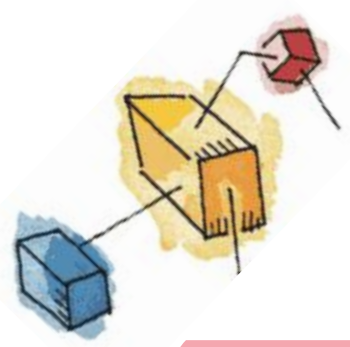




Requirements: Logical Organization

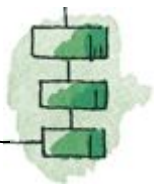
- Memory is organized linearly (usually)
- Programs are written in modules
  - Modules can be written and compiled independently
- Different degrees of protection given to modules (read-only, execute-only)
- Share modules among processes
- Segmentation helps here





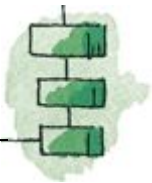
Requirements: Physical Organization

- Cannot leave the programmer with the responsibility to manage memory
- Memory available for a program plus its data may be insufficient
  - Overlaying allows various modules to be assigned the same region of memory but is time consuming to program
- Programmer does not know how much space will be available



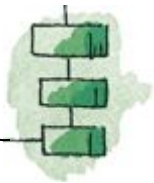
Partitioning

- An early method of managing memory
  - Pre-virtual memory
  - Not used much now
- But, it will clarify the later discussion of virtual memory if we look first at partitioning
  - Virtual Memory has evolved from the partitioning methods



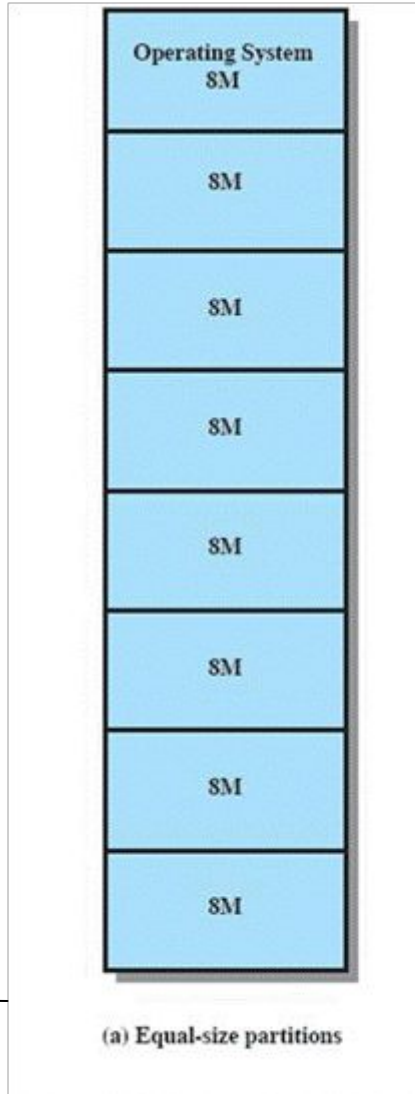
Types of Partitioning

- Fixed Partitioning
- Dynamic Partitioning
- Simple Paging
- Simple Segmentation
- Virtual Memory Paging
- Virtual Memory Segmentation

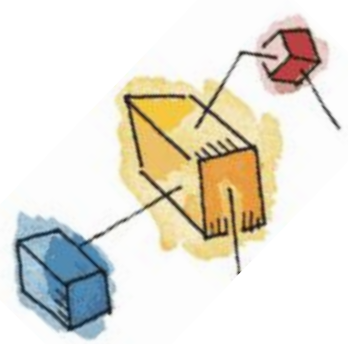


Fixed Partitioning

- Equal-size partitions (see fig 7.3a)
  - Any process whose size is less than or equal to the partition size can be loaded into an available partition
- The operating system can swap a process out of a partition
  - If none are in a ready or running state

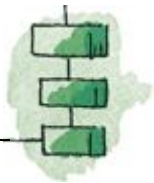


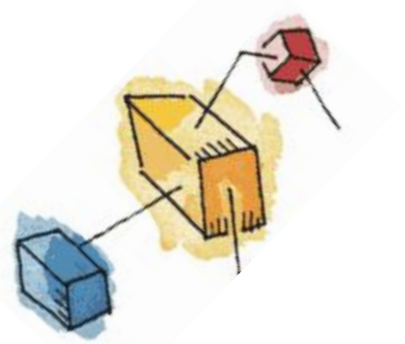
(a) Equal-size partitions



Fixed Partitioning Problems

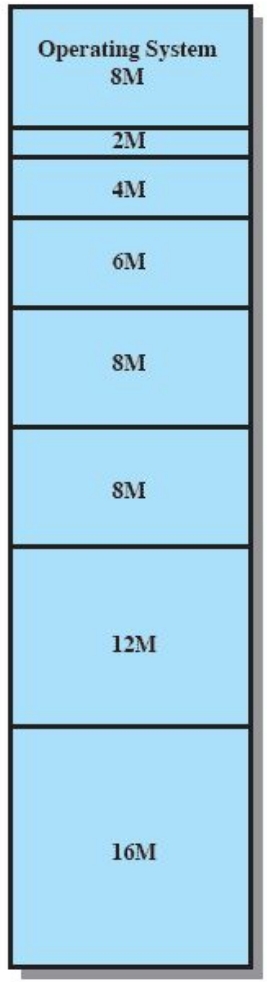
- A program may not fit in a partition.
  - The programmer must design the program with overlays
- Main memory use is inefficient.
  - Any program, no matter how small, occupies an entire partition.
  - This is results in ***internal fragmentation.***





Solution – Unequal Size Partitions

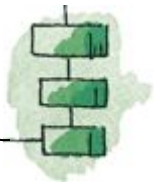
- Lessens both problems
  - but doesn't solve completely
- In Fig 7.3b,
  - Programs up to 16M can be accommodated without overlay
  - Smaller programs can be placed in smaller partitions, reducing internal fragmentation



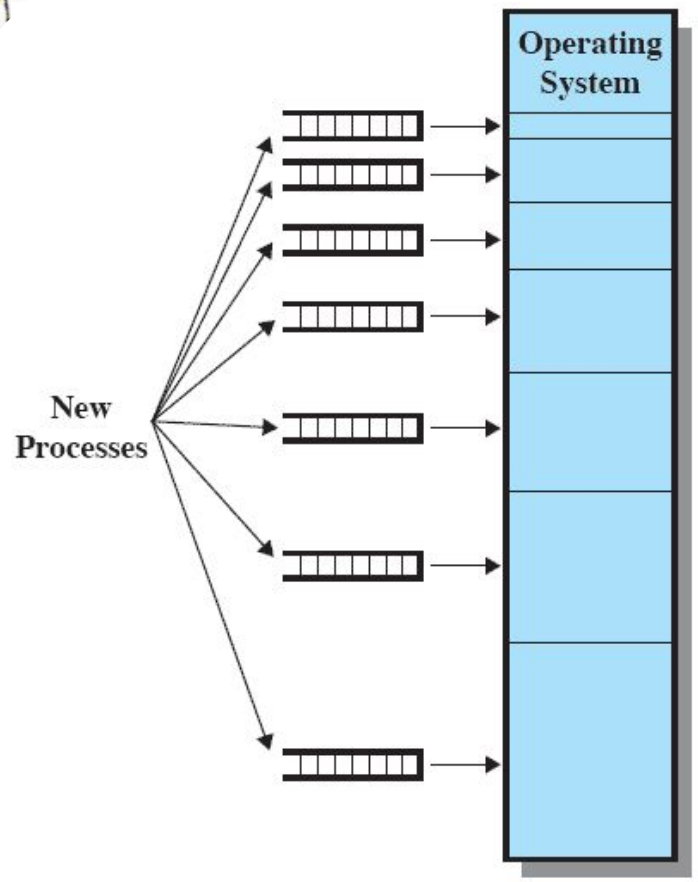
(b) Unequal-size partitions

Placement Algorithm

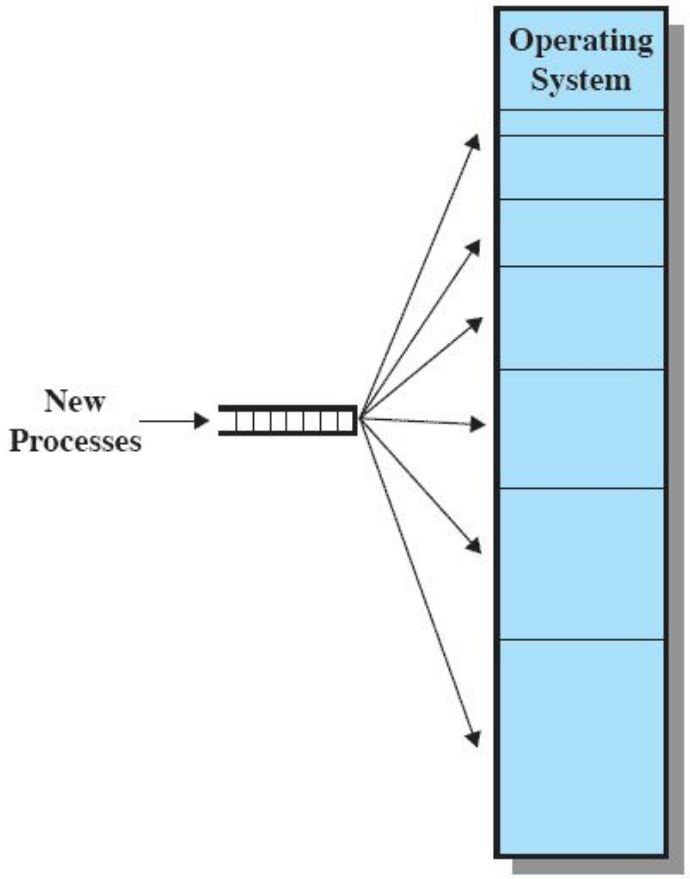
- Equal-size
  - Placement is trivial (no options)
- Unequal-size
  - Can assign each process to the smallest partition within which it will fit
  - Queue for each partition
  - Processes are assigned in such a way as to minimize wasted memory within a partition



Fixed Partitioning

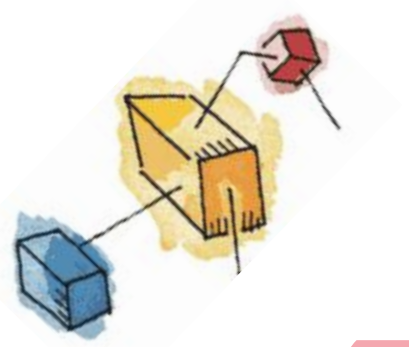


(a) One process queue per partition



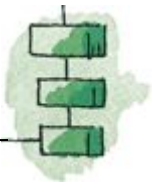
(b) Single queue

Figure 7.3 Memory Assignment for Fixed Partitioning



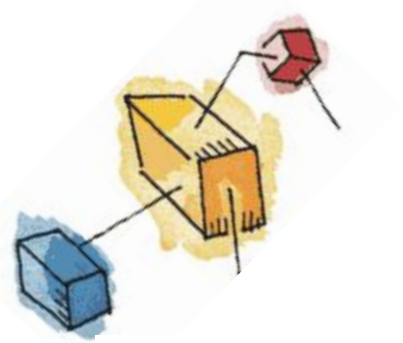
Remaining Problems with Fixed Partitions

- The number of active processes is limited by the system
  - I.E limited by the pre-determined number of partitions
- A large number of very small process will not use the space efficiently
  - In either fixed or variable length partition methods

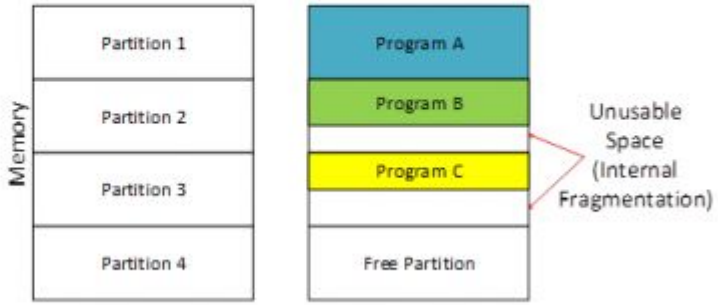


Difference Between Internal and External fragmentation

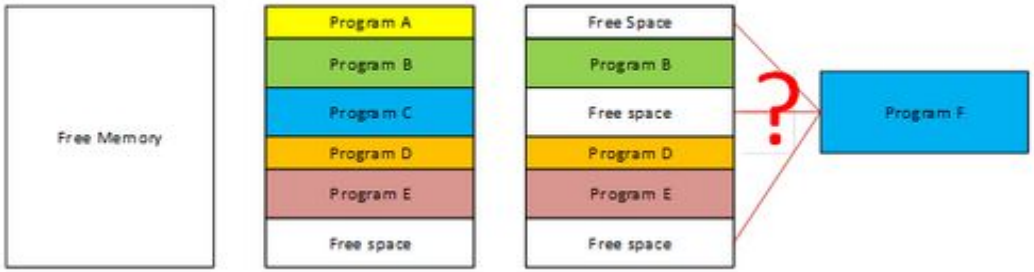
- Whenever a process is loaded or removed from the physical memory block, it creates a small hole in memory space which is called fragment.
- Due to fragmentation, the system fails in allocating the contiguous memory space to a process even though it have the requested amount of memory but, in a non-contiguous manner.
- The fragmentation is further classified into two categories Internal and External Fragmentation.

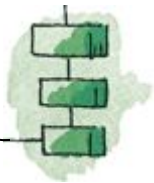


What is Internal Fragmentation?

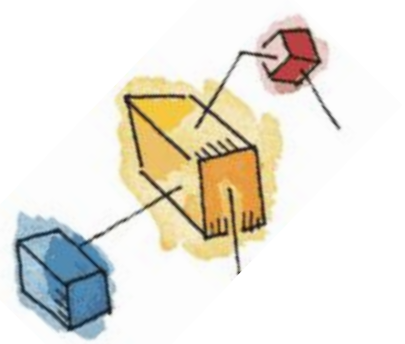


What is External Fragmentation?



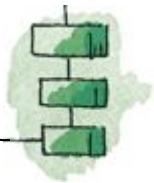


Internal fragmentation	External fragmentation
It occurs when fixed sized memory blocks are allocated to the processes.	It occurs when variable size memory space are allocated to the processes dynamically.
When the memory assigned to the process is slightly larger than the emory requested by the process this creates free space in the allocated block causing internal fragmentation.	When the process is removed from the memory, it creates the free space in the memory causing external fragmentation.
The memory must be partitioned into variable sized blocks and assign the best fit block to the process.	Compaction, paging and segmentation.
The problem of internal fragmentation can be reduced, but it can not be totally eliminated.	The paging and segmentation help in utilising the space freed due to external fragmentation by allowing a process to occupy the memory in a non-contiguous manner.
Internal fragmentation is the area occupied by a process but cannot be used by the process. This space is unusable by the system until the process release the space	External fragmentation exists when total free memory is enough for the new process but it's not contiguous and can't satisfy the request. Storage is fragmented into small holes.



Dynamic Partitioning

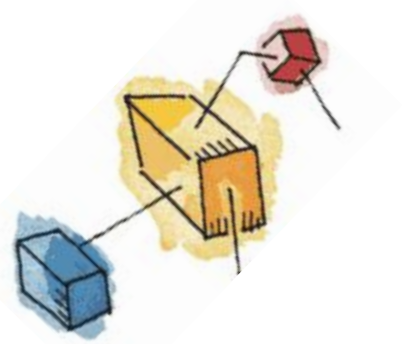
- Partitions are of variable length and number
- Process is allocated exactly as much memory as required



Dynamic Partitioning Example

- ***External Fragmentation***
- Memory external to all processes is fragmented
- Can resolve using ***compaction***
  - OS moves processes so that they are contiguous
  - Time consuming and wastes CPU time

Refer to Figure 7.4

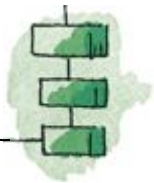


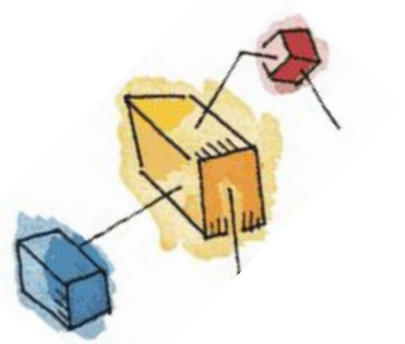
Dynamic Partitioning

- Operating system must decide which free block to allocate to a process
- Best-fit algorithm
  - Chooses the block that is closest in size to the request
  - Worst performer overall
  - Since smallest block is found for process, the smallest amount of fragmentation is left
  - Memory compaction must be done more often

Dynamic Partitioning

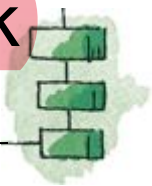
- First-fit algorithm
  - Scans memory form the beginning and chooses the first available block that is large enough
  - Fastest
  - May have many process loaded in the front end of memory that must be searched over when trying to find a free block





Dynamic Partitioning

- Next-fit
  - Scans memory from the location of the last placement
  - More often allocate a block of memory at the end of memory where the largest block is found
  - The largest block of memory is broken up into smaller blocks
  - Compaction is required to obtain a large block at the end of memory



Allocation

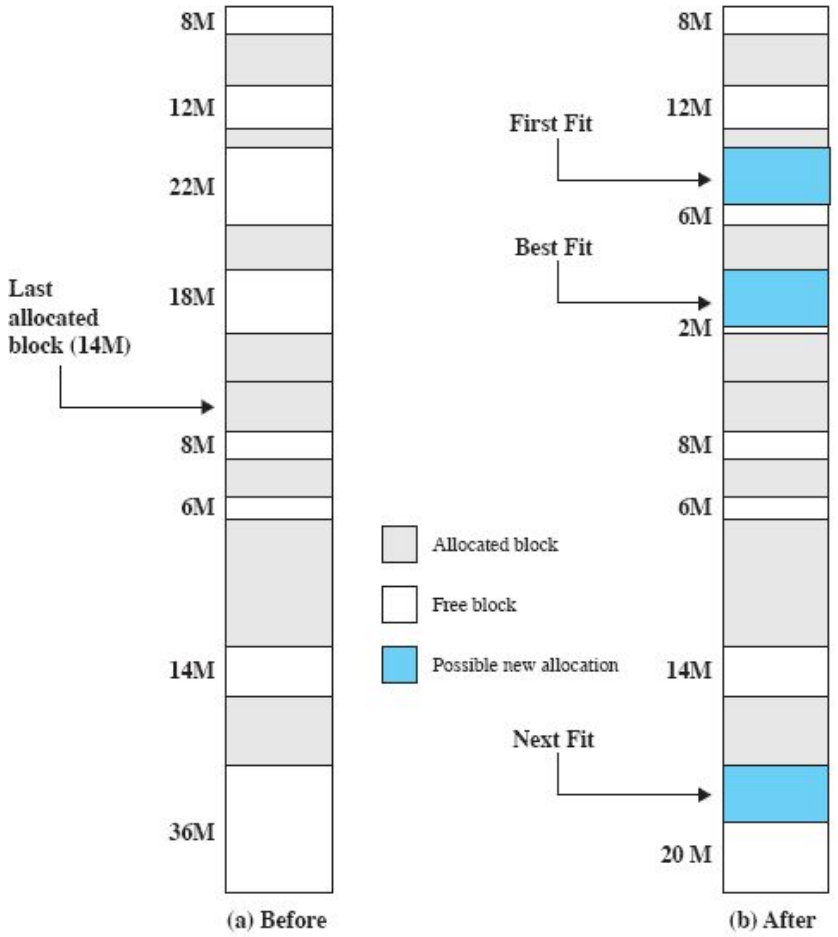
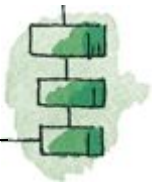


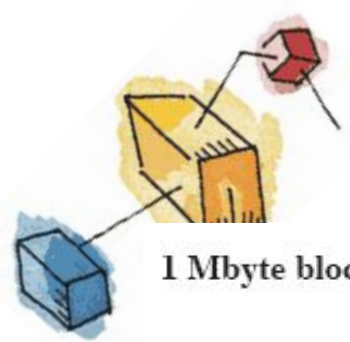
Figure 7.5 Example Memory Configuration before and after Allocation of 16-Mbyte Block

Buddy System

- Entire space available is treated as a single block of 2^U
- If a request of size *s* where $2^{U-1} < s <= 2^U$
  - entire block is allocated
- Otherwise block is split into two equal buddies
  - Process continues until smallest block greater than or equal to *s* is generated



Example of Buddy System



1 Mbyte block	1 M				
Request 100 K	A = 128K	128K	256K	512K	
Request 240 K	A = 128K	128K	B = 256K	512K	
Request 64 K	A = 128K	C = 64K	64K	B = 256K	512K
Request 256 K	A = 128K	C = 64K	64K	B = 256K	D = 256K	256K
Release B	A = 128K	C = 64K	64K	256K	D = 256K	256K
Release A	128K	C = 64K	64K	256K	D = 256K	256K
Request 75 K	E = 128K	C = 64K	64K	256K	D = 256K	256K
Release C	E = 128K	128K	256K	D = 256K	256K	
Release E	512K			D = 256K	256K	
Release D	1M				

Figure 7.6 Example of Buddy System

Tree Representation of Buddy System

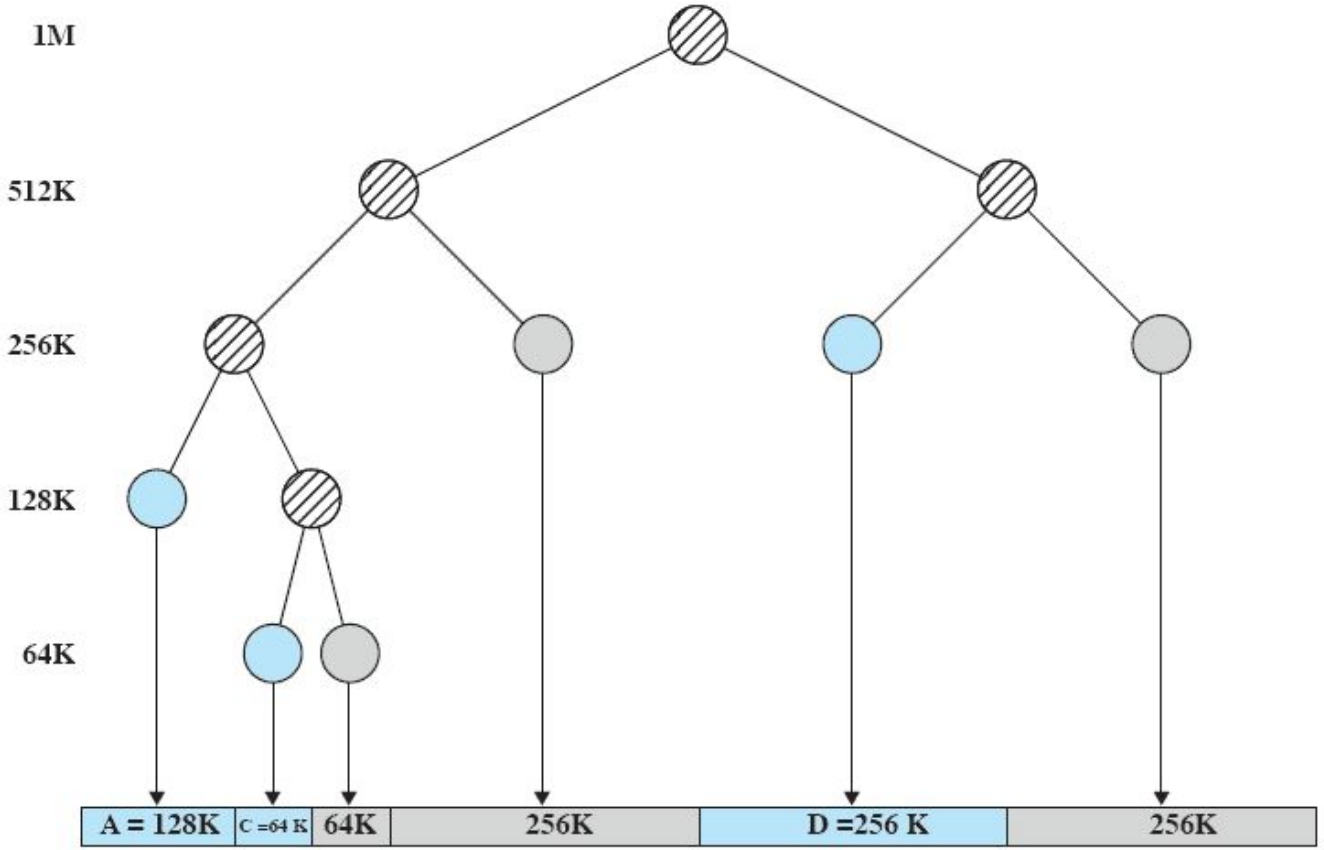
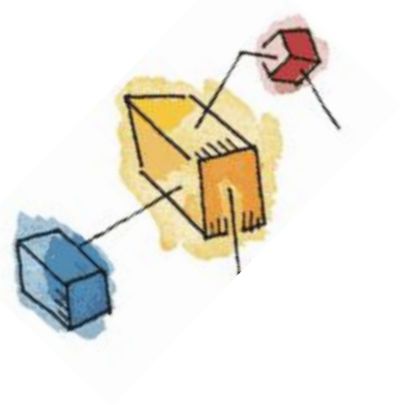
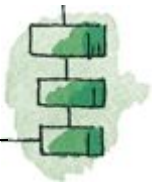
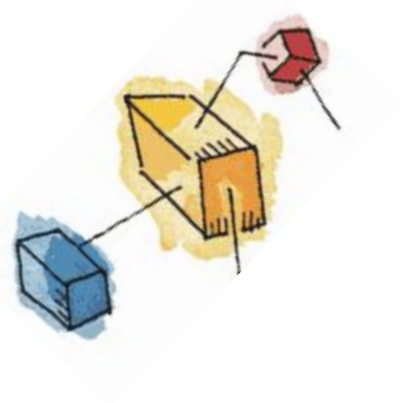


Figure 7.7 Tree Representation of Buddy System

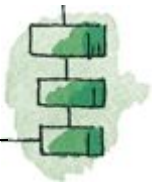


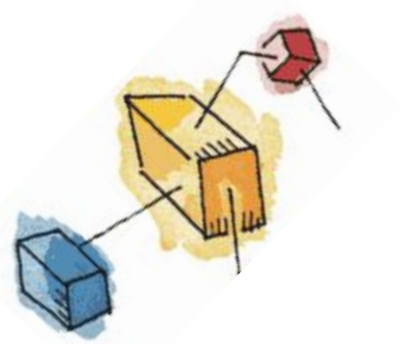
Given memory partitions of 100K, 500K, 200K, 300K, and 600K (in order), how would each of the First-fit, Best-fit, and Worst-fit algorithms place processes of 212K, 417K, 122K, and 426K (in order)? Which algorithm makes the most efficient use of memory?



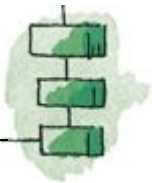


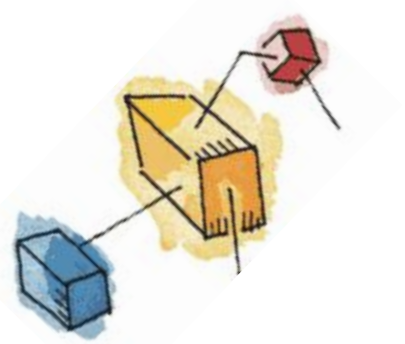
- First-fit:
  212k -> 500K (288 left)
  417k -> 600k (183 left)
  122k -> 288k (166k left)
  426k -> nowhere big enough left.



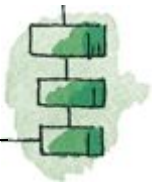


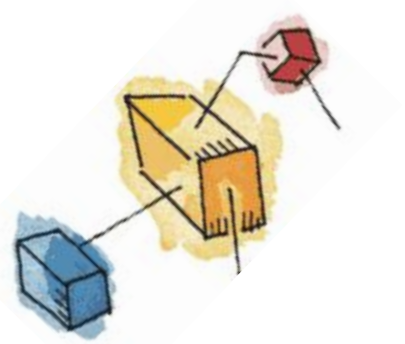
- Best-fit:
  212k -> 300k (88k left)
  417k -> 500k (83k left)
  122k -> 200k (78k left)
  426k -> 600k (174k left)





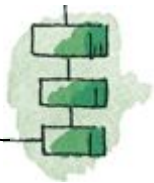
- Worst-fit:
  212k -> 600k (388k left)
  417k -> 500k (83k left)
  122k -> 388k (266k left)
  426k -> nowhere big enough again!





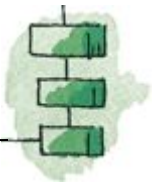
Relocation

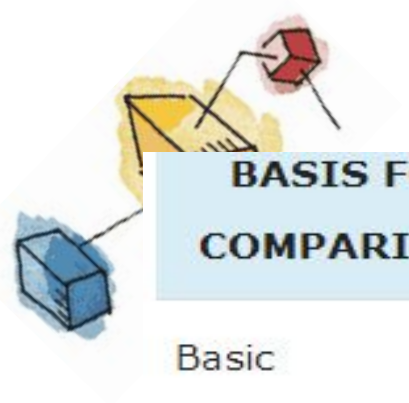
- When program loaded into memory the actual (absolute) memory locations are determined
- A process may occupy different partitions which means different absolute memory locations during execution
  - Swapping
  - Compaction



Addresses

- Logical
  - Reference to a memory location independent of the current assignment of data to memory.
- Relative
  - Address expressed as a location relative to some known point.
- Physical or Absolute
  - The absolute address or actual location in main memory.





BASIS FOR COMPARISON	LOGICAL ADDRESS	PHYSICAL ADDRESS
Basic	It is the virtual address generated by CPU	The physical address is a location in a memory unit.
Address Space	Set of all logical addresses generated by CPU in reference to a program is referred as Logical Address Space.	Set of all physical addresses mapped to the corresponding logical addresses is referred as Physical Address.
Visibility	The user can view the logical address of a program.	The user can never view physical address of program
Access	The user uses the logical address to access the physical address.	The user can not directly access physical address.
Generation	The Logical Address is generated by the CPU	Physical Address is Computed by MMU

Relocation

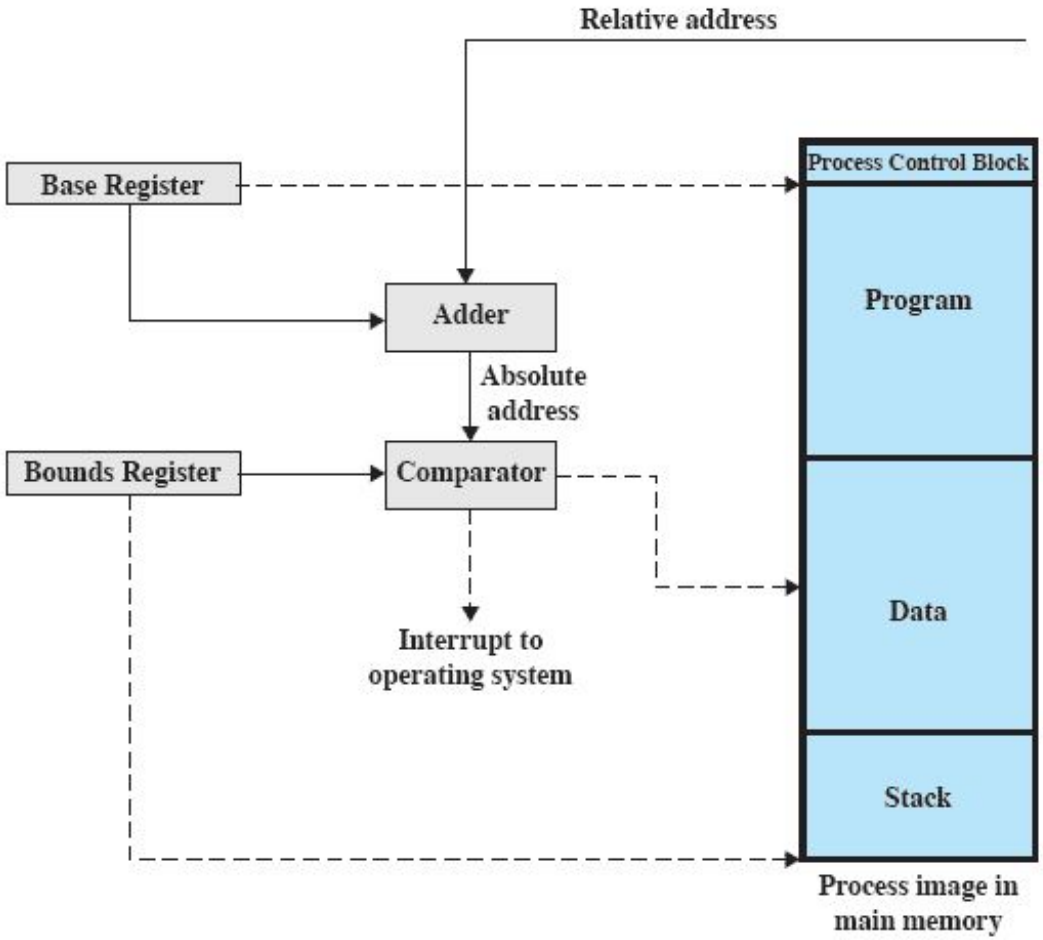
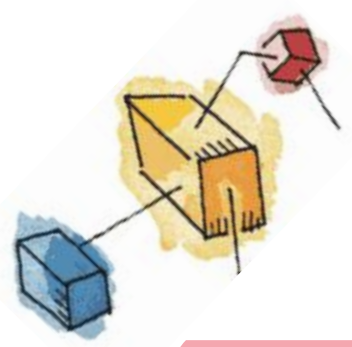
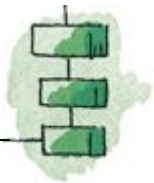


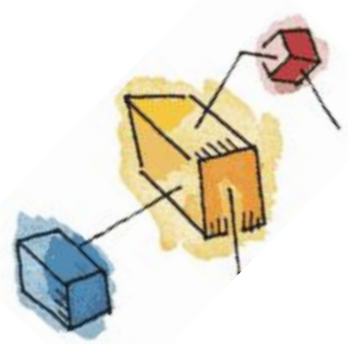
Figure 7.8 Hardware Support for Relocation



Registers Used during Execution

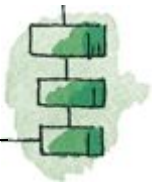
- Base register
  - Starting address for the process
- Bounds register
  - Ending location of the process
- These values are set when the process is loaded or when the process is swapped in





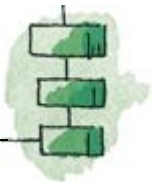
Registers Used during Execution

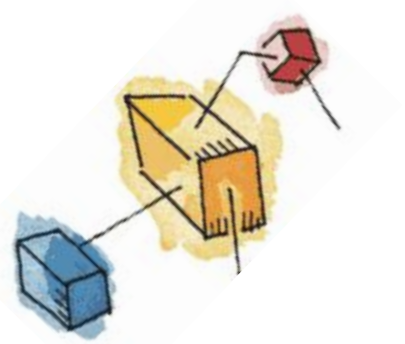
- The value of the base register is added to a relative address to produce an absolute address
- The resulting address is compared with the value in the bounds register
- If the address is not within bounds, an interrupt is generated to the operating system



Paging

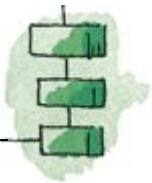
- Partition memory into small equal fixed-size chunks and divide each process into the same size chunks
- The chunks of a process are called ***pages***
- The chunks of memory are called ***frames***



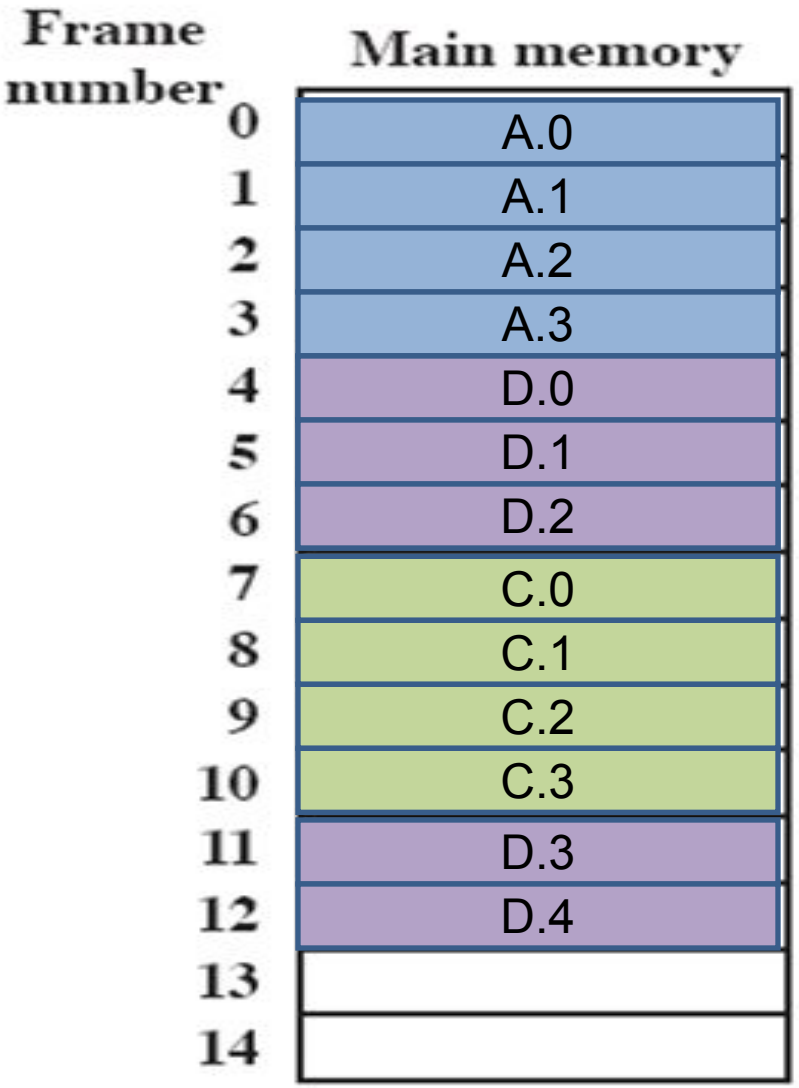


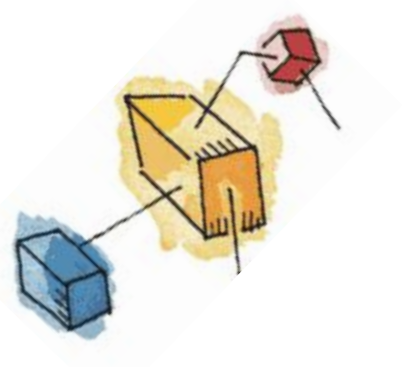
Paging

- Operating system maintains a page table for each process
  - Contains the frame location for each page in the process
  - Memory address consist of a page number and offset within the page



Processes and Frames





Page Table

0	0
1	1
2	2
3	3

Process A page table

0	—
1	—
2	—

Process B page table

0	7
1	8
2	9
3	10

Process C page table

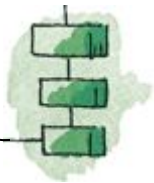
0	4
1	5
2	6
3	11
4	12

Process D page table

13
14

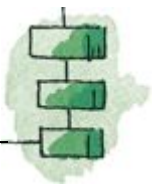
Free frame list

Figure 7.10 Data Structures for the Example of Figure 7.9 at Time Epoch (f)



Segmentation

- A program can be subdivided into segments
  - Segments may vary in length
  - There is a maximum segment length
- Addressing consist of two parts
  - a segment number and
  - an offset
- Segmentation is similar to dynamic partitioning



Logical Addresses

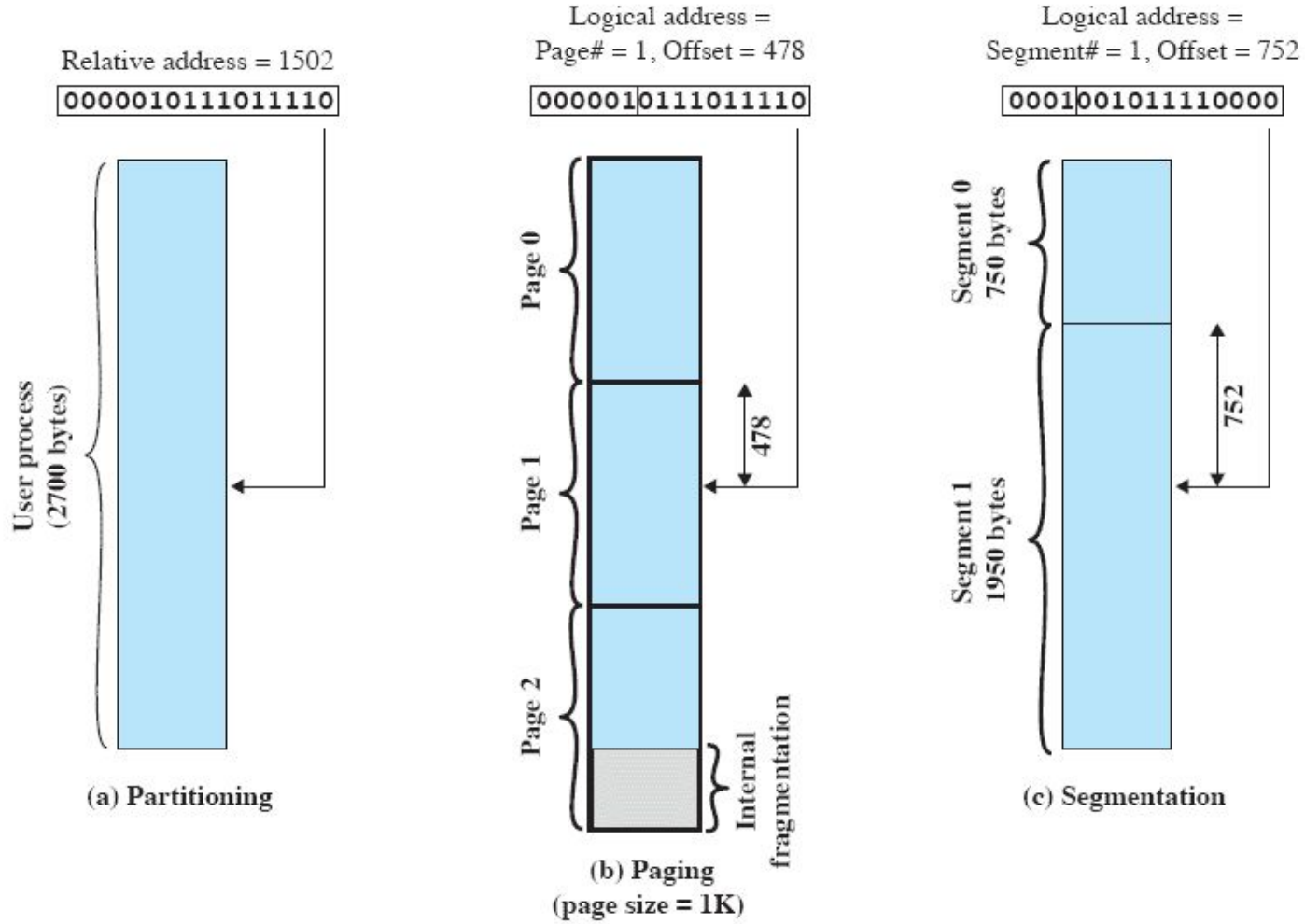
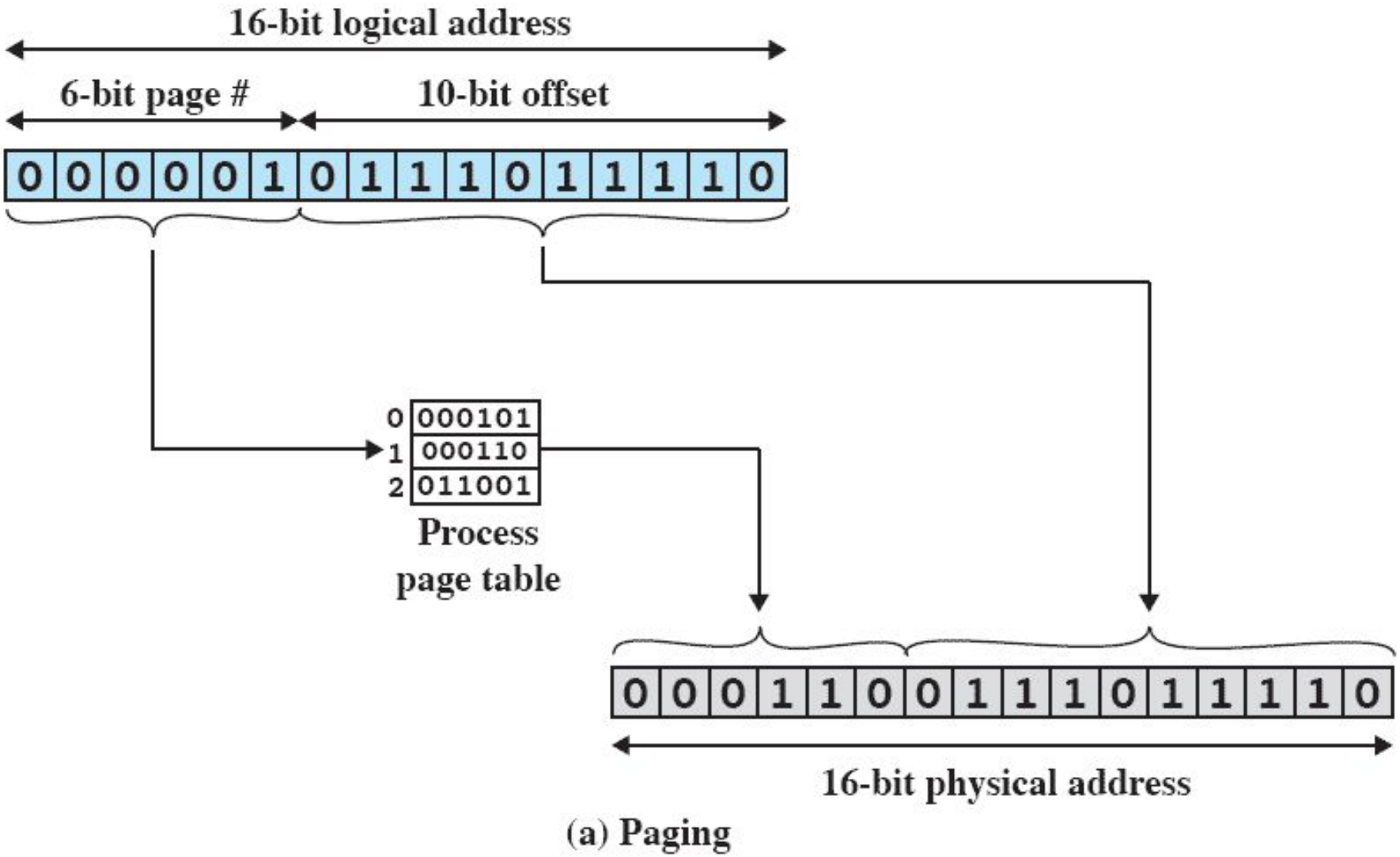


Figure 7.11 Logical Addresses

Paging



(a) Paging

Segmentation

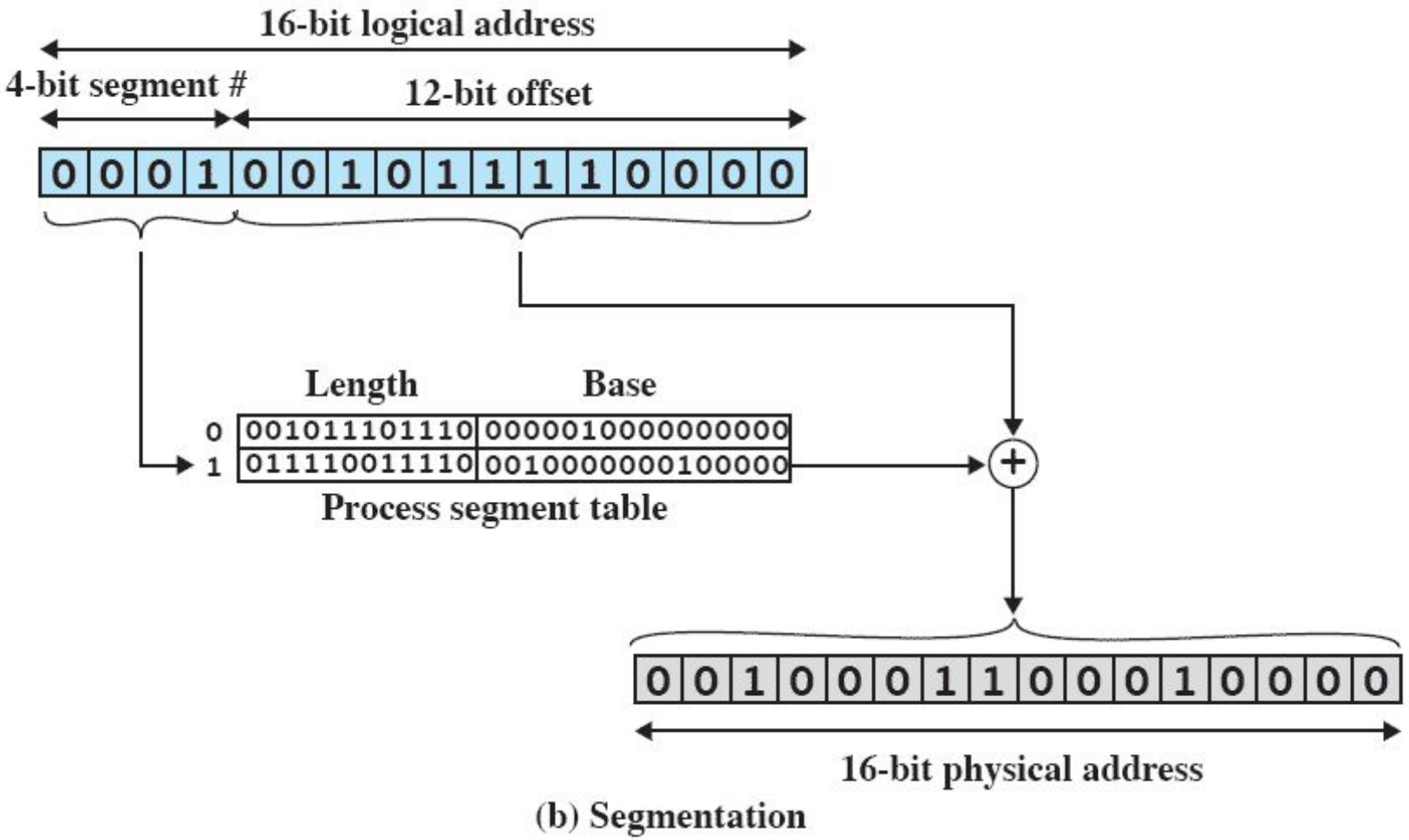


Figure 7.12 Examples of Logical-to-Physical Address Translation

Chapter 5: Memory Management

- Background
- Swapping
- Contiguous Allocation
- Paging
- Segmentation
- Segmentation with Paging

Background

- Address Binding:
  - Program resides on disk.
  - Program must be brought into memory and placed within a process for it to be run.
- *Input queue* – collection of processes on the disk that are waiting to be brought into memory to run the program.
- User programs go through several steps before being run.
- Addresses in source program are symbolic. e.g. count.
- Compiler binds Symbolic address to relocatable address.
- Linkage editor / loader binds relocatable address to absolute (physical)address.

Binding of Instructions and Data to Memory

Address binding of instructions and data to memory addresses can happen at three different stages.

- **Compile time**: If memory location known a priori, absolute code can be generated; must recompile code if starting location changes.
- **Load time**: Must generate *relocatable* code if memory location is not known at compile time.
- **Execution time**: Binding delayed until run time if the process can be moved during its execution from one memory segment to another. Need hardware support for address maps (e.g., *base* and *limit registers*).

Dynamic Loading

- Routine is not loaded until it is called
- Calling routine first checks whether other routine is loaded then it loads.
- Better memory-space utilization; unused routine is never loaded.
- Useful when large amounts of code are needed to handle infrequently occurring cases. e.g. error handling.

Dynamic Linking

- Linking postponed until execution time.
- Small piece of code, *stub*, used to locate the appropriate memory-resident library routine.
- Stub replaces itself with the address of the routine, and executes the routine.
- Operating system needed to check if routine is in processes' memory address.
- Dynamic linking is particularly useful for libraries.

Logical vs. Physical Address Space

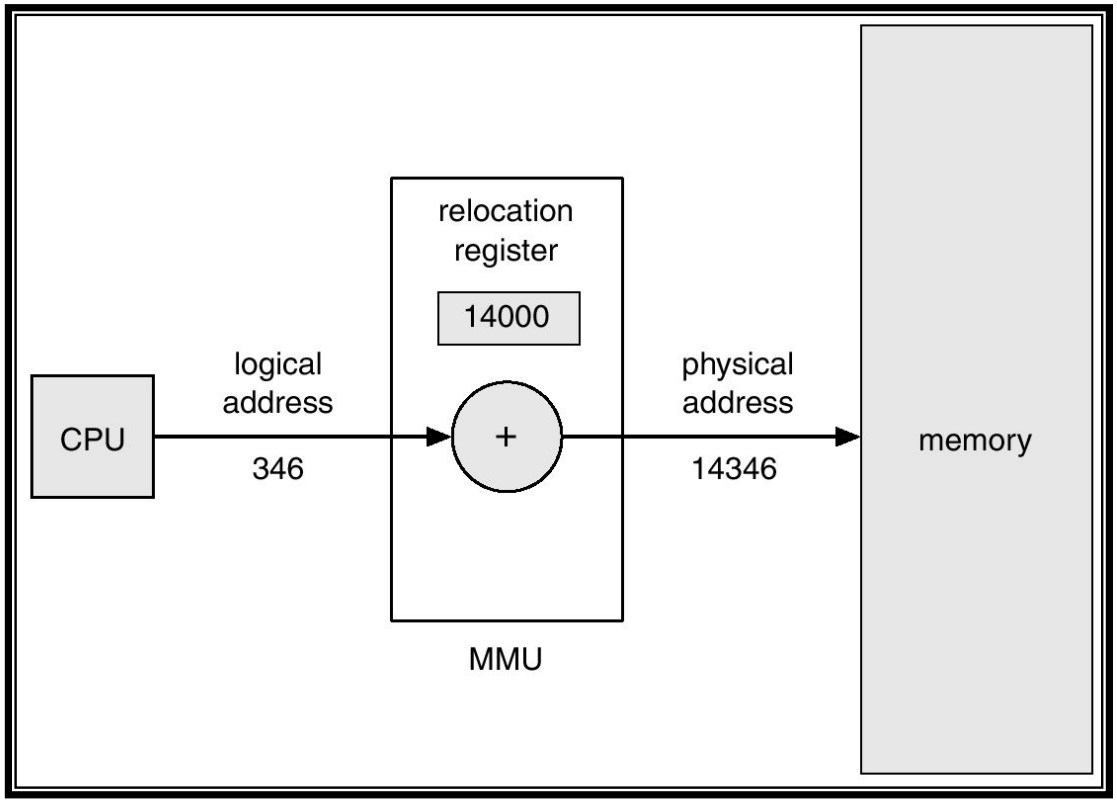
- The concept of a logical *address space* that is bound to a separate *physical address space* is central to proper memory management.
  - *Logical address* – generated by the CPU; also referred to as *virtual address*.
  - *Physical address* – address seen by the memory unit.

- Logical and physical addresses are the same in compile-time and load-time address-binding schemes; logical (virtual) and physical addresses differ in execution-time address-binding scheme.

Memory-Management Unit (MMU)

- Hardware device that maps virtual to physical address.
- In MMU scheme, the value in the relocation register is added to every address generated by a user process at the time it is sent to memory.
- The user program deals with *logical* addresses; it never sees the *real* physical addresses.

Dynamic relocation using a relocation register

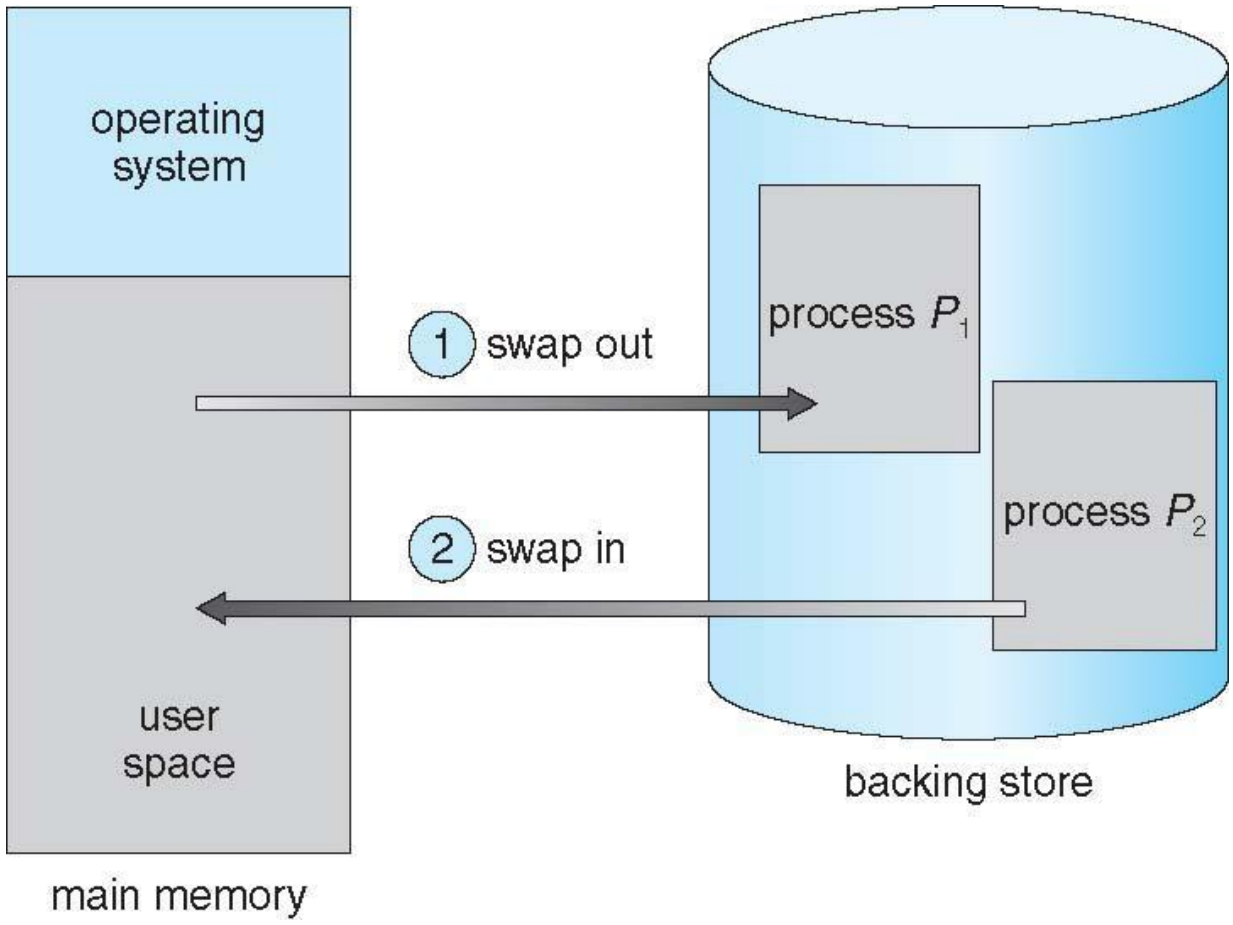


Relocation is base register

Swapping

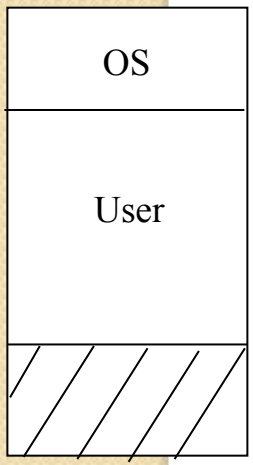
- A process can be *swapped* temporarily out of memory to a *backing store*, and then brought back into memory for continued execution.
- Backing store – fast disk large enough to accommodate copies of all memory images for all users; must provide direct access to these memory images.
- *Roll out, roll in* – swapping variant used for priority-based scheduling algorithms; lower-priority process is swapped out so higher-priority process can be loaded and executed.
- Major part of swap time is transfer time; total transfer time is directly proportional to the *amount* of memory swapped.
- Modified versions of swapping are found on many systems, i.e., UNIX, Linux, and Windows.

Schematic View of Swapping



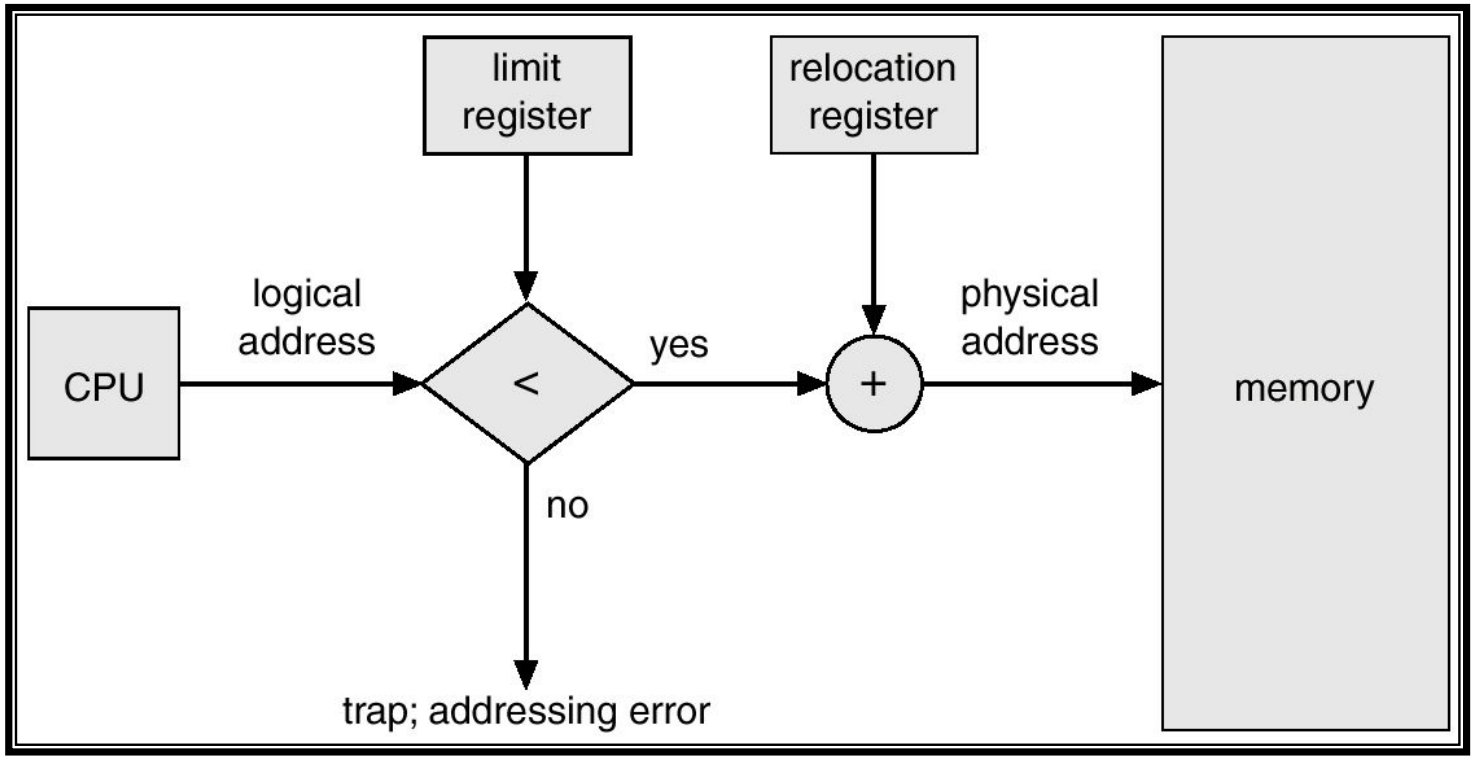
Contiguous Allocation

- Main memory usually into two partitions:
  - Resident operating system, usually held in low memory with interrupt vector.
  - User processes then held in high memory.
- **Single-partition allocation**
  - OS is need to be protected from malicious user processes & protect processes from each other.



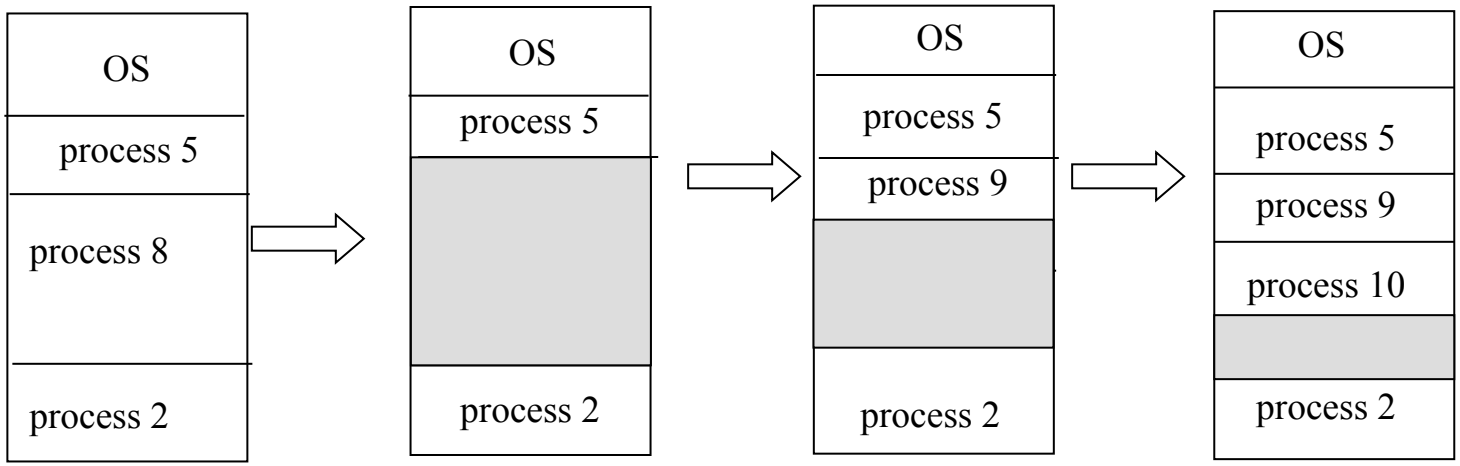
  - Relocation-register scheme used to protect user processes from each other, and from changing operating-system code and data.
  - Relocation register contains value of smallest physical address; limit register contains range of logical addresses – each logical address must be less than the limit register.

Hardware Support for Relocation and Limit Registers

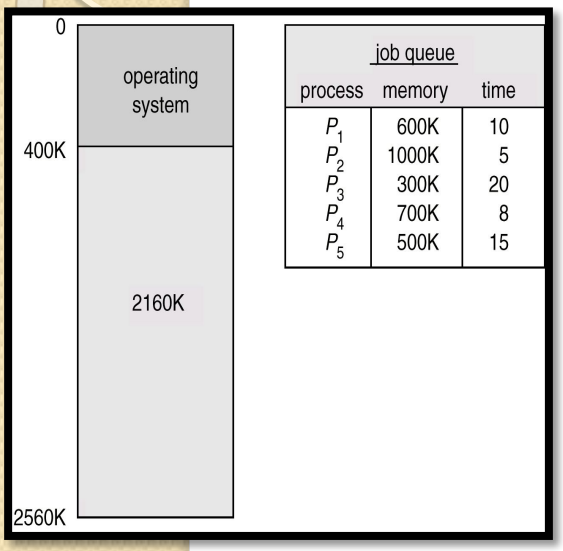


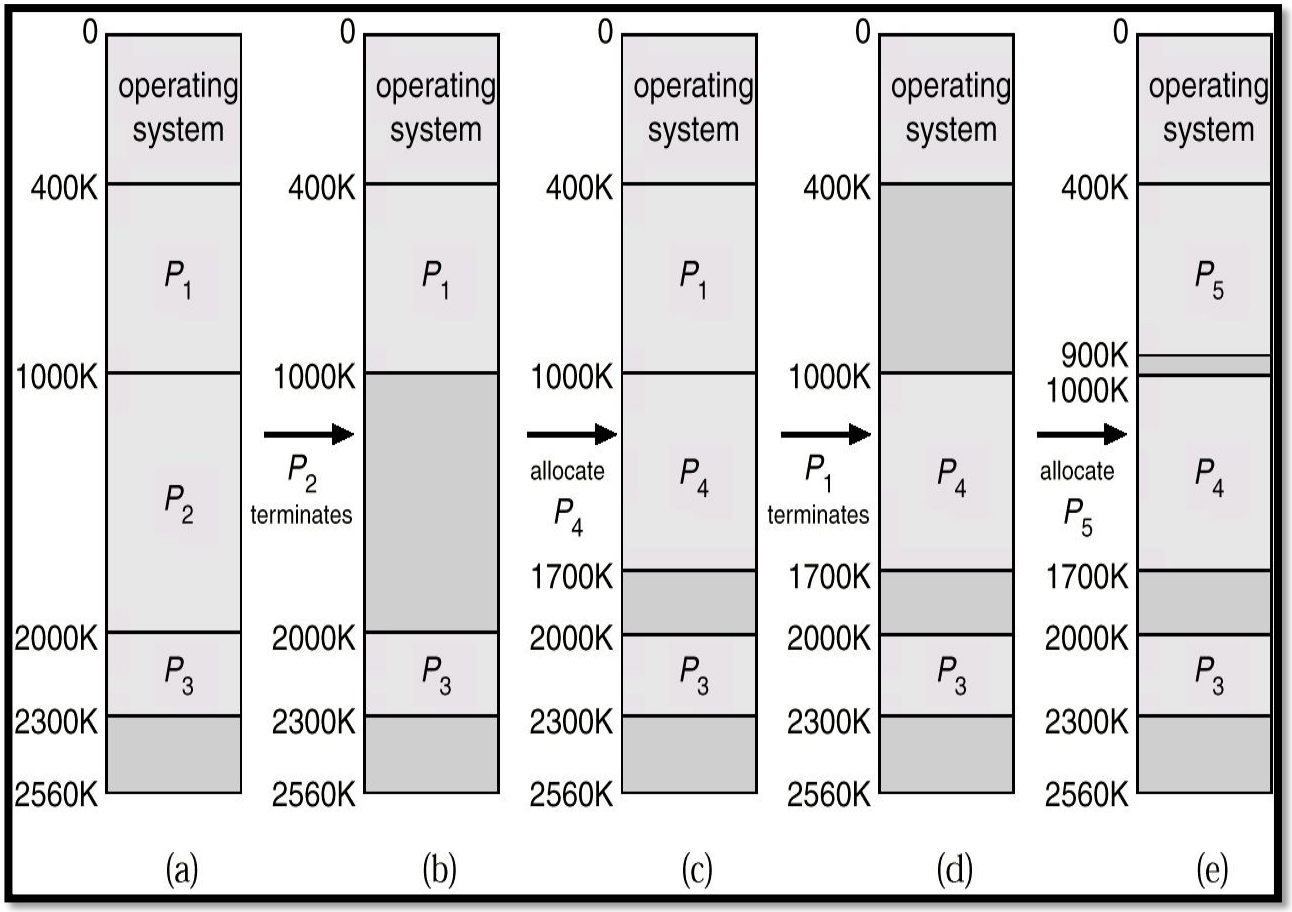
Contiguous Allocation (Cont.)

- **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)



Multiple partition Allocation





- How to satisfy request of size n from list of free holes?

Dynamic Storage-Allocation Problem

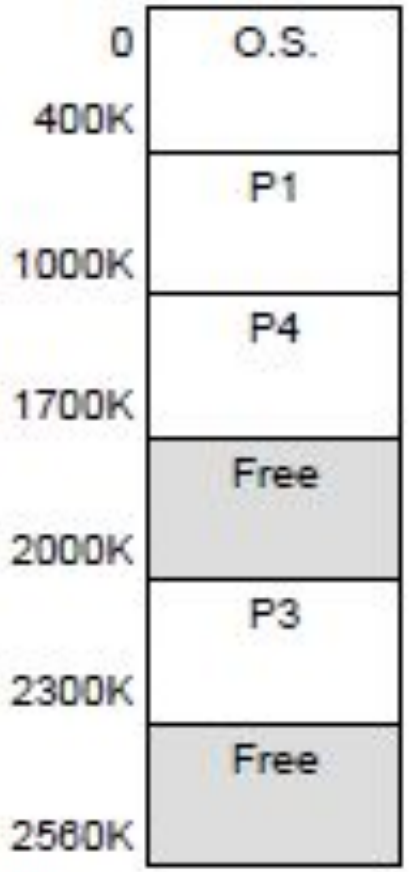
How to satisfy a request of size *n* from a list of free holes.

- **First-fit**: Allocate the *first* hole that is big enough.
- **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.

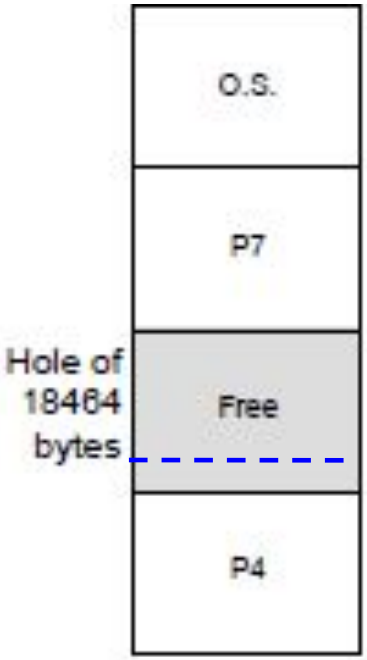
Fragmentation

- **External Fragmentation** – total memory space exists to satisfy a request, but it is not contiguous. It is fragmented into a large number of small holes.
- Example: We have a total external fragmentation of (300+260)=560KB.
- If P5 is 500KB, then this space would be large enough to run P5.
- But the space is not contiguous.



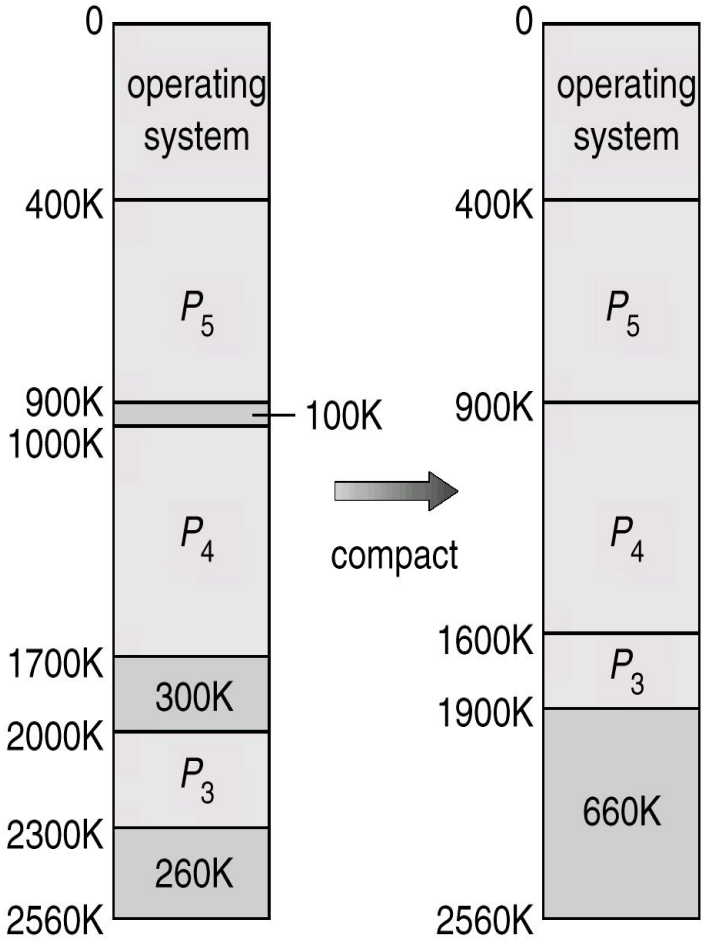
Fragmentation

- **Internal Fragmentation** – allocated memory may be slightly larger than requested memory; this size difference is memory internal to a partition, but not being used.
- Example: Assume next request is for 18462 bytes.
- – If we allocate exactly the requested block, we are left with a hole of 2 bytes.
- The overhead to keep track of this hole will be larger than the hole itself. So, we ignore this small hole (internal fragmentation).



External fragmentation: Solution

- Reduce external fragmentation by compaction
  - Shuffle memory contents to place all free memory together in one large block.
  - Compaction is possible *only* if relocation is dynamic, and is done at execution time.

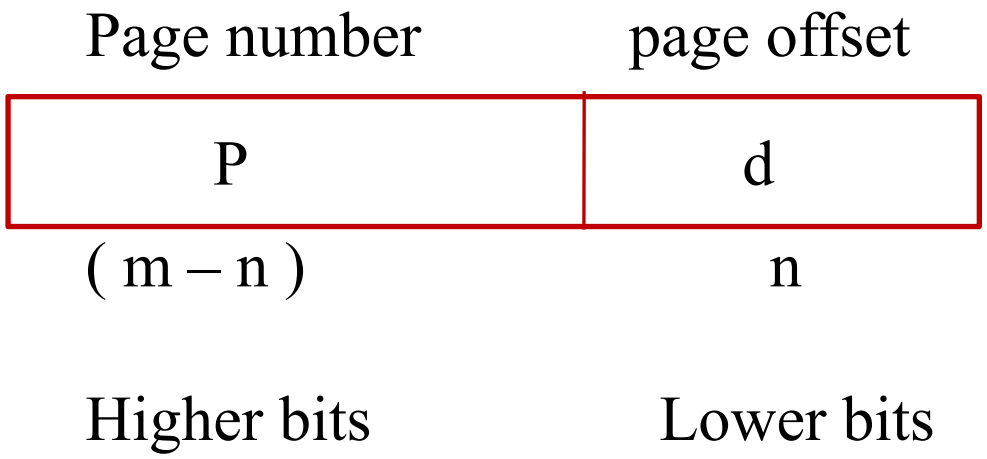


Paging

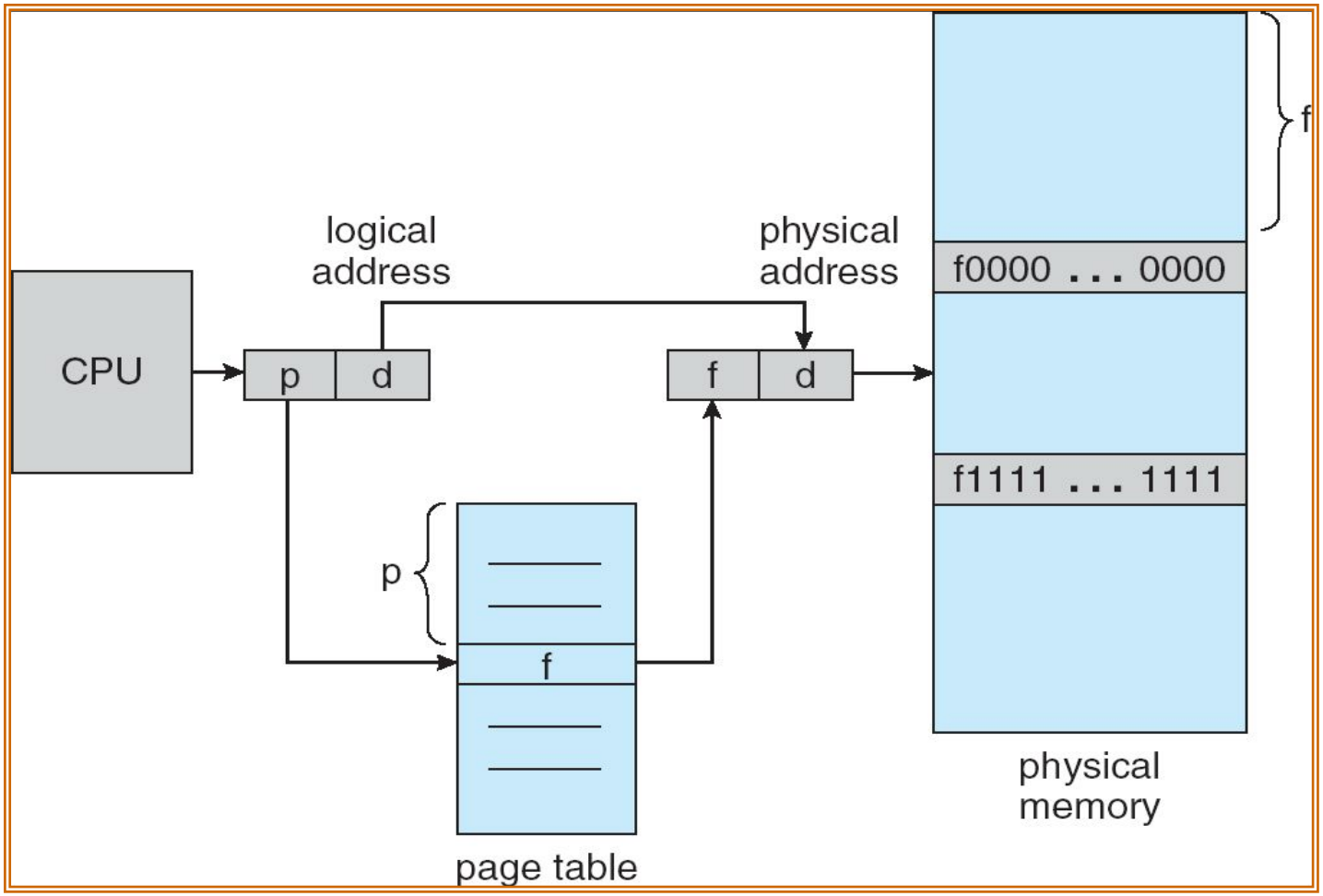
- Logical address space of a process can be noncontiguous; process is allocated physical memory whenever the latter is available.
- Divide physical memory into fixed-sized blocks called **frames** (size is power of 2, between 512 bytes and 8192 bytes).
- Divide logical memory into blocks of same size 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: No external fragmentation, but have some internal fragmentation.(if pages are 512 bytes, a process of 1280 bytes would need 2 pages plus 256 bytes.)

Address Translation Scheme

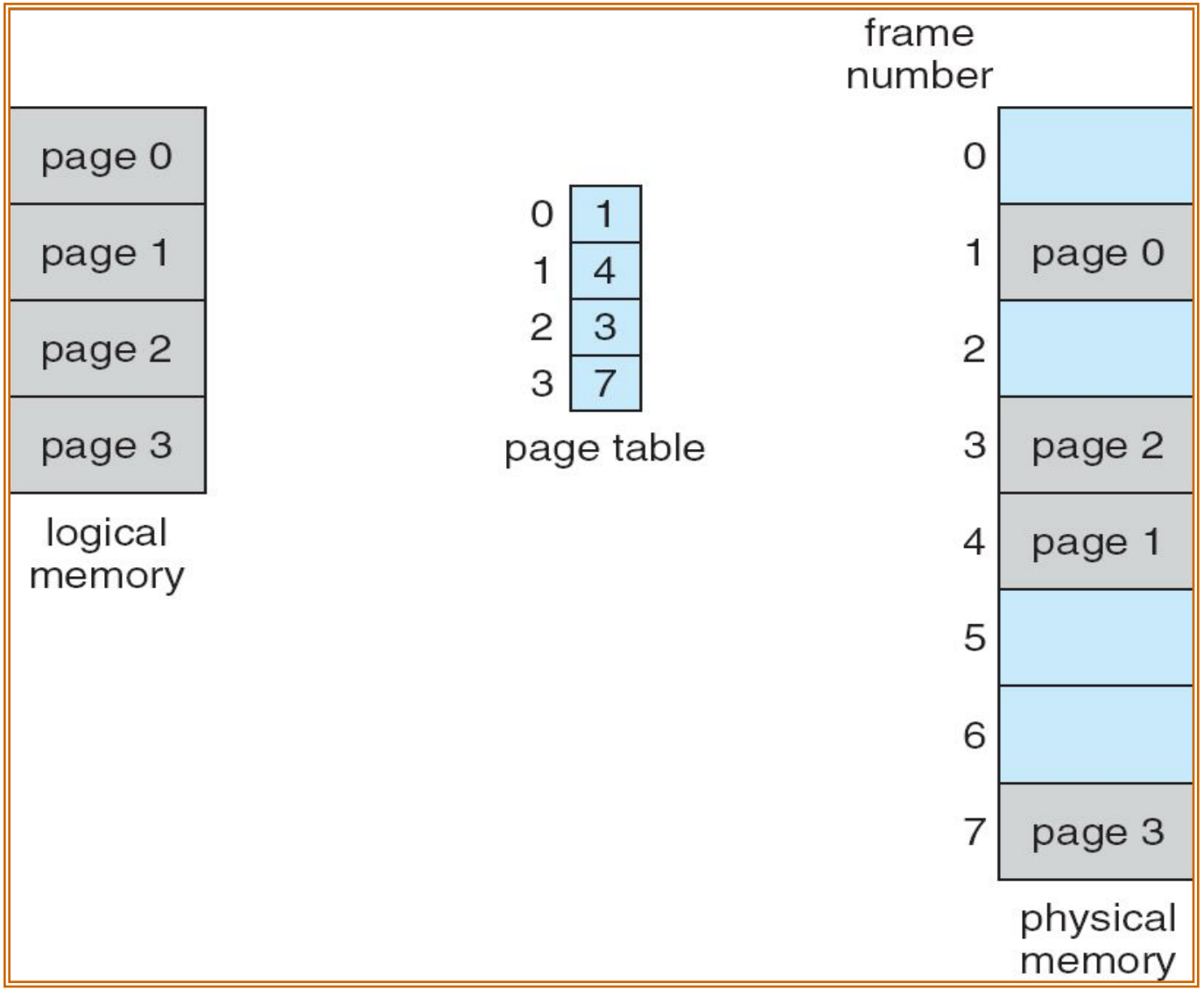
- Address generated by CPU is divided into:
  - *Page number (p)* – used as an index into a *page table* which contains base address of each page in physical memory.
  - *Page offset (d)* – combined with base address to define the physical memory address that is sent to the memory unit.



Address Translation Architecture

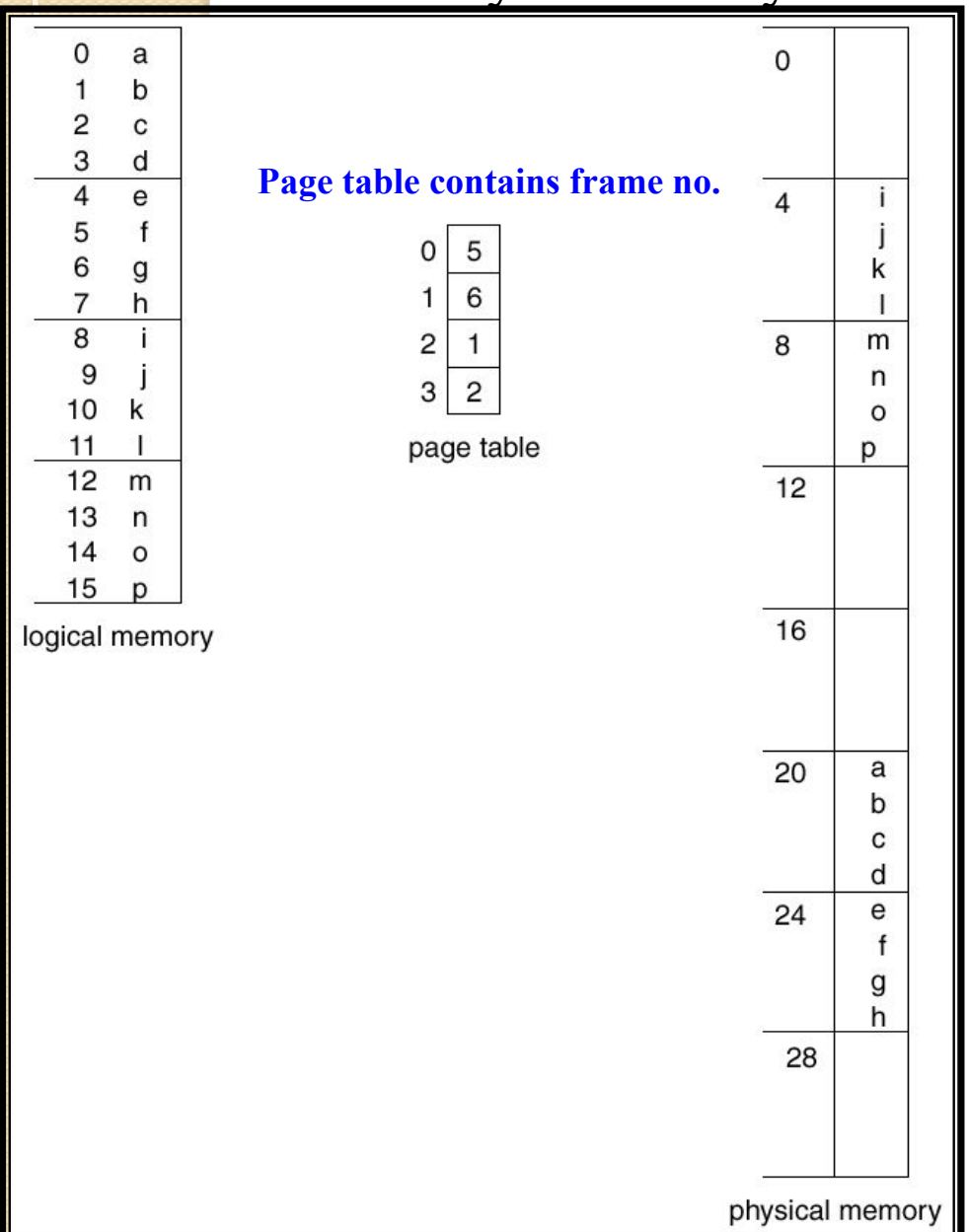


Paging Example



Paging Example:

for a 32-byte memory with 4 byte pages



Logical memory space=2^4=16 bytes

Page size=2^2=4 bytes

m=4

n=2

m-n = page no bits

N=offset bits

Logical address 0, is page 0, offset = 0

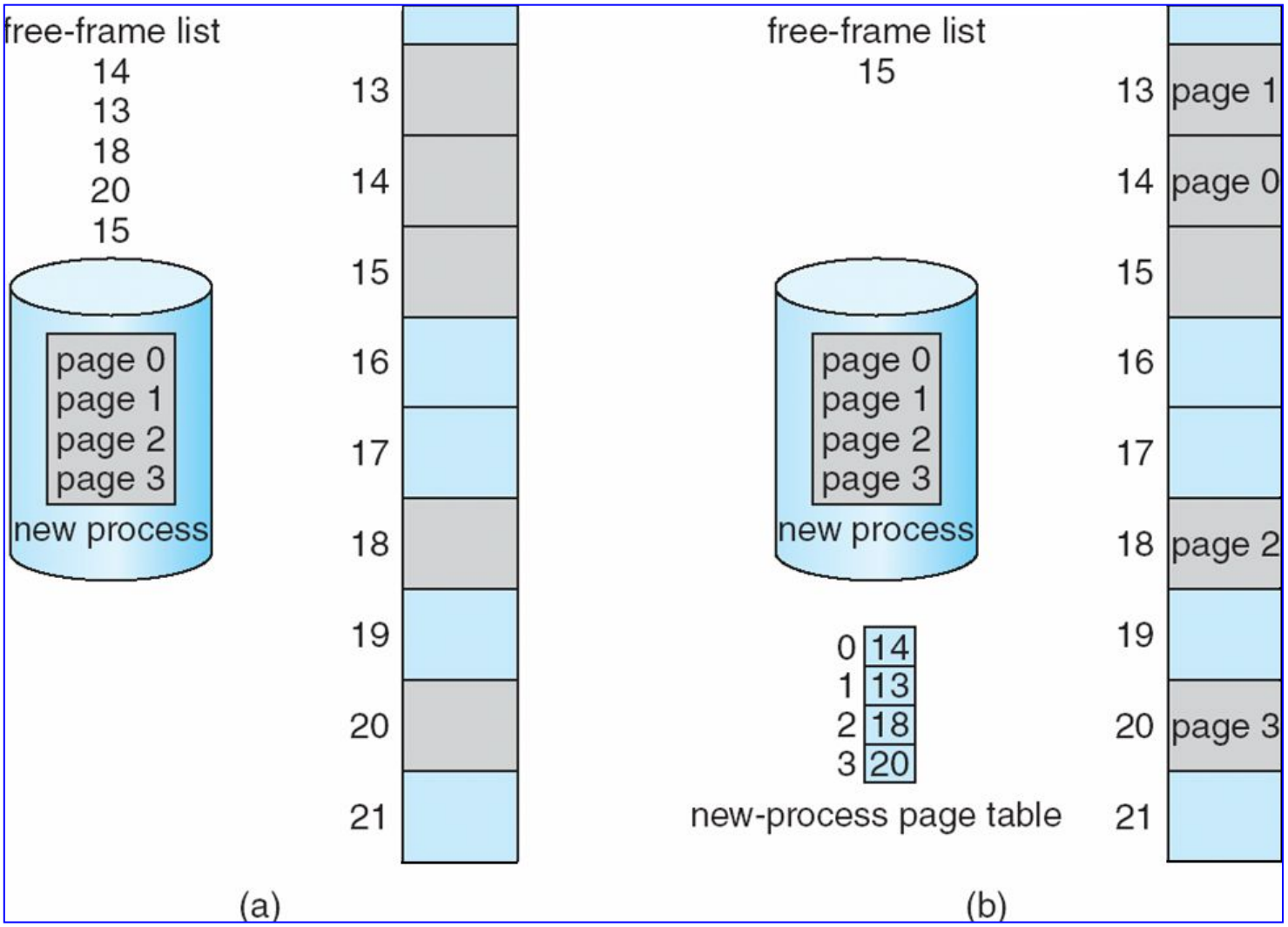
So physical address= (5 * 4) + 0 = 20

Logical address 3, is page 0, offset = 3

So physical address = (5 * 4) + 3 = 23

- As a concrete (although minuscule) example, consider the memory in Figure 8.9.
-
- Here, in the logical address, n= 2 and m = 4. Using a page size of 4 bytes and a physical memory of 32 bytes (8 pages), we show how the user's view of memory can be mapped into physical memory. Logical address 0 is page 0, offset 0.
-
- Indexing into the page table, we find that page 0 is in frame 5. Thus, logical address 0 maps to physical address 20 [= (5 x 4) + 0].
- Logical address 3 (page 0, offset 3) maps to physical address 23 [= (5 x 4) + 3].
-
- Logical address 4 is page 1, offset 0; according to the page table, page 1 is mapped to
- frame 6.
-
- Thus, logical address 4 maps to physical address 24 [= (6 x 4) + O].
- Logical address 13 maps to physical address 9.

Free Frames



Before allocation

After allocation

Memory Protection

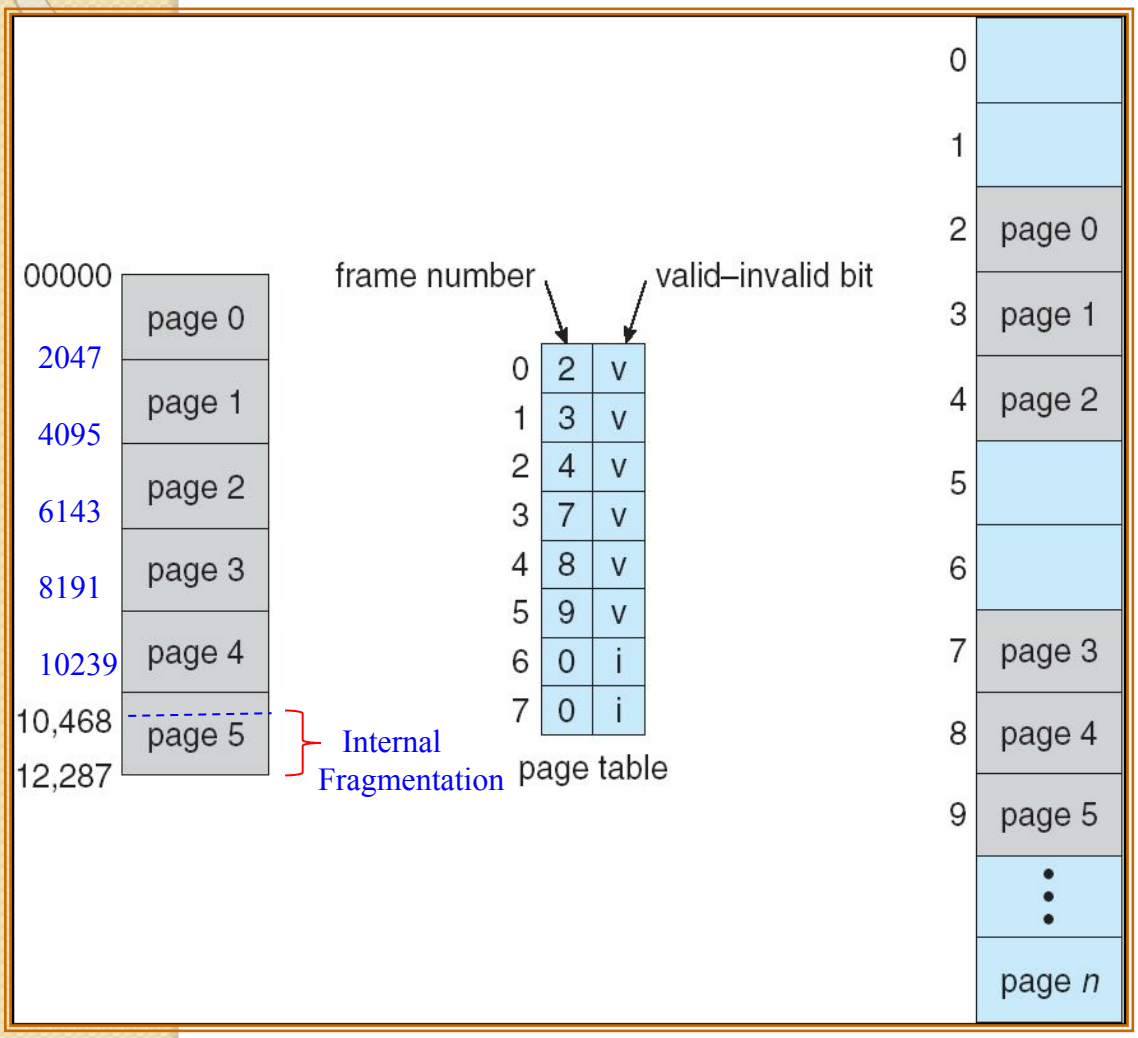
- Memory protection implemented by associating protection bit with each frame.
- One bit can define a page to be read-write or read only. Every reference to memory goes through the page table to find correct frame number. While computing physical address, protection bits are checked to verify, no writers are being made to read-only page.
- Illegal attempts are trapped by OS. One more bit is *Valid-invalid* bit
- *Valid-invalid* bit attached to each entry in the page table:
  - "valid" indicates that the associated page is in the process' logical address space, and is thus a legal page.
  - "invalid" indicates that the page is not in the process' logical address space.

Valid (v) or Invalid (i) Bit In A Page Table

Consider a system with 14 bit address space (0 - 16383)

1 page = 2 KB = 2^{11} = 2048 bytes

A program requires address space 0 - 10468



Here only 0-5 pages are valid, attempts to access page 6 & 7 set protection bit to invalid. Page 5 is classified as valid so, addresses upto 12, 287 are valid & addresess from 12288-16383 are invalid.

Segmentation

- Memory-management scheme that supports user view of memory.
- User prefer to see memory as collection of variable length segments than logical address space as array of bytes.
- A program is a collection of segments. A segment is a logical unit such as:

main program,

procedure,

function,

method,

object,

local variables, global variables,

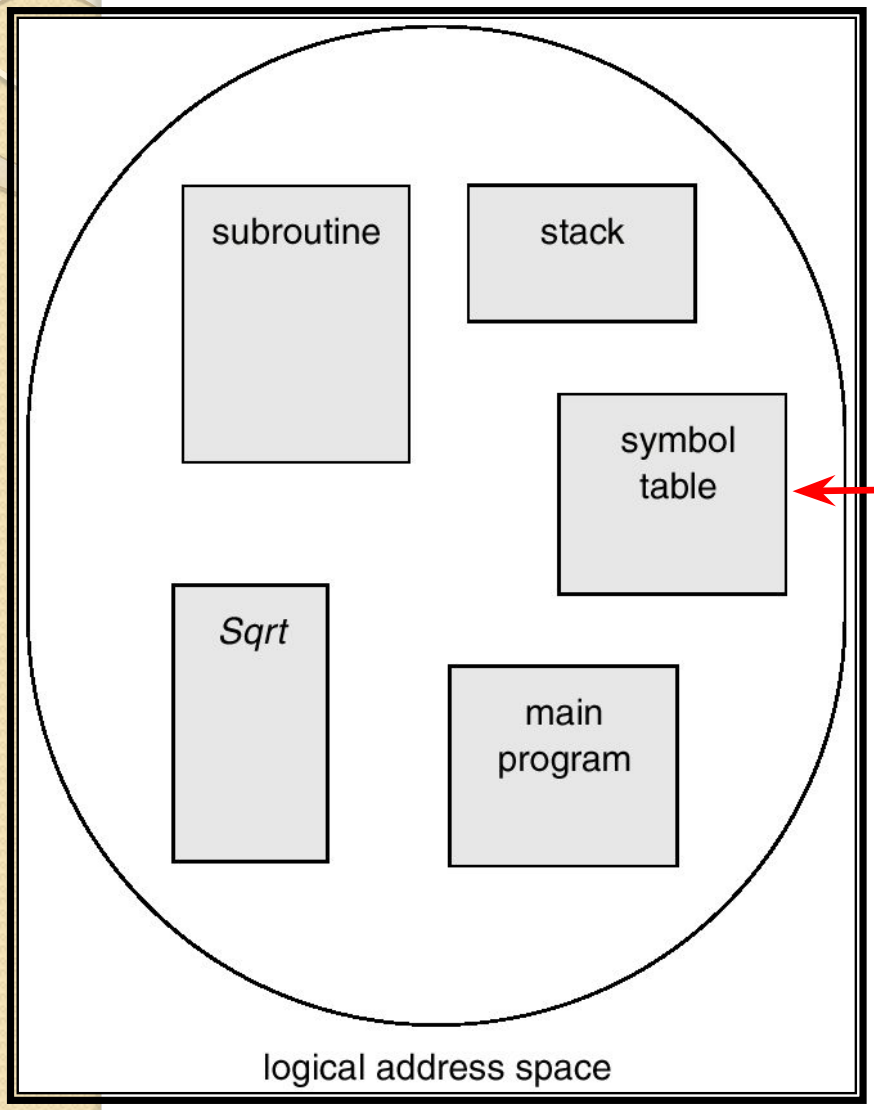
common block,

stack,

symbol table, arrays

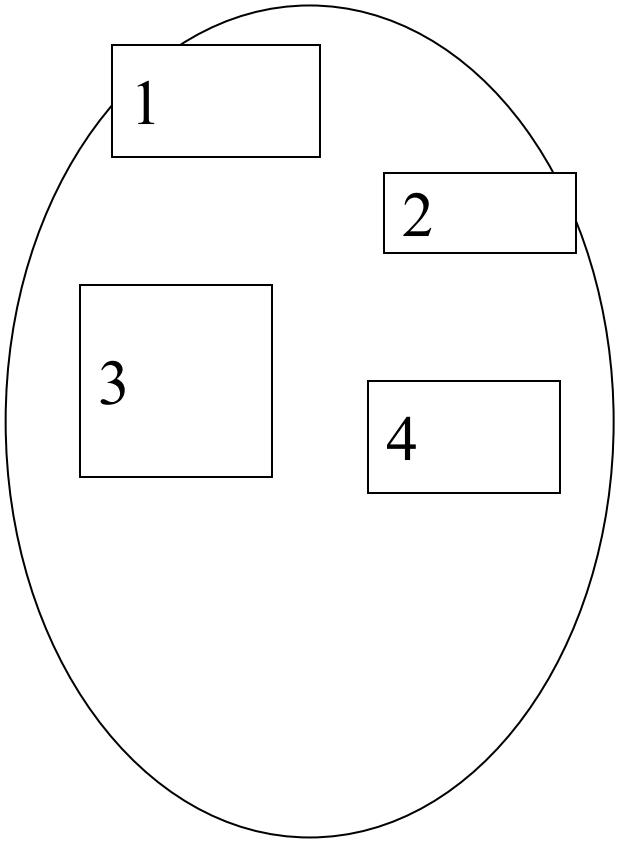
Compiler automatically constructs segments of input program.

User's View of a Program

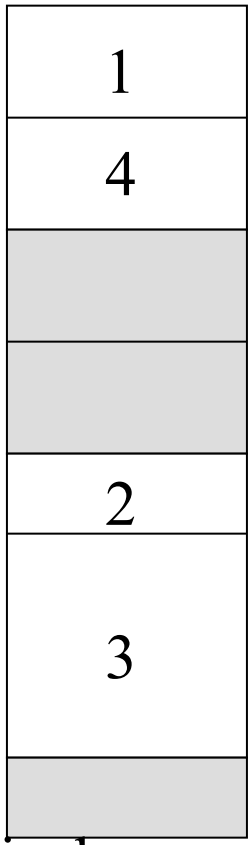


Segments without caring what addresses in memory these elements occupy

Logical View of Segmentation



user space



physical memory space

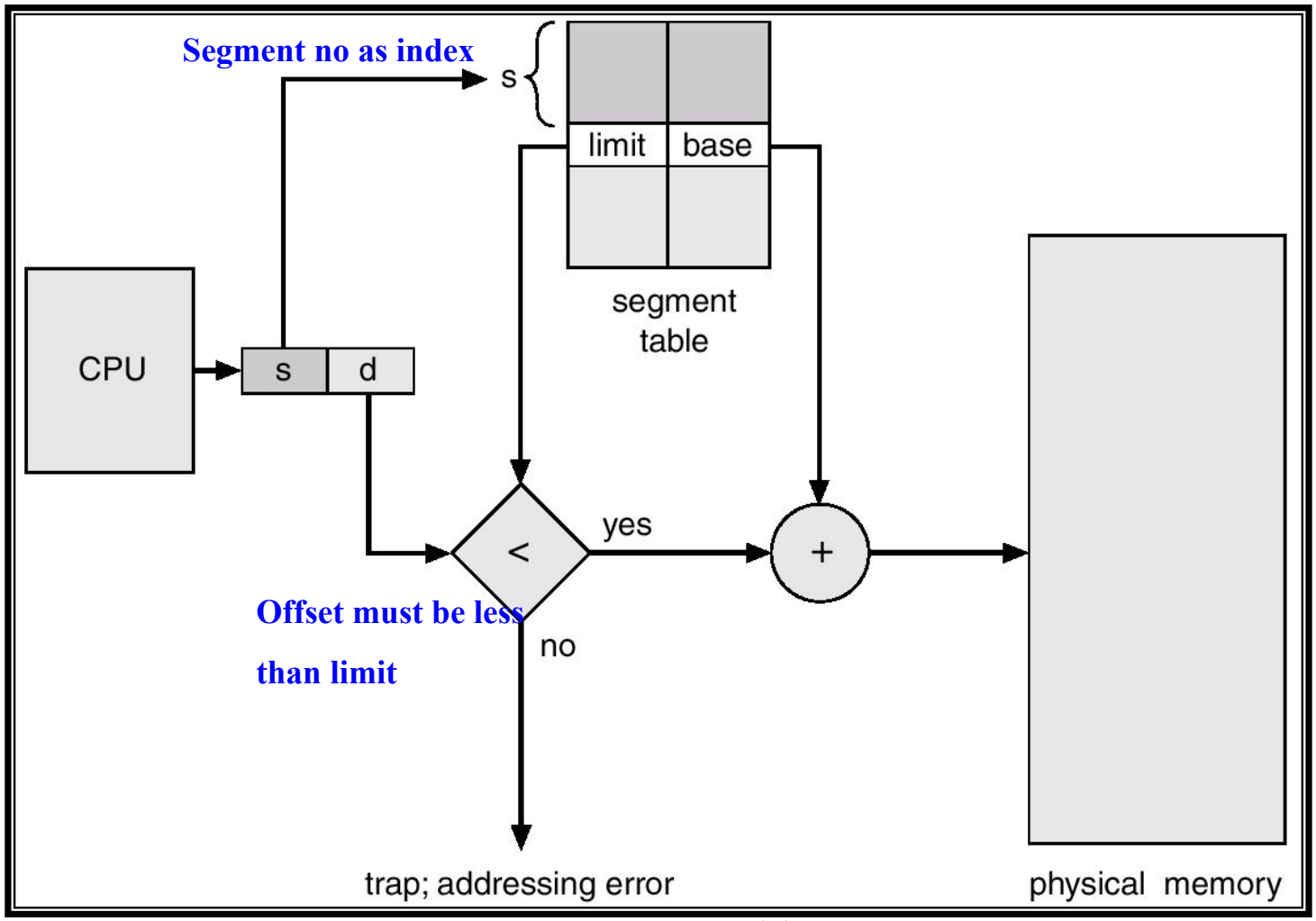
Segmentation Architecture

- Logical address space is collection of segments. Each segment has name & length.
- Logical address consists of a two tuple:

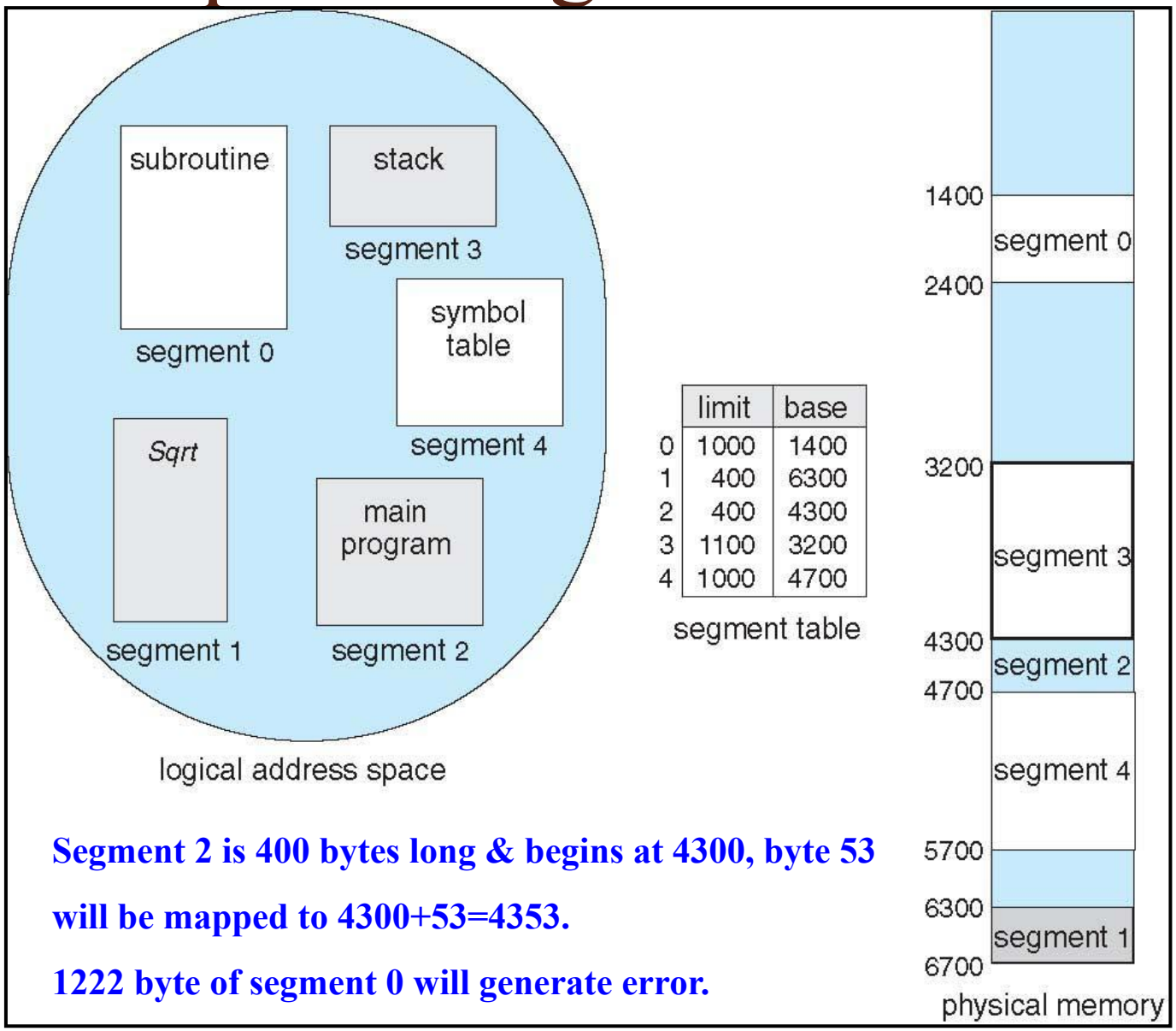
  <segment-number, offset>,

  for simplicity segment number than name
- *Segment table* – maps two-dimensional physical addresses; each table entry has:
  - ***Segment base*** – contains the starting physical address where the segments reside in memory.
  - ***Segment limit*** – specifies the length of the segment.
- *Segment-table base register (STBR)* points to the segment table's location in memory.
- *Segment-table length register (STLR)* indicates number of segments used by a program;

Segmentation Hardware



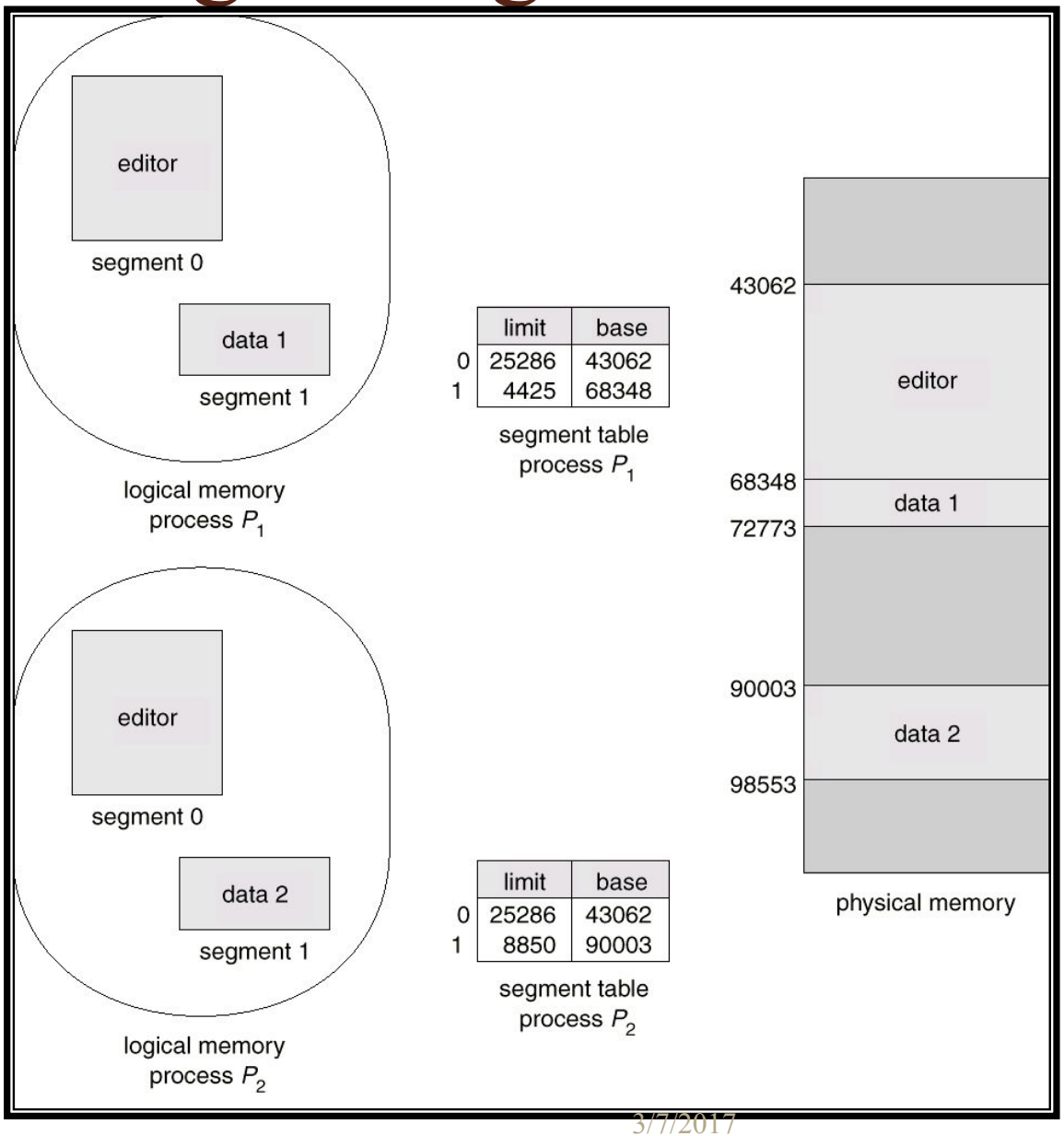
Example of Segmentation



Segment 2 is 400 bytes long & begins at 4300, byte 53 will be mapped to 4300+53=4353.

1222 byte of segment 0 will generate error.

Sharing of Segments



Segment Protection and Sharing

- With each entry of segment table associate:
  - Validation bit = 0 => illegal segment
  - Read/write/execute privilege
- Protection bits associated with segments; code sharing occurs a segment level.
- Sharing is simple but subtle consideration. Code segment contain references. Conditional branching may give transfer address and segment number of code segment is segment number of transfer address. Sqrt( ) function may get segment no 4 in one process & seg no 17 by another process.

Fragmentation

- Segmentation cause external fragmentation, when all blocks of free memory are too small to accommodate a segment.
- In this case, the process may simply have to wait :
  - until more memory (or at least a larger hole) becomes available, or
  - until compaction creates a larger hole.
- **Solution:**
- Define each process to be one segment, which reduces to the variable-sized partition scheme.
- At the other extreme, each byte as segment . Then every byte need a base register for its relocation, doubling memory use!
- Next logical solution is fixed-sized, small segments ---is paging.

Segmentation with Paging

- This solution differs from pure segmentation in that the segment-table entry contains not the base address of the segment, but rather the base address of a *page table* for this segment.

Segmentation with Paging – Intel 386

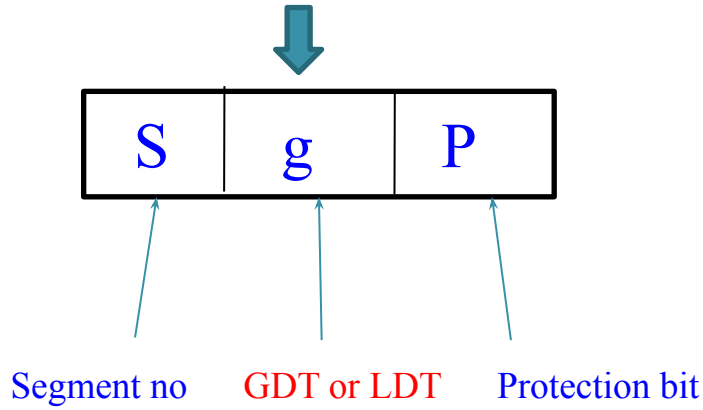
- The 386 uses segmentation with paging for memory management.
- Maximum number of segments per process = 16 KB
- Each segment can be as large as 4 GB= 2^{32} bytes.
- Page size is 4 KB.
- Logical-address space of a process is divided into 2 partitions.
- Up to 8 KB segments that are private to that process. kept in the **local descriptor table (LDT)**

a) Up to 8 KB segments that are shared among

b) all the processes. kept in the **global descriptor table (GDT).**

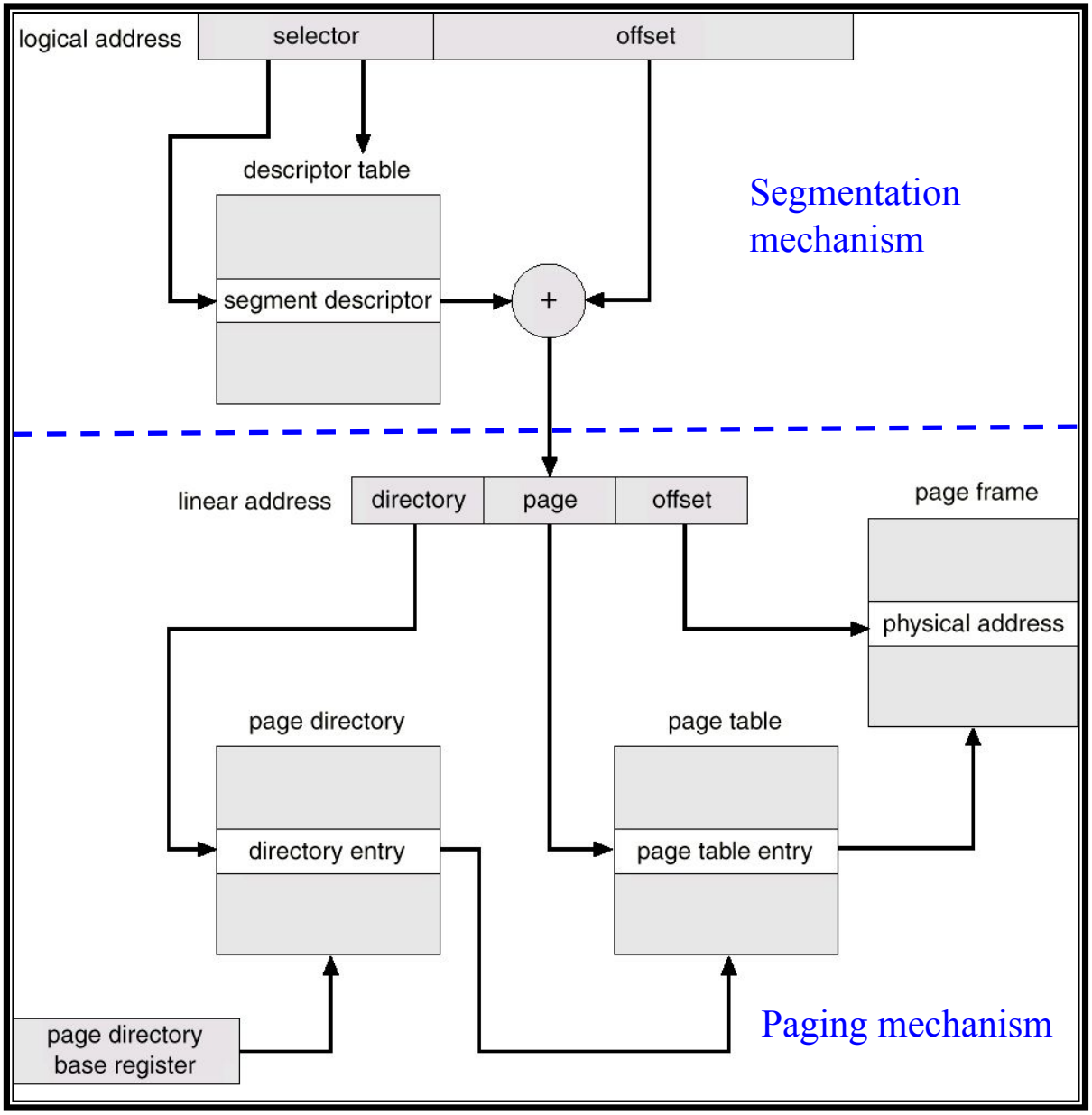
Segmentation with Paging – Intel 386

- Each segment is paged.
- Logical address is pair of (selector, offset).



- First logical address is converted into linear address to give page directory, page and offset.
- Then linear address is converted into physical address

Intel 30386 Address Translation



Virtual Memory

Background

Need of Virtual memory : in many cases, the entire program is not needed.

1. Programs often have code to handle unusual error conditions. Since these errors seldom, this code is almost never executed.
2. An array may be declared 100 × 100 elements, even though it is but used only 10 × 10 elements.
3. An assembler symbol table may have room for 3,000 symbols, although the average program has less than 200 symbols.

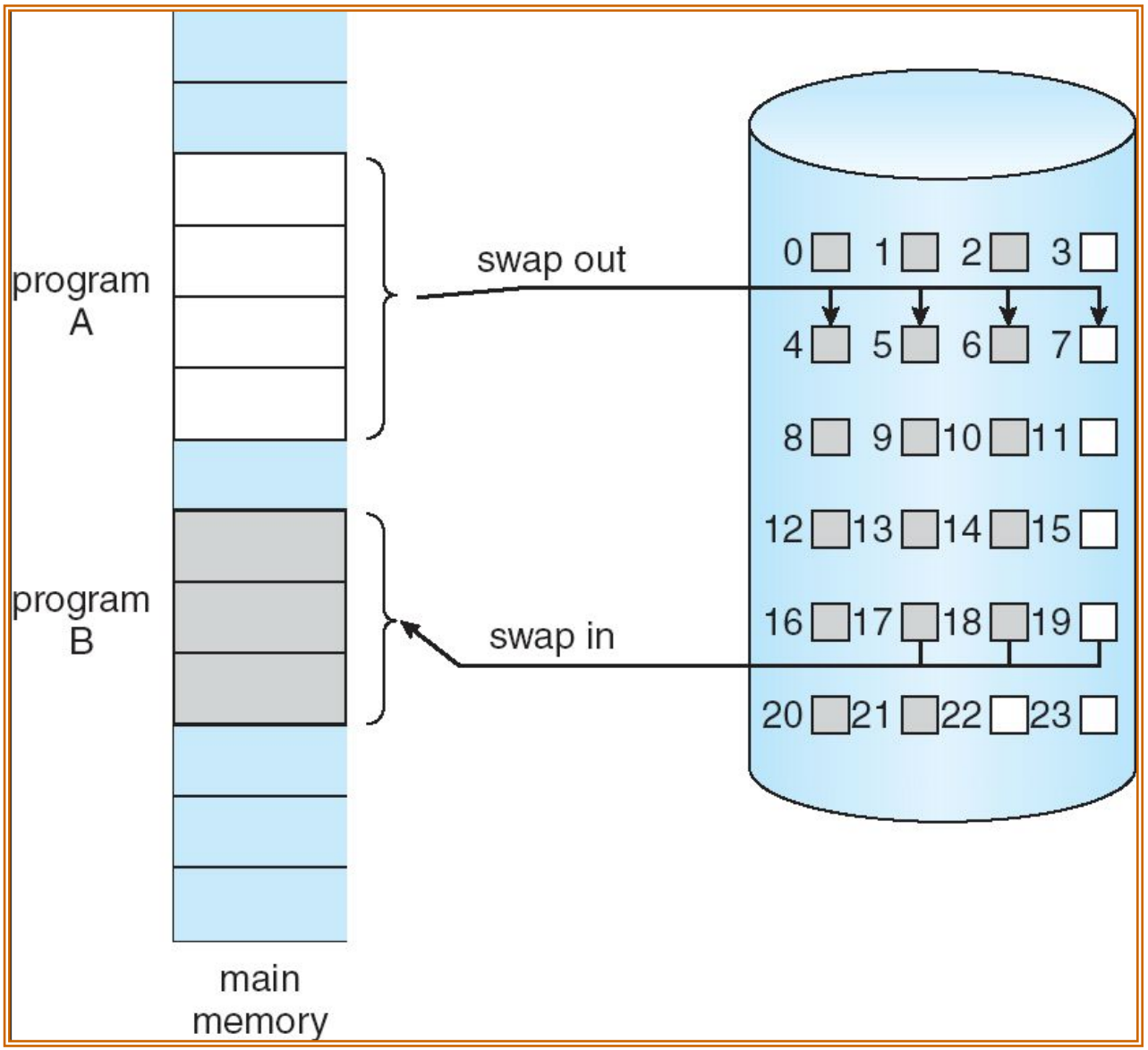
Demand Paging: paging with swapping

- Process is a sequence of pages hence pager swaps in and out pages from disk space.
- Bring a page into memory only when it is needed.
- Thus, it avoids reading into memory pages that will not be used anyway, decreasing the swap time and the amount of physical memory needed.

Demand Paging: paging with swapping

- Page is needed ⇒ reference to it
  - invalid reference ⇒ abort
  - not-in-memory ⇒ bring to memory

Transfer of a Paged Memory to Contiguous Disk Space



Valid-Invalid Bit

Access to page marked invalid causes a page-fault trap.

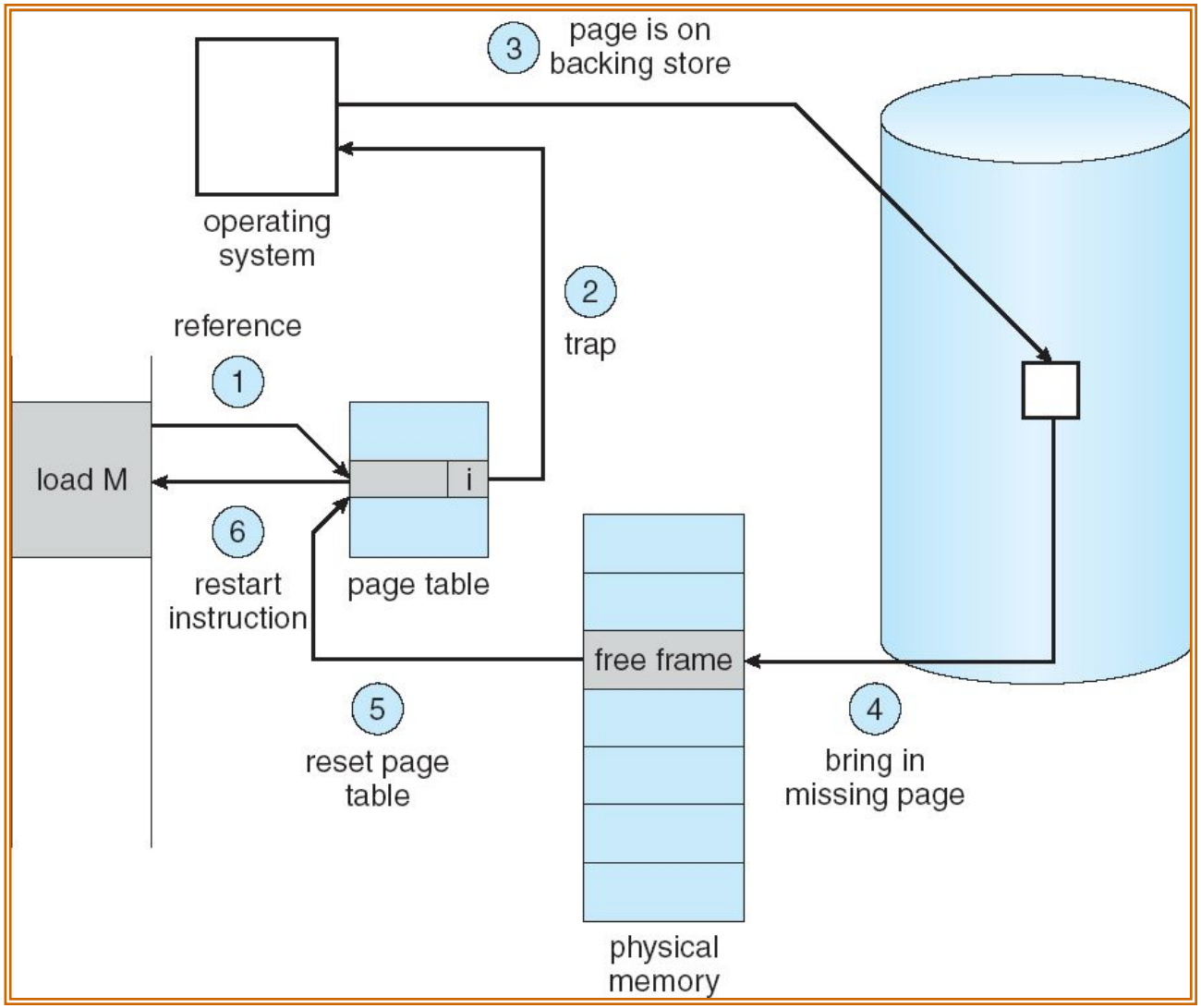
1. When bit is set to "valid" indicates the associated page is both legal and in memory.
2. If set to invalid, indicates either is page not valid (not present in logical space) or is valid but currently on disk.

Page Fault: steps to handle page fault

- If there is a reference to a page that was not brought into memory, first reference to that page will trap to operating system: **page fault**

1. Check internal table, to determine whether reference was valid or invalid.
2. If the reference was invalid, terminate the process. If it was valid, but not yet brought in memory, then bring page in.
3. Get empty frame from the free-frame list
4. Swap page into frame.
5. Reset tables, Set validation bit = **v**.
6. Restart the instruction that caused the page fault

Steps in Handling a Page Fault



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.

Basic Page Replacement

1. Find the location of the desired page on disk.

2. Find a free frame:

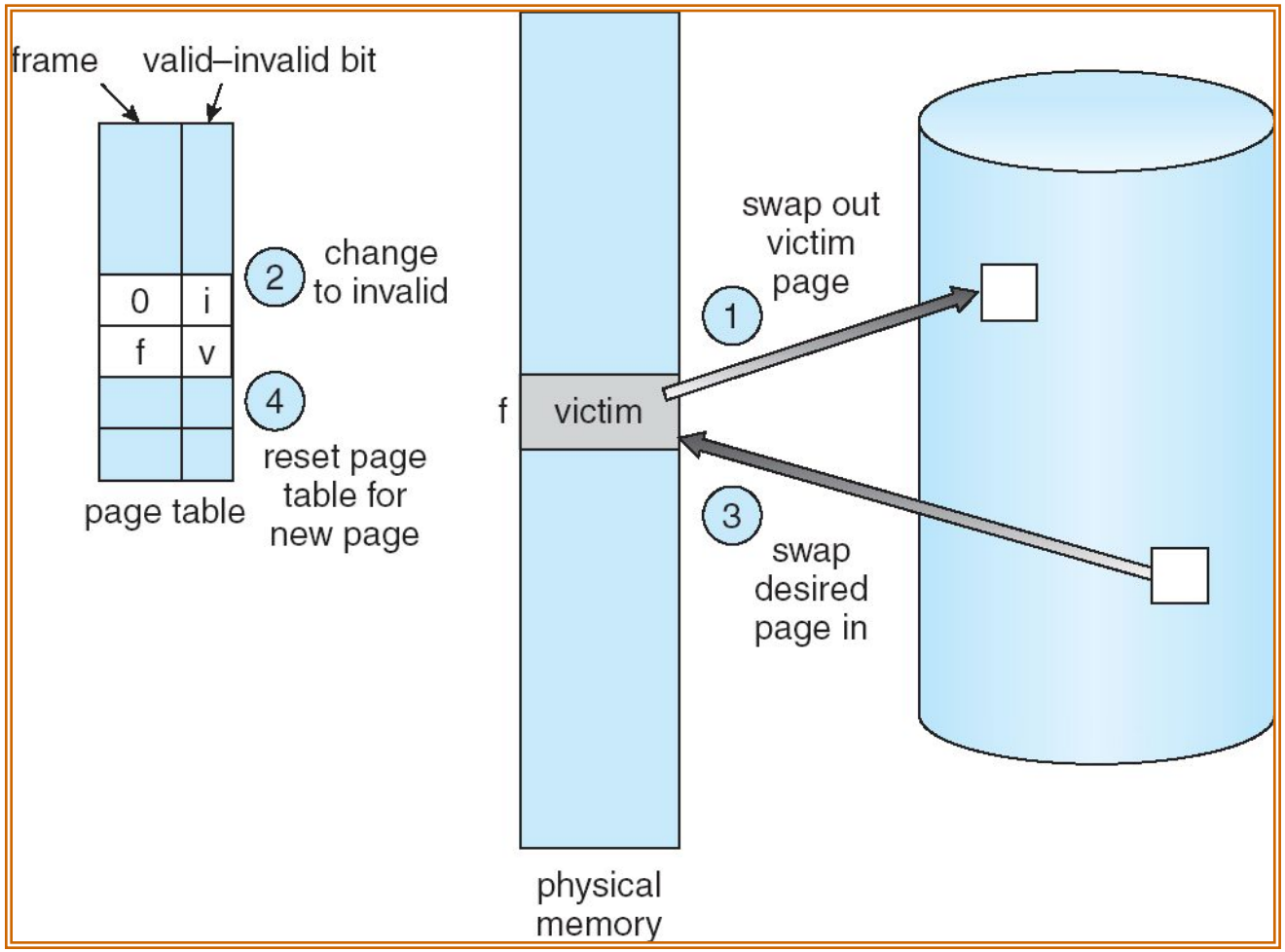
   - If there is a free frame, use it.

   - If there is no free frame, use a page replacement algorithm to select a *victim* frame.

3. Read the desired page into the (newly) free frame. Update the page and frame tables.

4. Restart the process.

Page Replacement



Page Replacement Algorithms

- Want lowest page-fault rate
- Evaluate algorithm by running it on a particular string of memory references (reference string) and computing the number of page faults on that string
- The reference string is

1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5

First-In-First-Out (FIFO) Algorithm

- Reference string: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5
- 3 frames (3 pages can be in memory at a time per process)

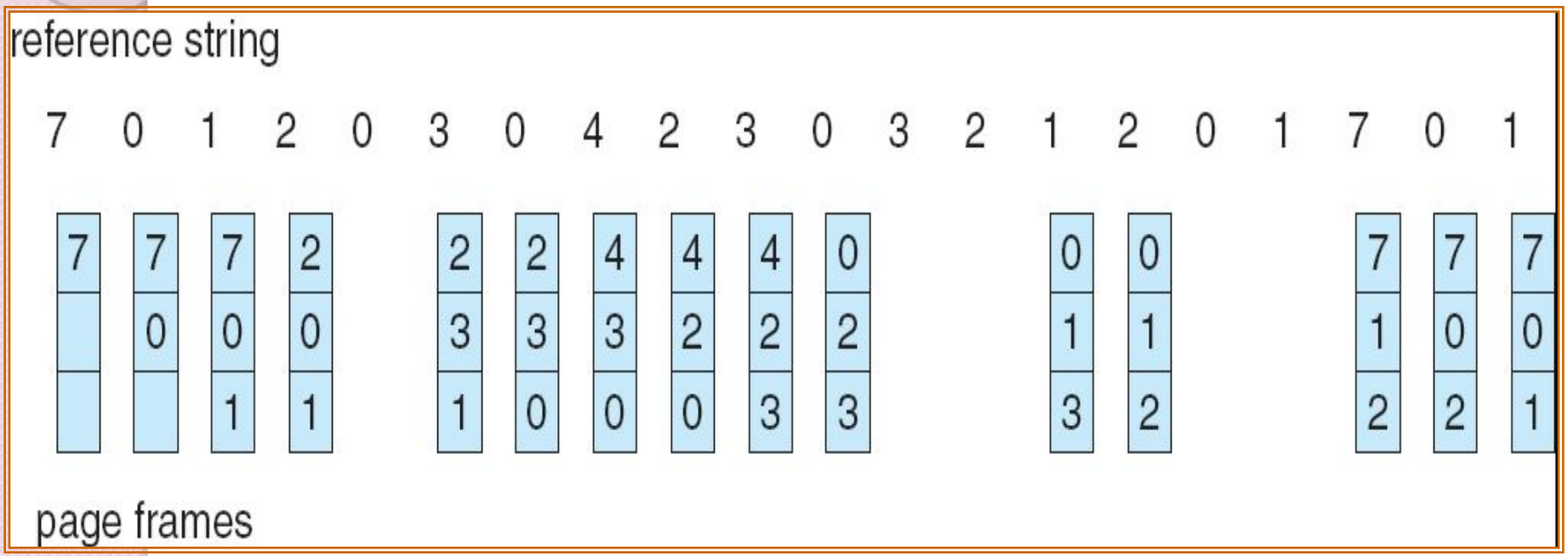
1	1	4	5	
2	2	1	3	9 page faults
3	3	2	4	

- 4 frames

1	1	5	4	
2	2	1	5	10 page faults
3	3	2		
4	4	3		

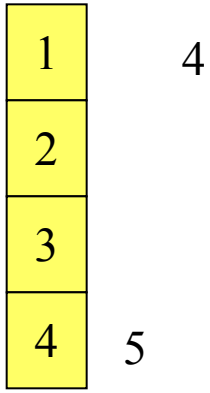
- **Belady's Anomaly:** more frames ⇒ more page faults

FIFO Page Replacement



Optimal Algorithm

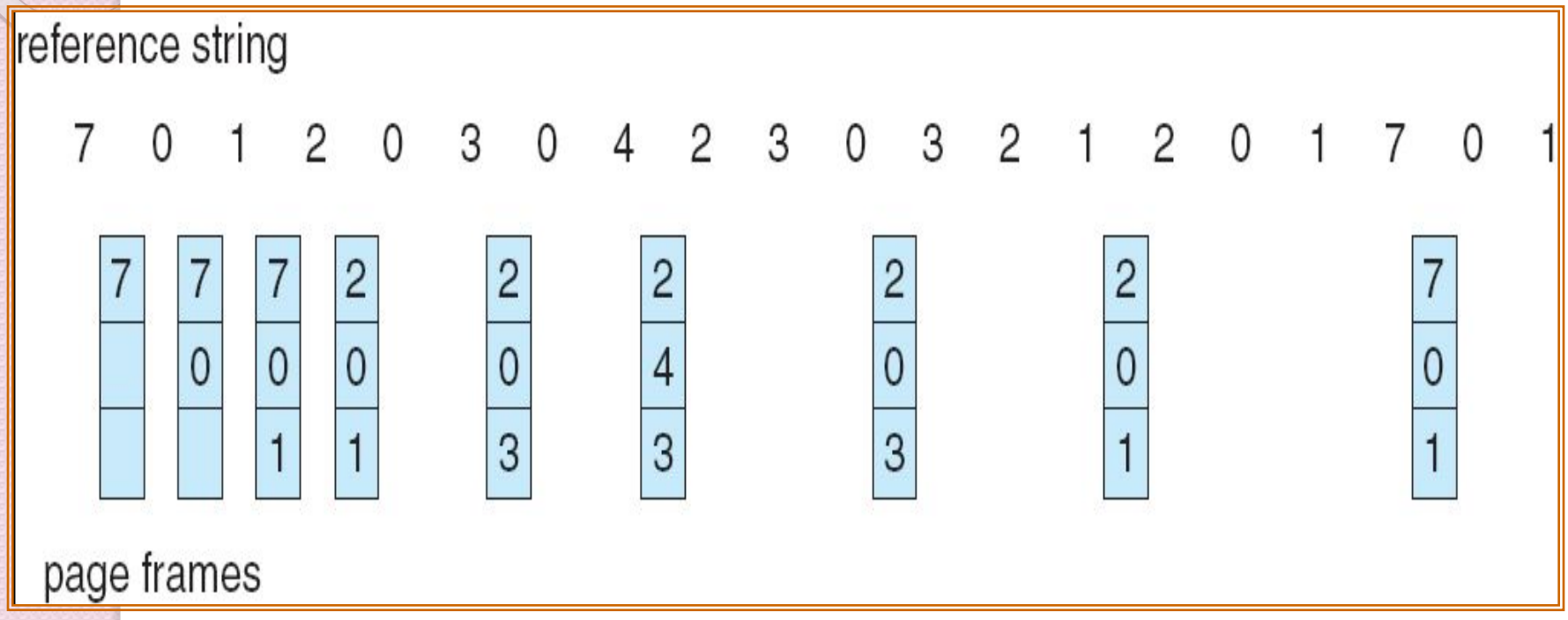
- Replace page that will not be used for longest period of time
- 4 frames example : **1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5**



6 page faults

- How do you know this?
- Used for measuring how well your algorithm performs

Optimal Page Replacement



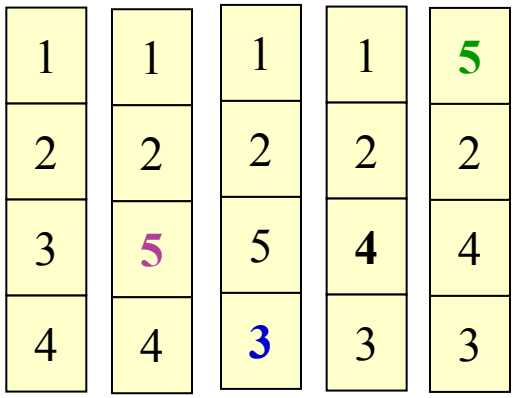
Least Recently Used (LRU) Algorithm

- LRU replacement associates with each page the time of that page last use.
- When a page must be replaced, LRU chooses the page that has not been used for the longest period of time.
- The major problem is how to implement LRU replacement.
- Two implementations are feasible:
    1. counter implementation
    2. stack implementation.

Least Recently Used (LRU) Algorithm

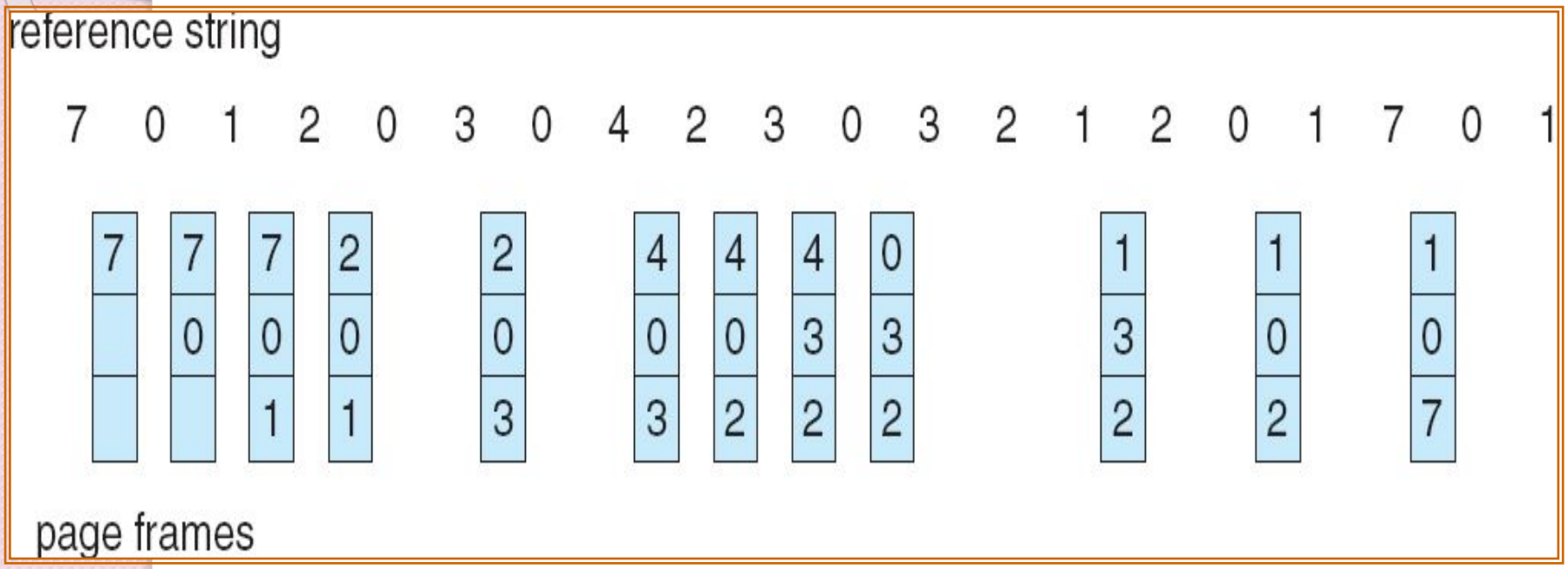
- Reference string:

1, 2, 3, 4, 1, 2, **5**, 1, 2, **3**, **4**, **5**



- Note: both implementation of LRU require hardware support.
- The updating of the clock fields or stack must be done for *every memory reference.*
- If we were to use an interrupt for every reference to allow software to update such data structures, it would slow every memory reference

LRU Page Replacement

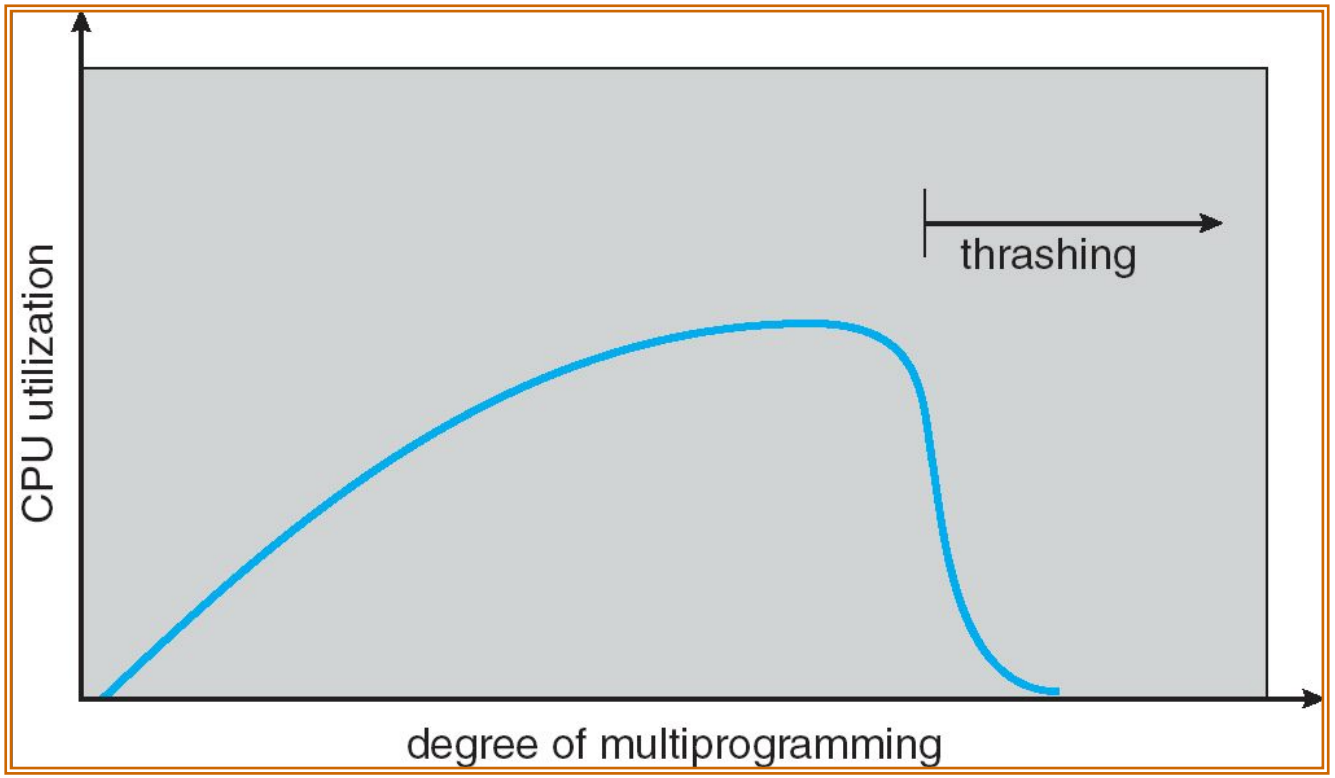


Thrashing : high paging activity

- If a process does not have "enough" pages, the page-fault rate is very high. This leads to:
  - low CPU utilization
  - operating system thinks that it needs to increase the degree of multiprogramming
  - another process added to the system

- **Thrashing** ≡ a process is busy swapping pages in and out (than execution)

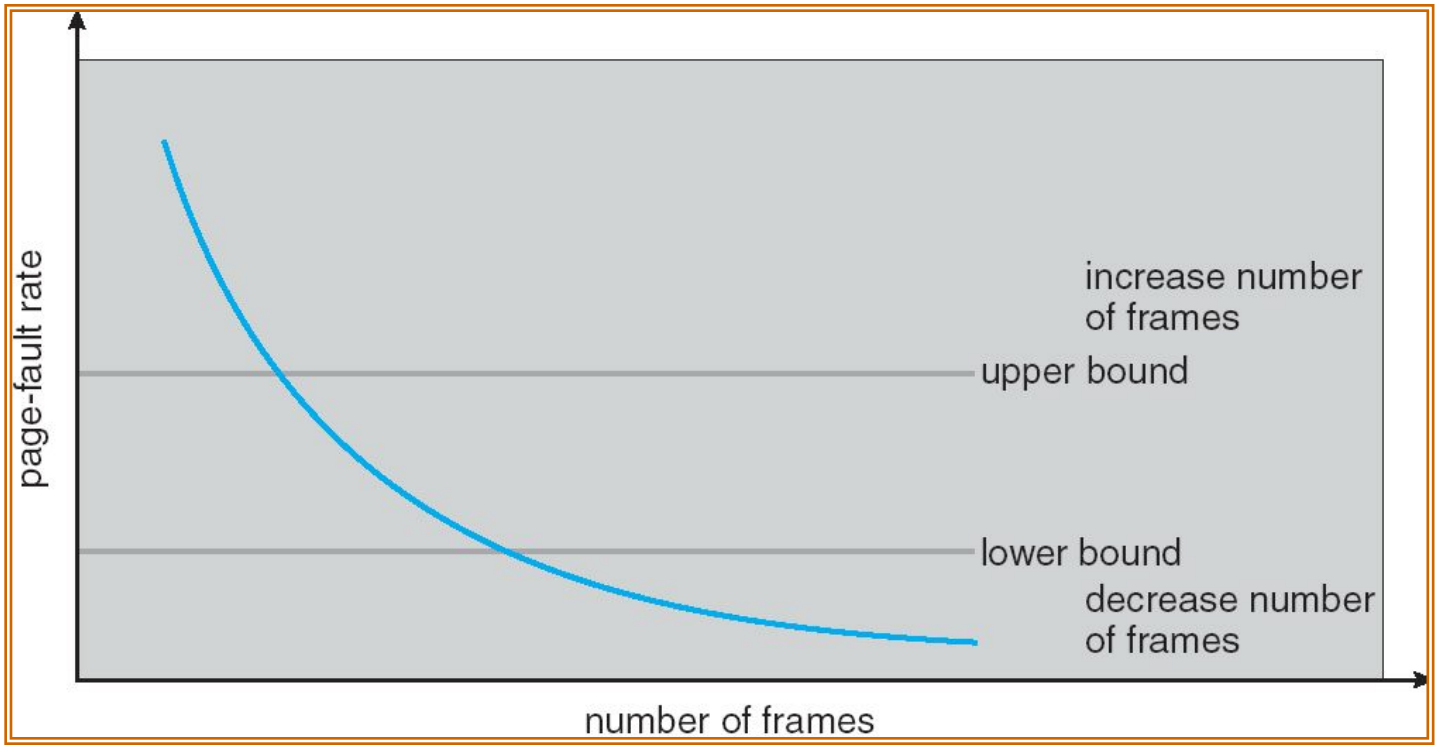
Thrashing (Cont.)



CPU utilization *increases* with degree of multiprogramming slowly, until maximum, if degree of programming increased than that CPU utilization suddenly drops.

Page-Fault Frequency Scheme

- Establish "acceptable" page-fault rate
  - If actual rate too low, process loses frame
  - If actual rate too high, process gains frame



Difference between paging and Segmentation

Paging	Segmentation
A page is a contiguous range of memory addresses which is mapped to physical memory.	A segment is an independent address space. Each segment has addresses in a range from 0 to maximum value.
It has only one linear address space	It has many address spaces
Procedures and data cannot be separated	Procedures and data can be separated
Procedures cannot be shared between users	Procedures can be shared between users
Procedures and data cannot be protected separately	Procedures and data can be protected separately
Compilation cannot be done separately A page is a fixed size	Compilation can be done separately Segment is of arbitrary size.
A page is a physical unit	A segment is a logical unit