Memory Management

Chapter 6

1

1

Chapter Objectives

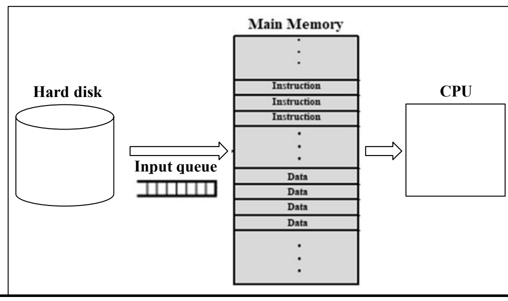
- To provide a detailed description of various ways of organizing memory hardware.
- To discuss various memory-management techniques, including paging and segmentation.
- To describe the benefits of a virtual memory system.
- To explain the concepts of demand paging, page replacement algorithms, and allocation of page frames.

2

2

Background

- 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
- Memory Management - Subdividing memory to accommodate multiple processes
- Memory needs to be allocated to ensure a reasonable supply of ready processes to consume available processor time



3

3

Memory Management Requirements

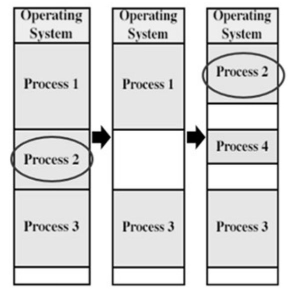
1. Relocation
2. Protection
3. Sharing
4. Logical Organization
5. Physical Organization

4

4

Memory Management Requirements

1. Relocation
   - Programmer does not know where the program will be placed in memory when it is executed
   - While the program is executing, it may be swapped to disk and returned to main memory at a different location (relocated)
   - Memory references must be translated in the code to actual physical memory address



5

5

Addresssing Requirements for a Process

- OS must know the locations/addresses of:
  - Process control info
  - Execution stack
  - Entry point to begin execution
- Processor must deal with memory references within program:
  - Branch instructions contain an address to reference the instruction to be executed next
  - Data reference instructions contain the address of the byte/word referenced
- CPU & OS s/w must be able to translate memory refs in the code → into actual physical memory addresses (its location in MM)

6

6

Memory Management Requirements

2. 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 runtime
   - Memory protection requirement must be satisfied by the processor (hardware) rather than the operating system (software)
     - Operating system cannot anticipate all of the memory references a program will make

7

7

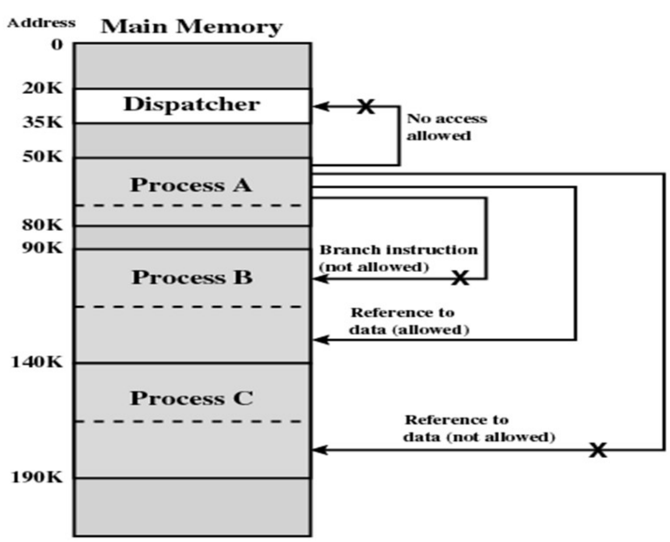


Figure 8.14 Protection Relationships Between Segments

8

8

Memory Management Requirements

3. 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

9

9

Memory Management Requirements

4. Logical Organization
   - 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

10

10

Memory Management Requirements

5. Physical Organization
   - Memory available for a program plus its data may be insufficient
     - Overlaying allows various modules to be assigned the same region of memory
   - Programmer does not know how much space will be available

11

11

Memory Partitioning

- Fixed partitioning
  - Equal-size partitions
  - Unequal-sized partitions
- Dynamic partitioning

12

12

Fixed Partitioning

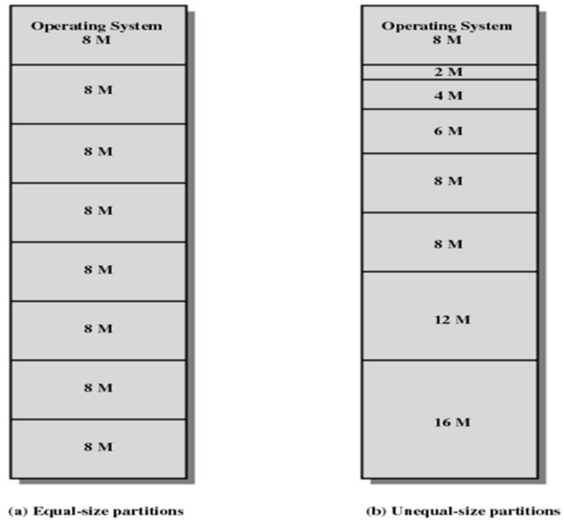


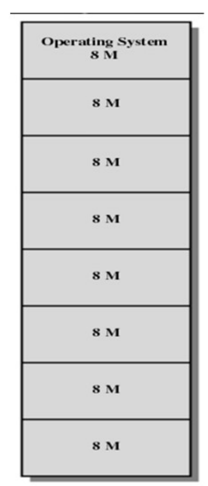
Figure 7.2 Example of Fixed Partitioning of a 64-Mbyte Memory

13

13

Fixed Partitioning

- Equal-size partitions
  - Any process whose size is less than or equal to the partition size can be loaded into an available partition
  - If all partitions are full, the operating system can swap a process out of a partition
  - A program may not fit in a partition. The programmer must design the program with overlays

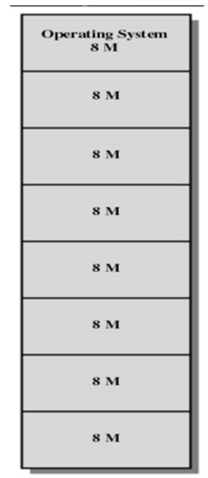


14

14

Fixed Partitioning

- Main memory use is inefficient. Any program, no matter how small, occupies an entire partition. This is called internal fragmentation.
  – E.g: A program of size 2MB occupies an 8MB partition → wasted space internal to a partition, as the data loaded is smaller than the partition size.
- Equal-size partitions
  – Because all partitions are of equal size, it does not matter which partition is used

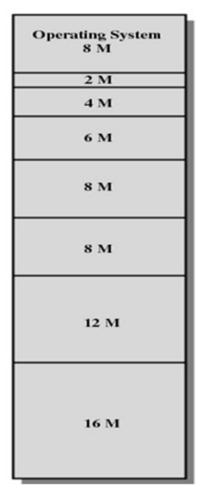


15

15

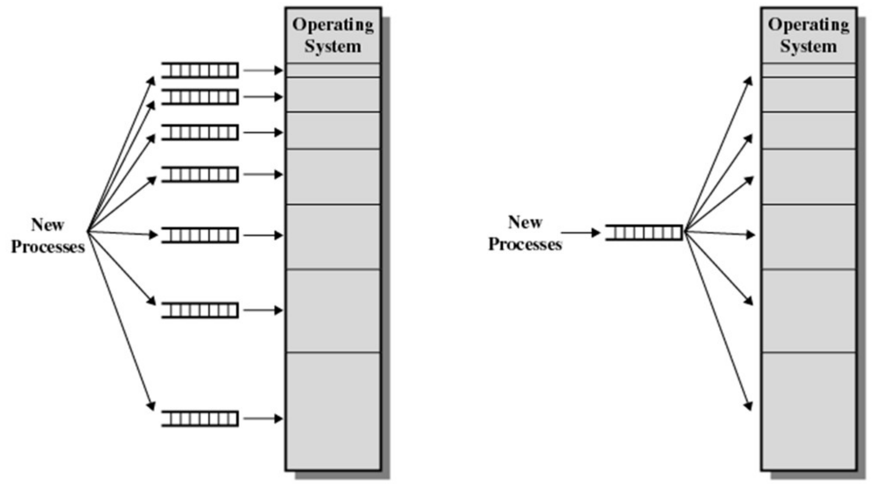
Placement Algorithm with Partitions

- Unequal-size partitions (also fixed)
  – 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



16

16



(a) One process queue per partition

(b) Single queue

Figure 7.3 Memory Assignment for Fixed Partitioning

17

17

Dynamic Partitioning

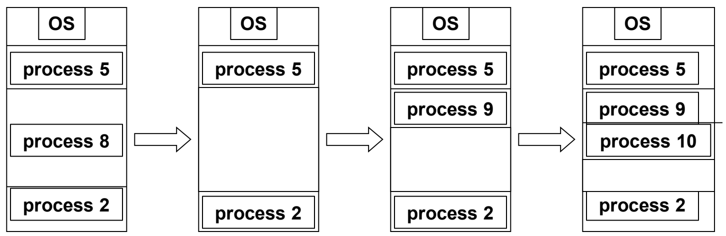
- Partitions are of variable length and number
- Process is allocated exactly as much memory as required
- Eventually get holes in the memory. This is called external fragmentation
- Must use compaction to shift processes so they are contiguous and all free memory is in one block

18

18

Contiguous 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)



19

19

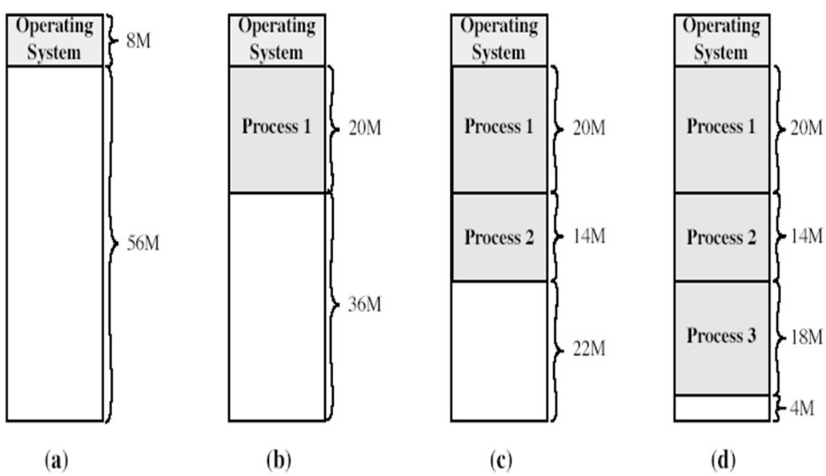


Figure 7.4 The Effect of Dynamic Partitioning

20

20

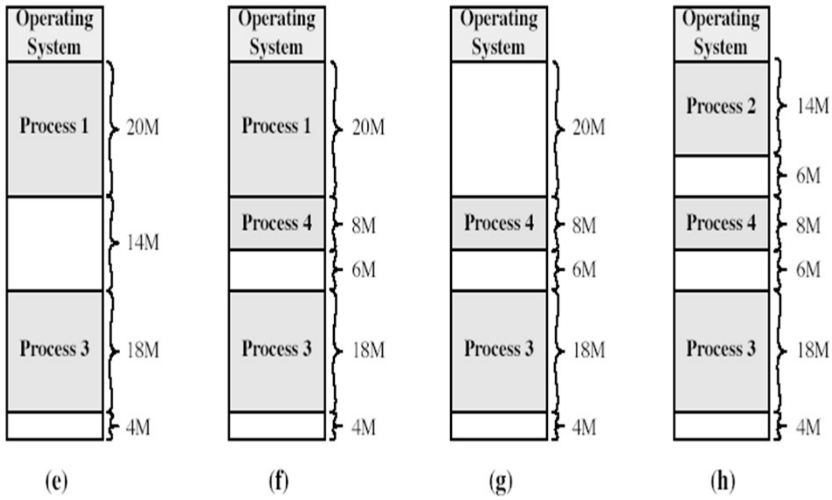


Figure 7.4 The Effect of Dynamic Partitioning

21

21

Dynamic Partitioning

Difficulty with compaction:

- It is a time-consuming procedure – wasteful of processor time.
- Therefore, needs dynamic relocation capability, i.e. it must be possible to move a program from one region to another (in MM) without invalidating the memory references in the program.

22

22

Dynamic Partitioning Placement Algorithms

- Operating system must decide which free block to allocate to a process
  - Best-fit
  - Worst-fit
  - First-fit
  - Next-fit
- First-fit and best-fit better than worst-fit in terms of speed and storage utilization

23

23

Best-fit algorithm

- Allocate the *smallest* hole that is big enough; must search entire list, unless ordered by size. Produces the smallest leftover hole.
- 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

24

24

Worst-fit algorithm

- Allocate the *largest* hole; must also search entire list.
- Produces the largest leftover hole. Worst performer overall

25

25

First-fit algorithm

- Allocate the *first* hole that is big enough
- 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

26

26

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

27

27

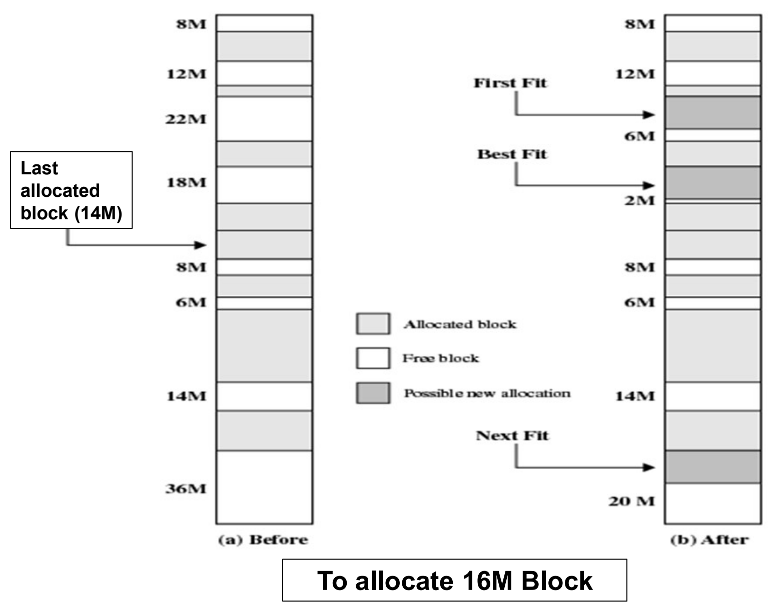
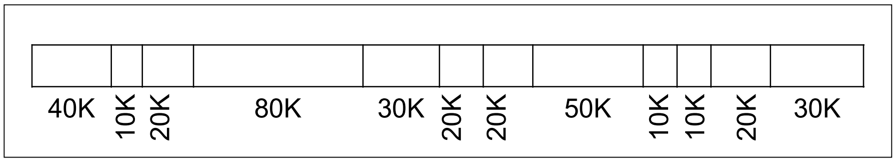


Figure 7.5 Example Memory Configuration Before and After Allocation of 16 Mbyte Block

28

Example

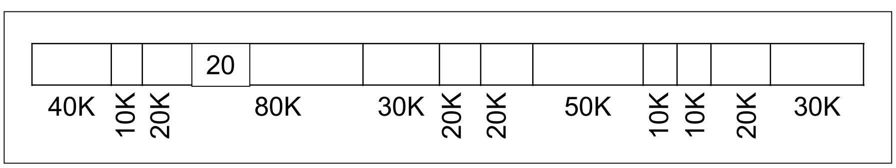


- The shaded areas are allocated blocks; the white areas are free blocks.
- The next FOUR memory requests are 20K, 50K, 10K and 30K (loaded in that order).
- Using the following placement algorithms, show the partition allocated for the requests.
  - First-fit
  - Best-fit
  - Next-fit

29

29

Example: First-fit

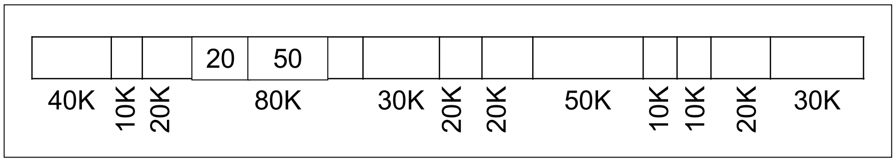


- 20K, 50K, 10K and 30K (in that order).
  - Allocate for 20K

30

30

Example: First-fit

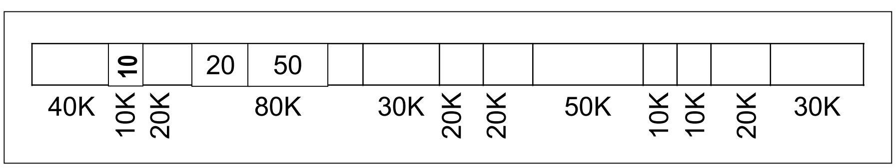


- 20K, 50K, 10K and 30K (in that order).
  - Allocate for 20K
  - Allocate for 50K

31

31

Example: First-fit

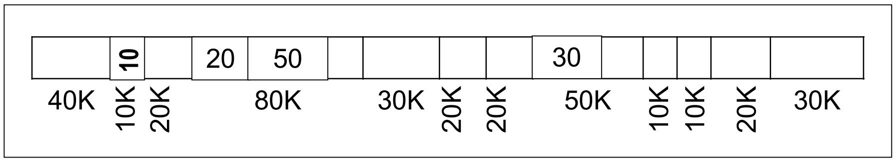


- 20K, 50K, 10K and 30K (in that order).
  - Allocate for 20K
  - Allocate for 50K
  - Allocate for 10K

32

32

Example: First-fit

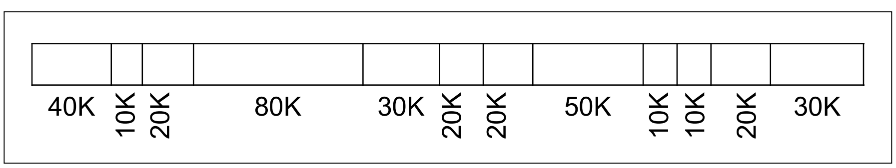


- 20K, 50K, 10K and 30K (in that order).
  - Allocate for 20K
  - Allocate for 50K
  - Allocate for 10K
  - Allocate for 30K

33

33

Example: Best-fit

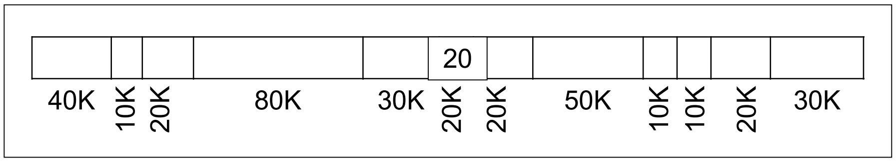


- 20K, 50K, 10K and 30K (in that order).

34

34

Example: Best-fit

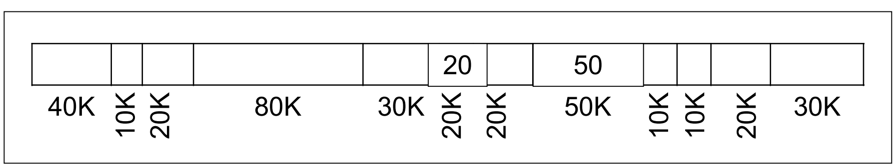


- 20K, 50K, 10K and 30K (in that order).
  - Allocate for 20K

35

35

Example: Best-fit

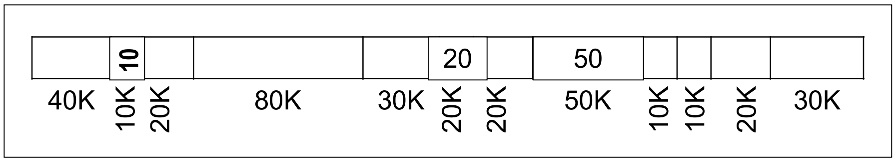


- 20K, 50K, 10K and 30K (in that order).
  - Allocate for 20K
  - Allocate for 50K

36

36

Example: Best-fit

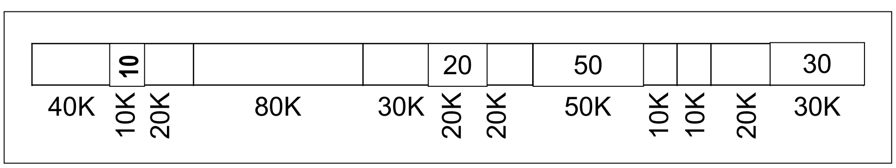


- 20K, 50K, 10K and 30K (in that order).
  - Allocate for 20K
  - Allocate for 50K
  - Allocate for 10K

37

37

Example: Best-fit

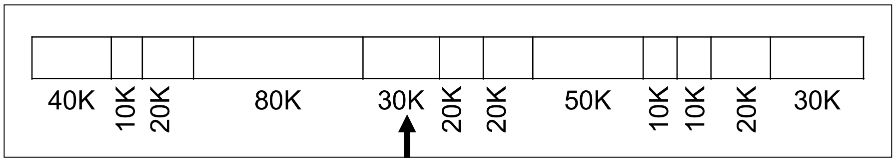


- 20K, 50K, 10K and 30K (in that order).
  - Allocate for 20K
  - Allocate for 50K
  - Allocate for 10K
  - Allocate for 30K

38

38

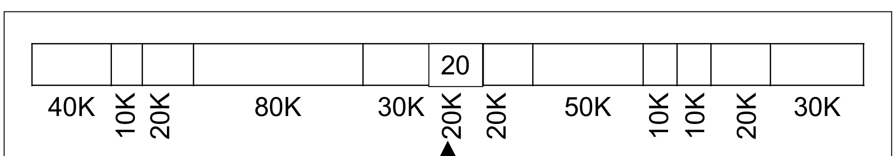
Example: Next-fit



most recently added block

- 20K, 50K, 10K and 30K (in that order).

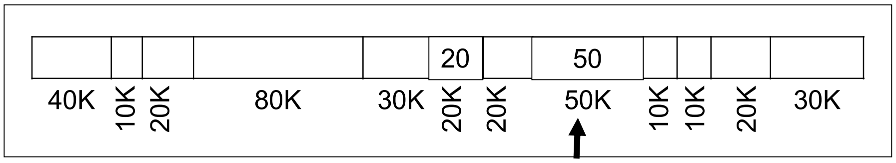
Example: Next-fit



most recently added block

- 20K, 50K, 10K and 30K (in that order).
  - Allocate for 20K

Example: Next-fit



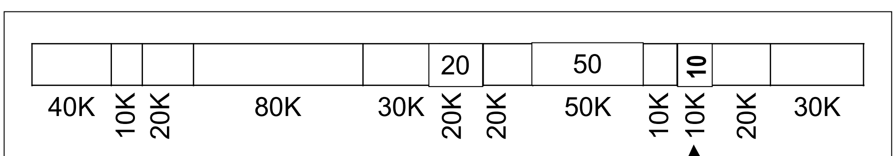
most recently added block

- 20K, 50K, 10K and 30K (in that order).
  - Allocate for 20K
  - Allocate for 50K

41

41

Example: Next-fit



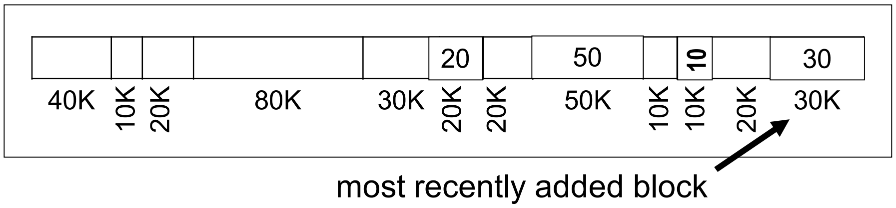
most recently added block

- 20K, 50K, 10K and 30K (in that order).
  - Allocate for 20K
  - Allocate for 50K
  - Allocate for 10K

42

42

Example: Next-fit

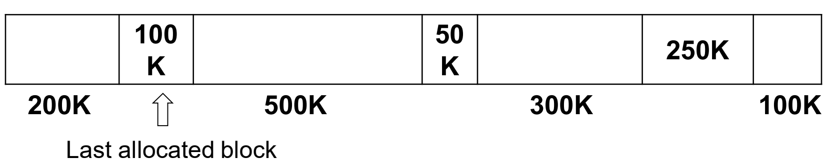


- 20K, 50K, 10K and 30K (in that order).
  - Allocate for 20K
  - Allocate for 50K
  - Allocate for 10K
  - Allocate for 30K

43

43

Exercise



Show the partition allocated for the processes of size 110K, 150K, 300K and 200K (loaded in that order):

- First-fit
- Best-fit
- Next-fit

44

44

Buddy System

- Entire space available is treated as a single block of 2^U
- If a request of size s such that $2^{U-1} < s <= 2^U$, 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

45

45

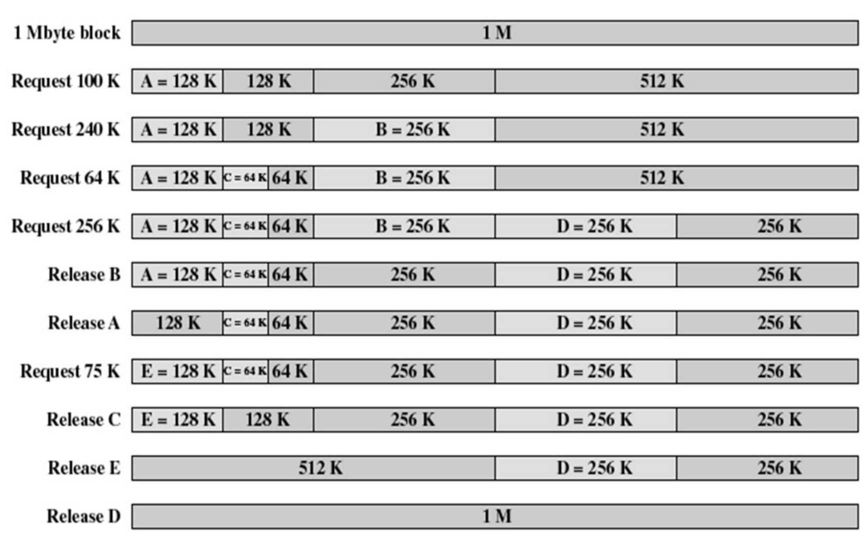


Figure 7.6 Example of Buddy System

46

46

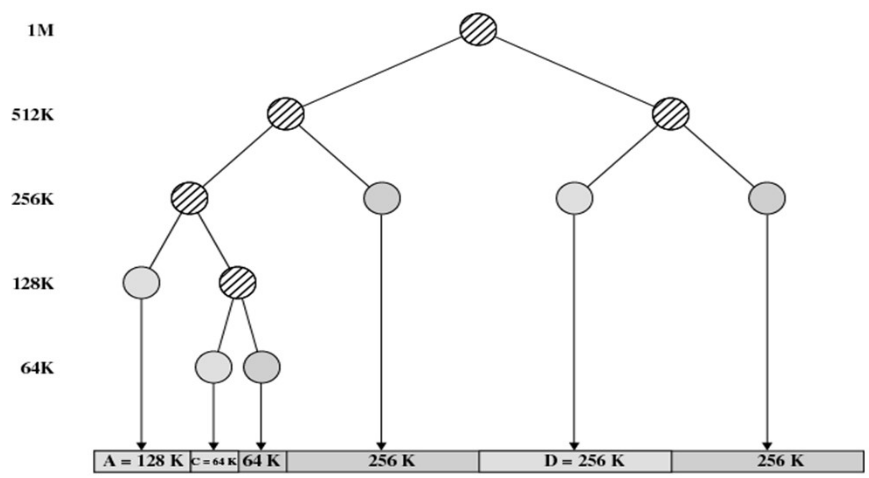


Figure 7.7 Tree Representation of Buddy System

47

47

Example

If a memory size is 240MB. By using the buddy partitioning scheme, shade the memory insertion for the memory space request as follows:

- Request A = 30MB,
- Request B = 50MB,
- Request C = 60MB,
- Release A,
- Request D = 40MB,
- Release B,
- Release C, and
- Release D.

48

48

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 (from swapping)
- Compaction will also cause a program to occupy a different partition which means different absolute memory locations

49

49

Addresses

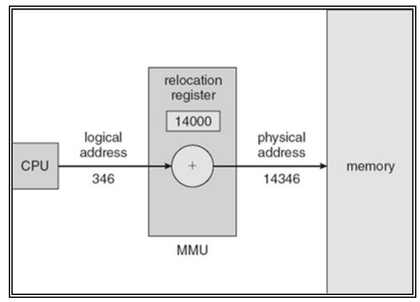
- Logical
  - Reference to a memory location independent of the current assignment of data to memory
  - Translation must be made to the physical address
  - **Logical address** – generated by the CPU; also referred to as *virtual address*
- Relative
  - Address expressed as a location relative to some known point
- Physical
  - The absolute address or actual location in main memory
  - **Physical address** – address seen by the memory unit

50

50

Memory-Management Unit (MMU)

- 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



51

51

Registers Used during Execution

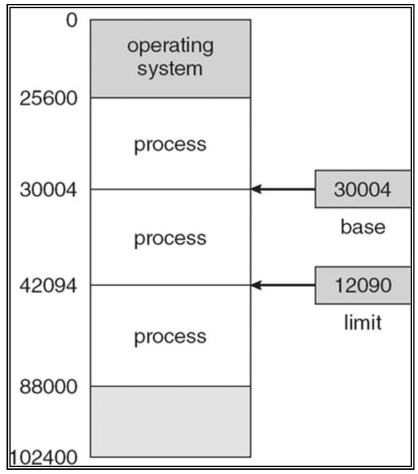
- Base register
  - Starting address for the process
- Limit register
  - Ending location of the process
- These values are set when the process is loaded or when the process is swapped in

52

52

Dynamic relocation using a relocation register

- 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 limit register
- If the address is not within limit, an interrupt is generated to the OS

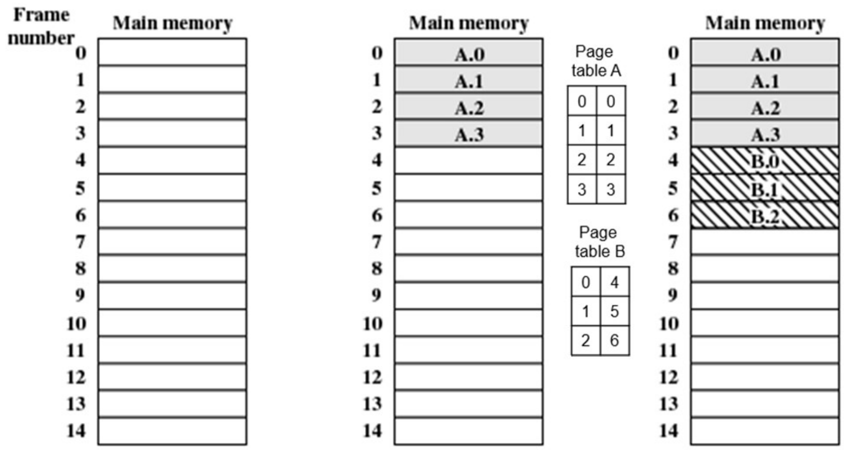


A base and a limit register define a logical address space

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 <u>pages</u> and chunks of memory are called <u>frames</u>
- 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

Assignment of Process Pages to Free Frames

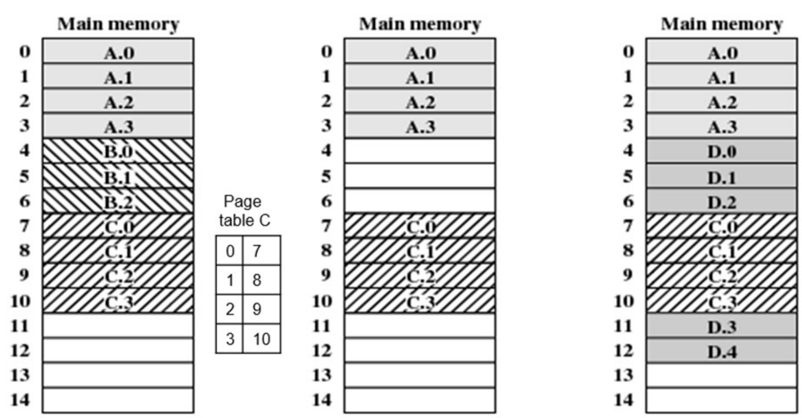


(a) Fifteen Available Frames (b) Load Process A (c) Load Process B

55

55

Assignment of Process Pages to Free Frames

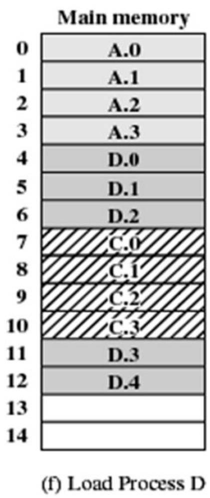


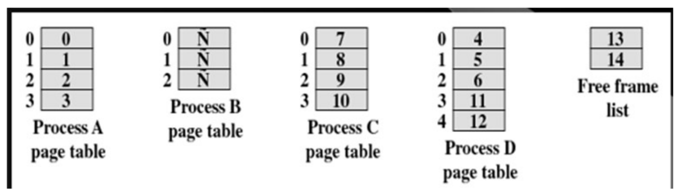
(d) Load Process C (e) Swap out B (f) Load Process D

Figure 7.9 Assignment of Process Pages to Free Frames

56

Page Tables for Example

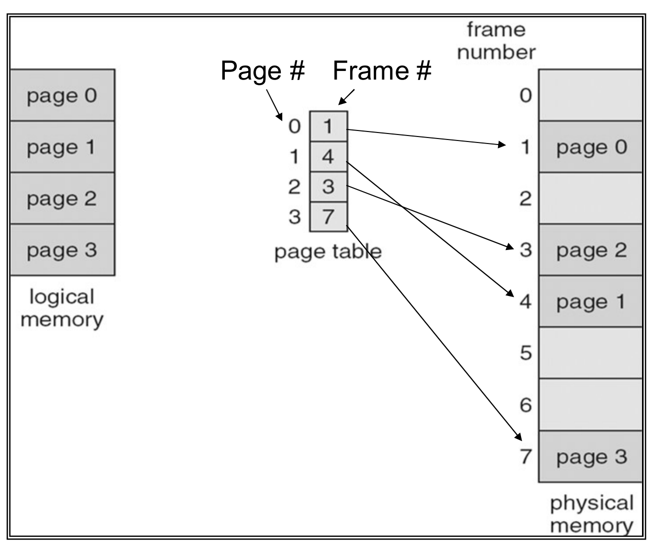




57

57

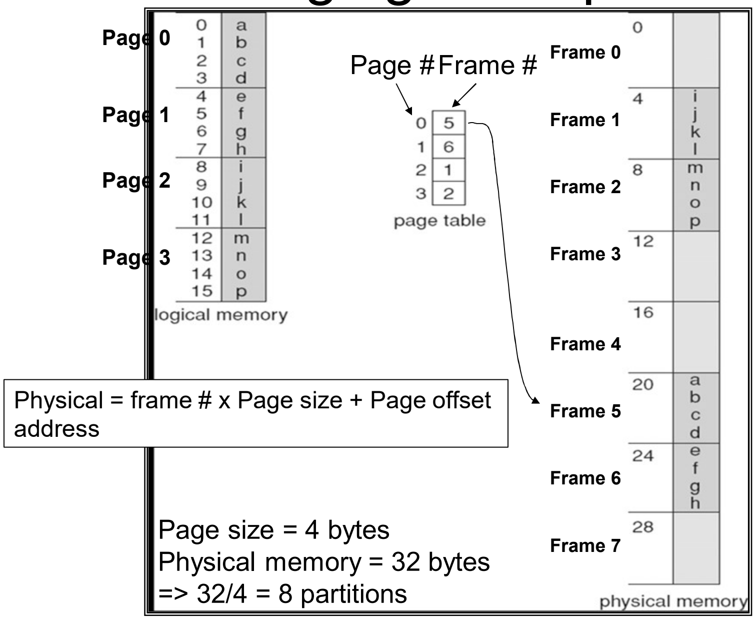
Paging Example



58

58

Paging Example



59

59

Example

- Memory address consist of a page number and offset within the page
- Consider the following page table. The page size is 8 bytes each.
- What are the physical addresses for the following logical addresses?

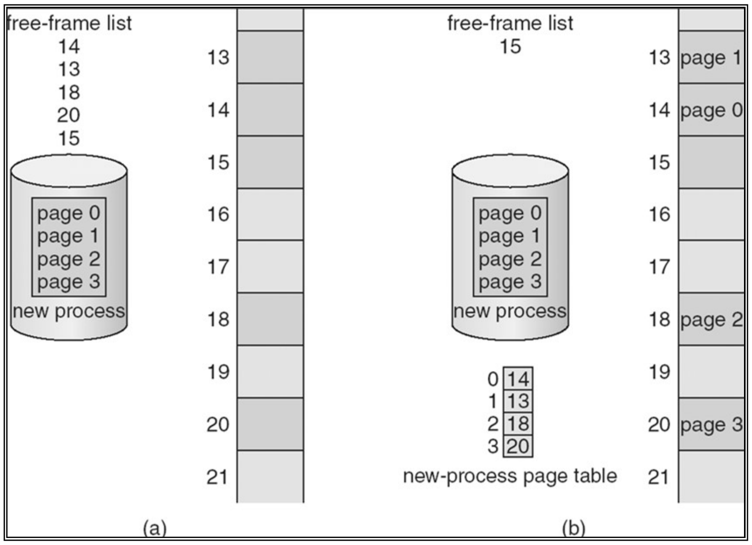
1. (1, 5)
2. (2, 0)
3. (0, 6)
4. (3, 8)

Page	Frame
0	6
1	2
2	0
3	3
4	5

60

60

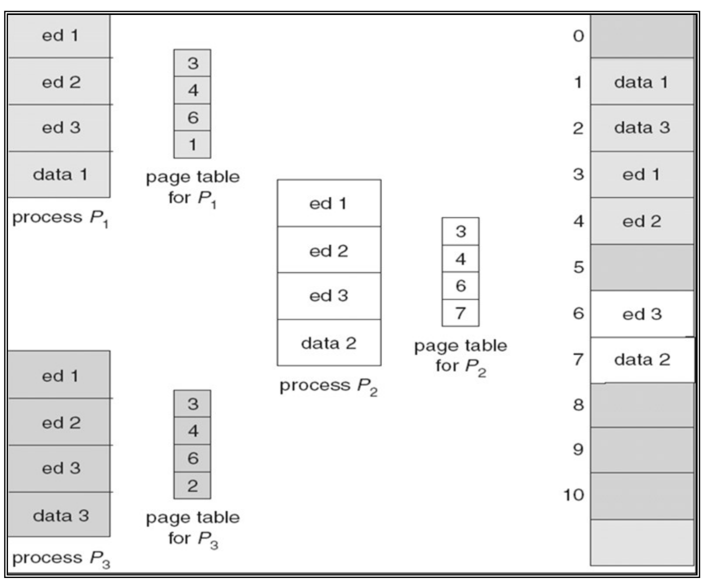
Free Frames



61

61

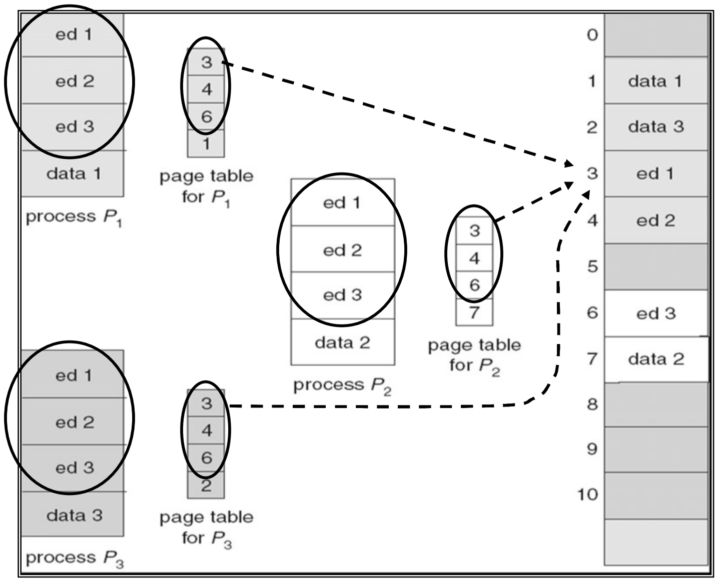
Shared Pages Example



62

62

Shared Pages Example



63

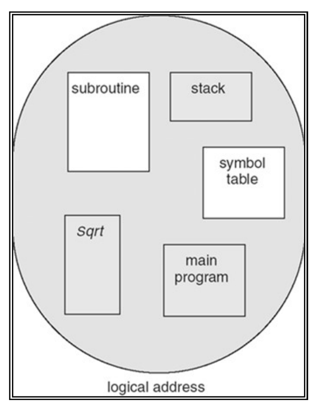
63

Segmentation

- Memory-management scheme that supports user view of memory
- A program is a collection of segments. A segment is a logical unit such as:

main program,
procedure,
function,
method,
object,
arrays

local variables,
global variables,
common block,
stack,
symbol table,

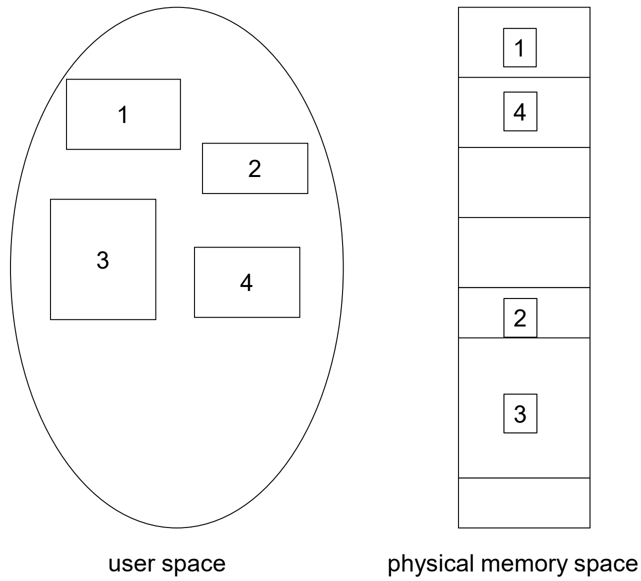


User's View of a Program

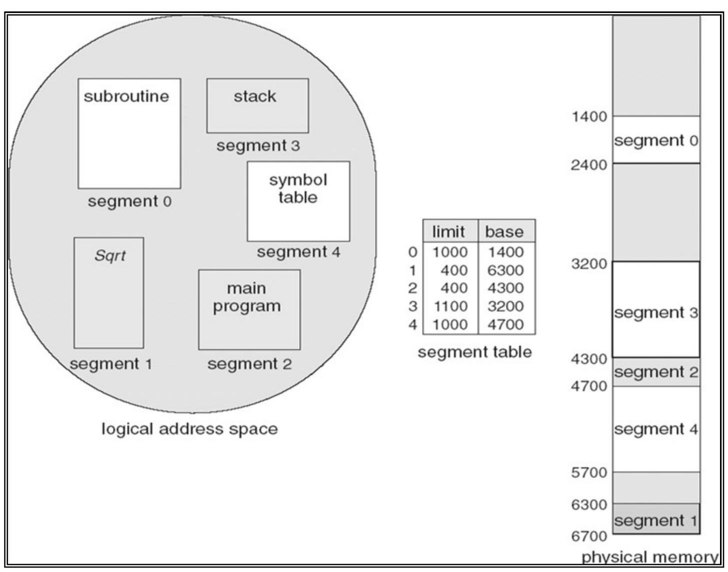
64

64

Logical View of Segmentation



Example of Segmentation



Example

- Addressing consist of two parts: a segment number and an offset
- Consider the following segment table.

Segment	Base	Length
0	660	248
1	1752	422
2	222	198
3	996	604

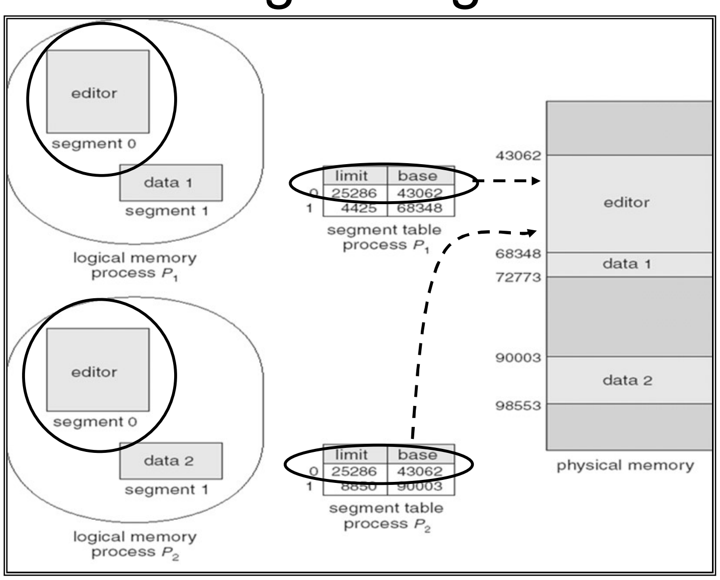
What are the physical addresses for the following logical addresses?

1. (0, 198)
2. (1, 530)
3. (2, 156)
4. (3, 444)
5. (0, 222)

67

67

Sharing of Segments



68

68

Segmentation

- All segments of all programs do not have to be of the same length
- There is a maximum segment length
- Addressing consist of two parts - a segment number and an offset
- Since segments are not equal, segmentation is similar to dynamic partitioning

69

69

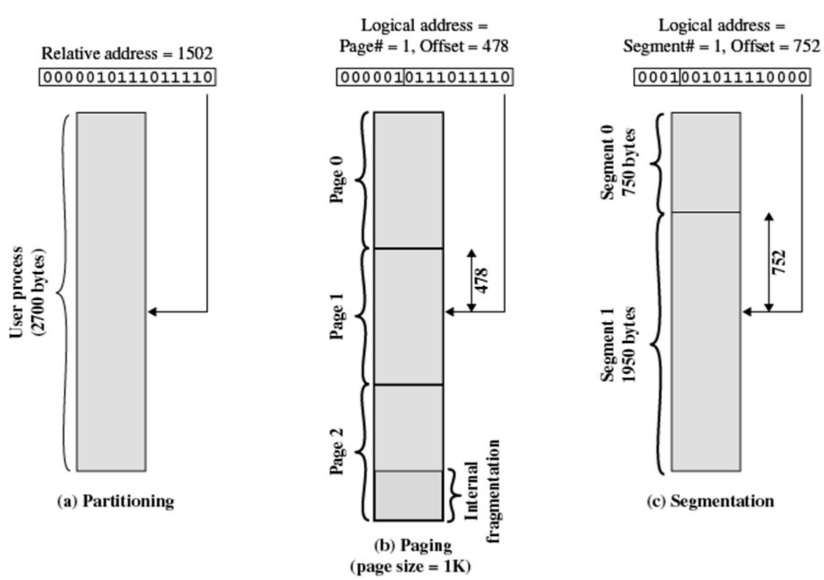
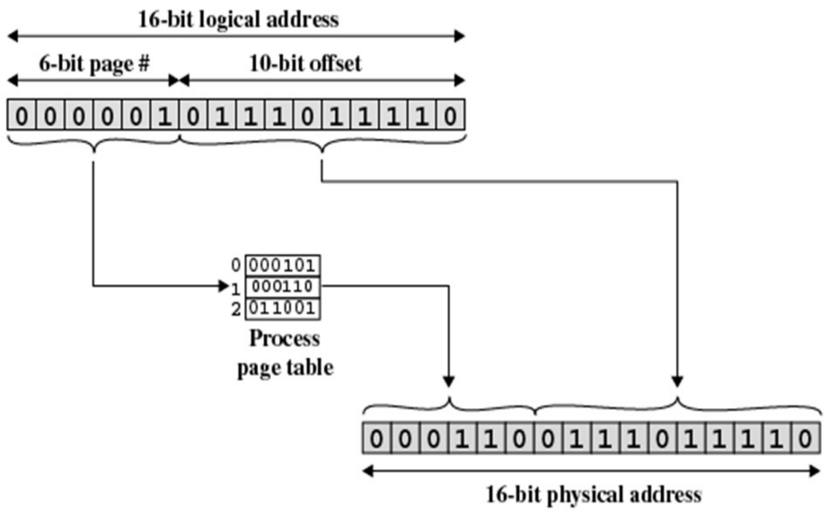


Figure 7.11 Logical Addresses

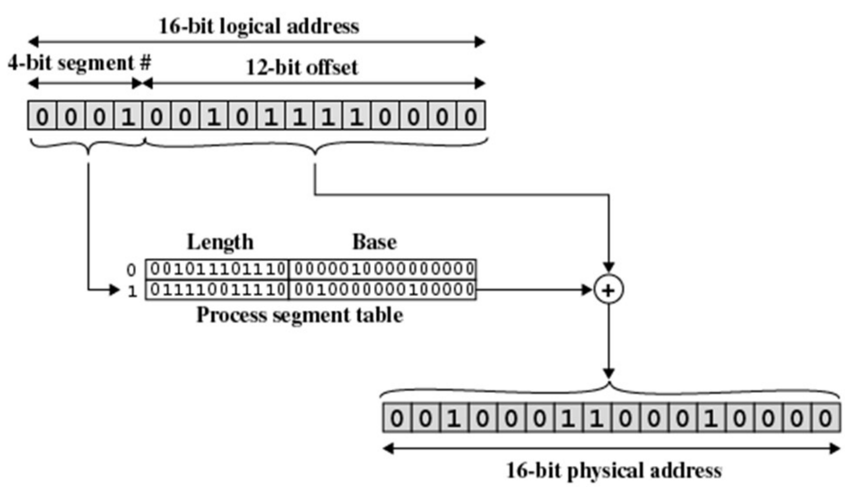
70



(a) Paging

71

71



(b) Segmentation

Figure 7.12 Examples of Logical-to-Physical Address Translation

72

72

- Consider a simple paging system which uses 16 bits addressing scheme. The page size is 1K each. Given a process page table as follows,

0	000101
1	000111
2	001001
3	001011
4	001100

- Determine the physical address of the following logical address for this scheme.
- 0001000001111110
- 0000111111111111

73

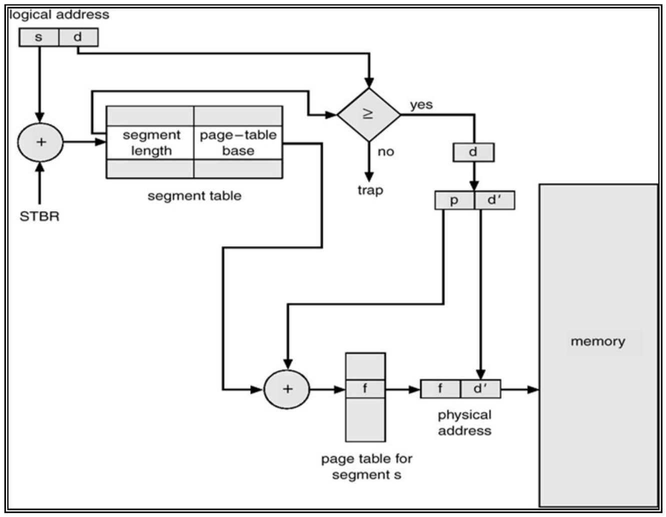
- Consider a simple segment system which uses 16 bits addressing scheme. The segment size is 1K each. Given a process segment table as follows,

Segment	**Length**	**Base**
0	11110 11111	0000000010100000
1	11100 11111	0000000001111000
2	11111 00000	0000000010011010
3	10000 00000	0000000010010000
4	11111 11111	0000000010101000

- Determine the physical address of the following logical address for this scheme.
- 0000010111100000
- 0001001110001111

74

MULTICS Address Translation Scheme



75

75

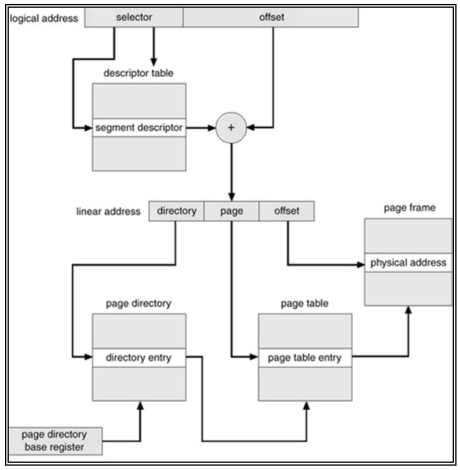
Segmentation with Paging – Intel 386

- As shown in the following diagram, the Intel 386 uses segmentation with paging for memory management with a two-level paging scheme

76

76

Intel 30386 Address Translation



77

77

Linux on Intel 80x86

- Uses minimal segmentation to keep memory management implementation more portable
- Uses 6 segments:
  - Kernel code
  - Kernel data
  - User code (shared by all user processes, using logical addresses)
  - User data (likewise shared)
  - Task-state (per-process hardware context)
  - LDT
- Uses 2 protection levels:
  - Kernel mode
  - User mode

78

78

Virtual Memory

79

79

Hardware and Control Structures

- Memory references are dynamically translated into physical addresses at run time
  - A process may be swapped in and out of main memory such that it occupies different regions
- A process may be broken up into pieces that do not need to be located contiguously in main memory
- All pieces of a process do not need to be loaded in main memory during execution

80

80

Execution of a Program

- Operating system brings into main memory a few pieces of the program
- **Resident set** - portion of process that is in main memory
- An interrupt is generated when an address is needed that is not in main memory ~ **memory fault**
- Operating system places the process in a **blocking state**
- Piece of process that contains the logical address is brought into main memory
  - Operating system issues a disk I/O Read request
  - Another process is dispatched to run while the disk I/O takes place
  - An interrupt is issued when disk I/O complete which causes the operating system to place the affected process in the **Ready state**

81

81

Advantages of Breaking up a Process

- More processes may be maintained in main memory
  - Only load in some of the pieces of each process
  - With so many processes in main memory, it is very likely a process will be in the Ready state at any particular time
- A process may be larger than all of main memory

82

82

Types of Memory

- Real memory
  - Main memory
- Virtual memory
  - Memory on disk
  - Allows for effective multiprogramming and relieves the user of tight constraints of main memory

83

83

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

84

84

Characteristic of Paging and Segmentation

Simple Paging	Virtual Memory Paging	Simple Segmentation	Virtual Memory Segmentation
Main memory partitioned into small fixed-size chunks called frames	Main memory partitioned into small fixed-size chunks called frames	Main memory not partitioned	Main memory not partitioned
Program broken into pages by the compiler or memory management system	Program broken into pages by the compiler or memory management system	Program segments specified by the programmer to the compiler (i.e., the decision is made by the programmer)	Program segments specified by the programmer to the compiler (i.e., the decision is made by the programmer)
Internal fragmentation within frames	Internal fragmentation within frames	No internal fragmentation	No internal fragmentation
No external fragmentation	No external fragmentation	External fragmentation	External fragmentation
Operating system must maintain a page table for each process showing which frame each page occupies	Operating system must maintain a page table for each process showing which frame each page occupies	Operating system must maintain a segment table for each process showing the load address and length of each segment	Operating system must maintain a segment table for each process showing the load address and length of each segment
Operating system must maintain a free frame list	Operating system must maintain a free frame list	Operating system must maintain a list of free holes in main memory	Operating system must maintain a list of free holes in main memory
Processor uses page number, offset to calculate absolute address	Processor uses page number, offset to calculate absolute address	Processor uses segment number, offset to calculate absolute address	Processor uses segment number, offset to calculate absolute address
All the pages of a process must be in main memory for process to run, unless overlays are used	Not all pages of a process need be in main memory frames for the process to run. Pages may be read in as needed	All the segments of a process must be in main memory for process to run, unless overlays are used	Not all segments of a process need be in main memory frames for the process to run. Segments may be read in as needed
	Reading a page into main memory may require writing a page out to disk		Reading a segment into main memory may require writing one or more segments out to disk

85

Demand Paging

- Bring a page into memory only when it is needed
  - Less I/O needed
  - Less memory needed
  - Faster response
  - More users
- Page is needed $\Rightarrow$ reference to it
  - invalid reference $\Rightarrow$ abort
  - not-in-memory $\Rightarrow$ bring to memory

86

86

Paging

- Each process has its own page table
- Each page table entry contains the frame number of the corresponding page in main memory
- A bit is needed to indicate whether the page is in main memory or not – present (P) bit

(a) Paging only

87

87

Modify Bit in Page Table

- Modify (M) bit is needed to indicate if the page has been altered since it was last loaded into main memory
- A.k.a dirty bit
- If no change has been made, the page does not have to be written to the disk when it needs to be swapped out
- To reduce overhead of page transfers – only modified pages are written to disk

88

88

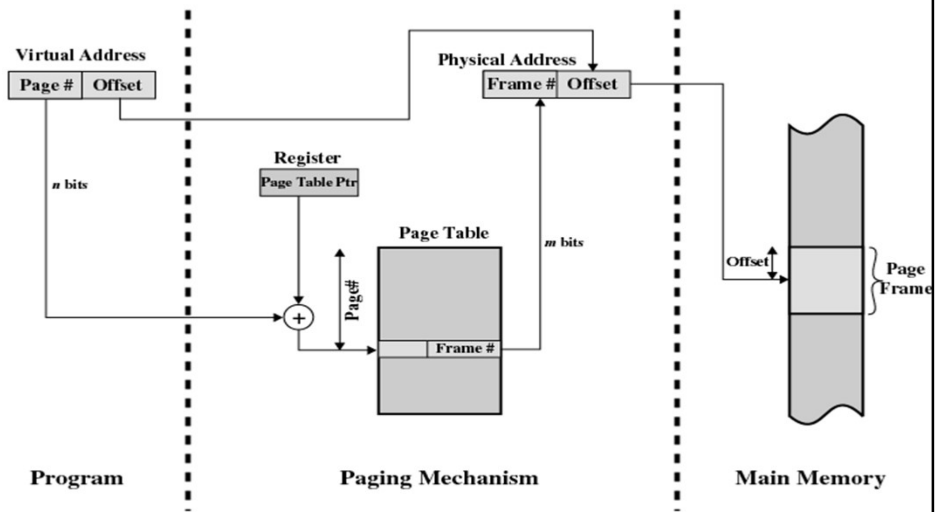


Figure 8.3 Address Translation in a Paging System

89

89

Page Tables

- The entire page table may take up too much main memory
- Page tables are also stored in virtual memory
- When a process is running, part of its page table is in main memory

90

90

Inverted Page Table

- Used on PowerPC, UltraSPARC, and IA-64 architecture
- Page number portion of a virtual address is mapped into a hash value
- Hash value points to inverted page table
- Fixed proportion of real memory is required for the tables regardless of the number of processes
- Page number
- Process identifier
- Control bits
- Chain pointer

91

91

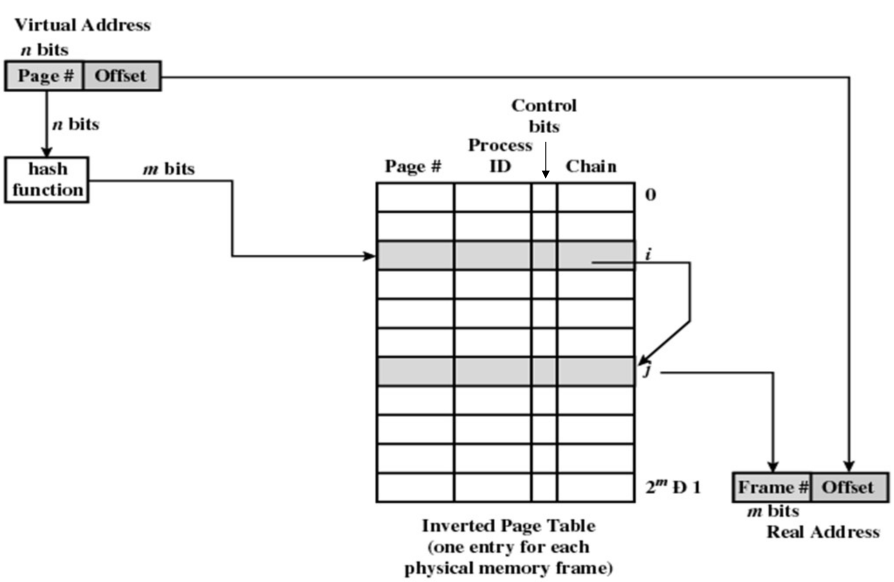


Figure 8.6 Inverted Page Table Structure

92

92

Page Size

- Smaller page size,
  - less amount of internal fragmentation,
  - more pages required per process
- More pages per process means larger page tables, so large portion of page tables in virtual memory
- Secondary memory is designed to efficiently transfer large blocks of data so a large page size is better
- Small page size, large number of pages will be found in main memory
- As time goes on during execution, the pages in memory will all contain portions of the process near recent references. Page faults low.
- Increased page size causes pages to contain locations further from any recent reference. Page faults rise.

93

93

Segmentation

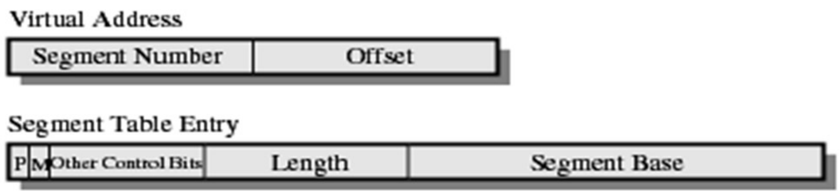
- May be unequal, dynamic size
- Simplifies handling of growing data structures
- Allows programs to be altered and recompiled independently
- Lends itself to sharing data among processes
- Lends itself to protection

94

94

Segment Tables

- Corresponding segment in main memory
- Each entry contains the length of the segment
- A bit is needed to determine if segment is already in main memory – present bit (P)
- Another bit is needed to determine if the segment has been modified since it was loaded in main memory – modify bit (M)



(b) Segmentation only

95

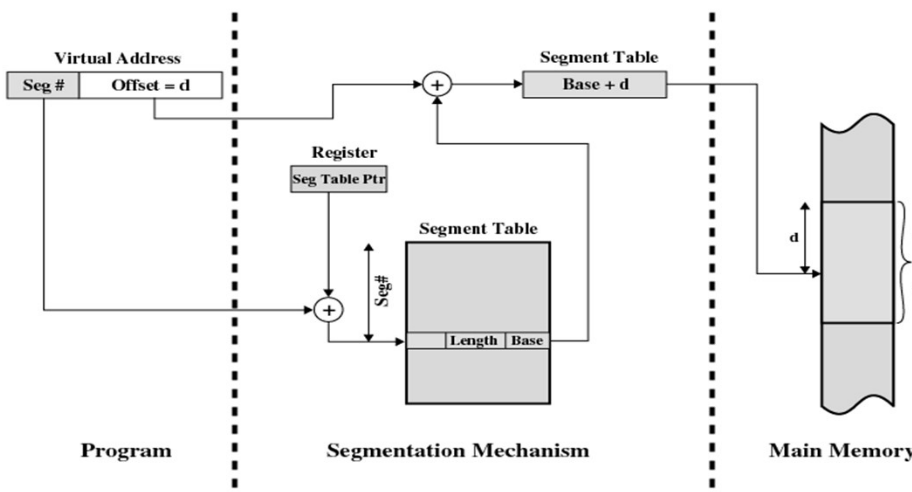


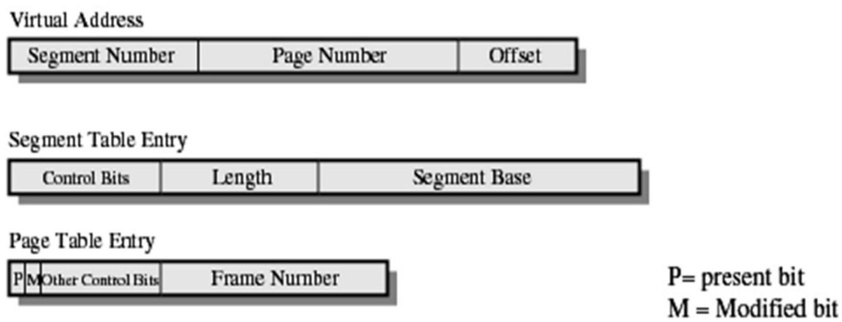
Figure 8.12 Address Translation in a Segmentation System

96

96

Combined Paging and Segmentation

- Paging is transparent to the programmer
- Segmentation is visible to the programmer
- Each segment is broken into fixed-size



(c) Combined segmentation and paging

97

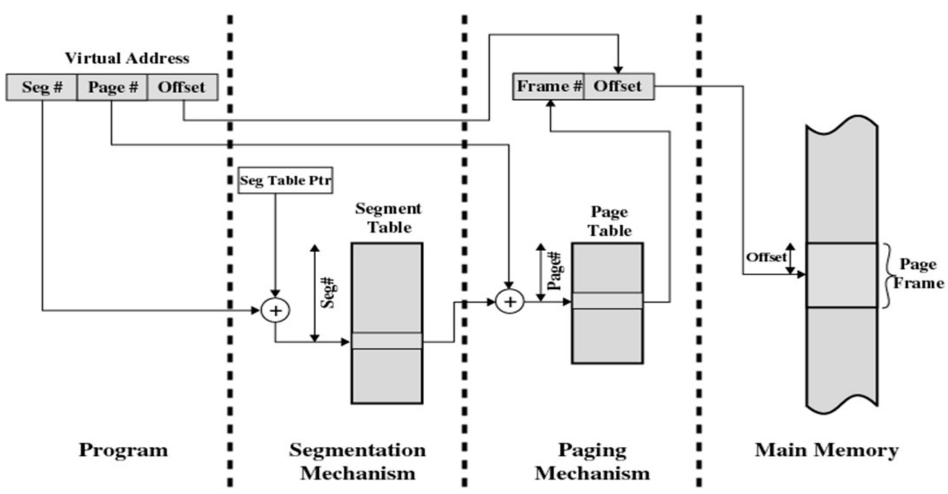


Figure 8.13 Address Translation in a Segmentation/Paging System

98

98

Present Bit

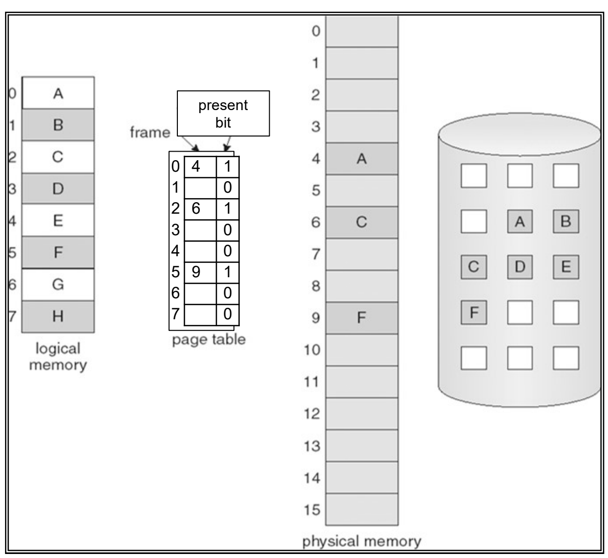
- Each page table entry is associated with a valid–invalid bit (a.k.a present bit, P) (1 ⇒ in-memory, 0 ⇒ not-in-memory)
- Initially present bit is set to 0 on all entries
- Example of a page table snapshot:
- During address translation, if present bit in page table entry is 0 ⇒ page fault

Frame #	Present bit
	1
	1
	1
	1
	0
⋮	
	0
	0

page table

99

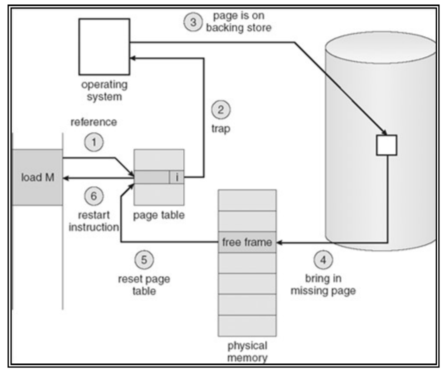
Page Table When Some Pages Are Not in Main Memory



100

Page Fault

- If there is ever a reference to a page,
  first reference will trap to OS --> page fault
- OS looks at another table to decide:
  - Invalid reference --> abort.
  - Just not in memory.
- Get empty frame.
- Swap page into frame.
- Reset tables, validation bit = 1.
- Restart instruction: Least Recently Used
  - block move
  - auto increment/decrement location



Steps in Handling a Page Fault

101

101

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

102

102

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

103

103

Replacement Policy

- Thrashing
  - Swapping out a piece of a process just before that piece is needed
  - The processor spends most of its time swapping pieces rather than executing user instructions
- Placement Policy
  - Which page is replaced?
  - Page removed should be the page least likely to be referenced in the near future
  - Most policies predict the future behavior on the basis of past behavior
- Frame Locking
  - If frame is locked, it may not be replaced
    - Kernel of the operating system
    - Control structures
    - I/O buffers
  - Associate a lock bit with each frame

104

104

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
- In all our examples, the reference string is
  1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5

105

105

Basic Replacement Algorithms

- First-in, first-out (FIFO)
- Least Recently Used (LRU)
- Optimal

- Second chance / Clock
- Counting

106

106

Basic Replacement Algorithms

- First-in, first-out (FIFO)
  - Treats page frames allocated to a process as a circular buffer
  - Pages are removed in round-robin style
  - Simplest replacement policy to implement
  - Page that has been in memory the longest is replaced
  - These pages may be needed again very soon

107

107

Basic Replacement Algorithms

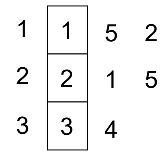
- Least Recently Used (LRU)
  - Replaces the page that has not been referenced for the longest time
  - By the principle of locality, this should be the page least likely to be referenced in the near future
  - Each page could be tagged with the time of last reference. This would require a great deal of overhead.
- Optimal policy
  - Selects for replacement that page for which the time to the next reference is the longest
  - Impossible to have perfect knowledge of future events

108

108

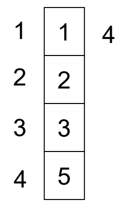
First-In-First-Out (FIFO)

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



8 page faults

- 4 frames



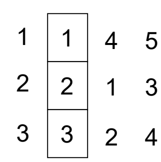
5 page faults

109

109

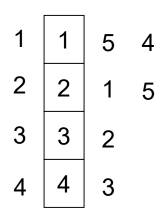
First-In-First-Out (FIFO)

- 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)



9 page faults

- 4 frames



10 page faults

- FIFO Replacement – Belady's Anomaly
  - more frames ⇒ more page faults

110

110

FIFO

- Given a reference string:

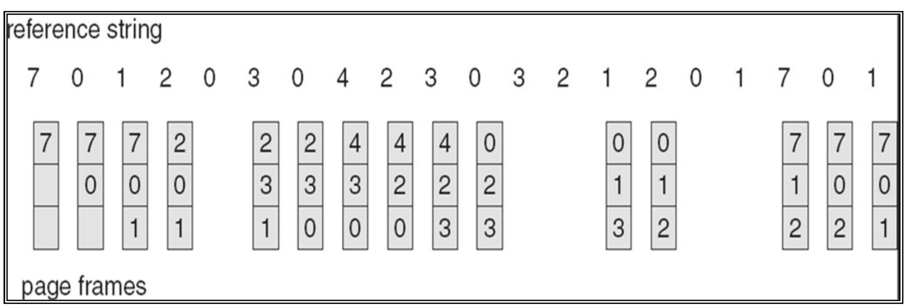
7 0 1 2 0 3 0 4 2 3 0 3 2 1 2 0 1 7 0 1

- Identify how many page faults would occur if there are:
  - 3 frames
  - 4 frames

111

111

FIFO Page Replacement



112

112

Least Recently Used (LRU)

- Reference string: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5

1	5	
2		
3	5	4
4	3	

- Counter implementation
  - Every page entry has a counter; every time page is referenced through this entry, copy the clock into the counter
  - When a page needs to be changed, look at the counters to determine which are to change

113

LRU

- Reference string: 1, 2, 3, 1, 2, 5, 3, 1, 4, 2, 5

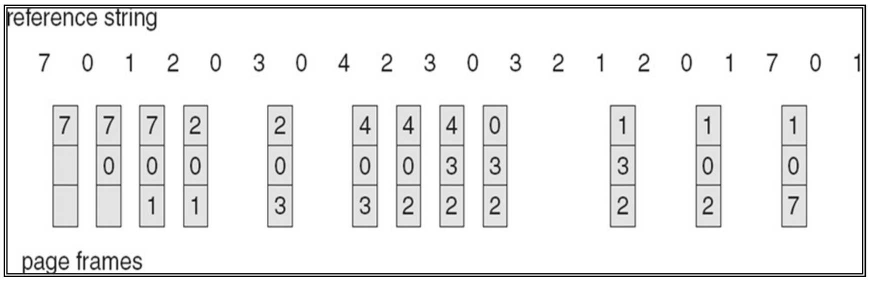
- Given a reference string:

7 0 1 2 0 3 0 4 2 3 0 3 2 1 2 0 1 7 0 1

- Identify how many page faults would occur if there are:
  - 3 frames
  - 4 frames

114

LRU Page Replacement

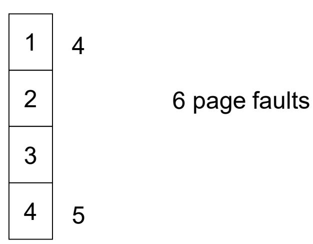


115

115

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



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

116

116

Optimal

- Reference string: 1, 2, 3, 1, 2, 5, 3, 1, 4, 2, 5
- Given a reference string:

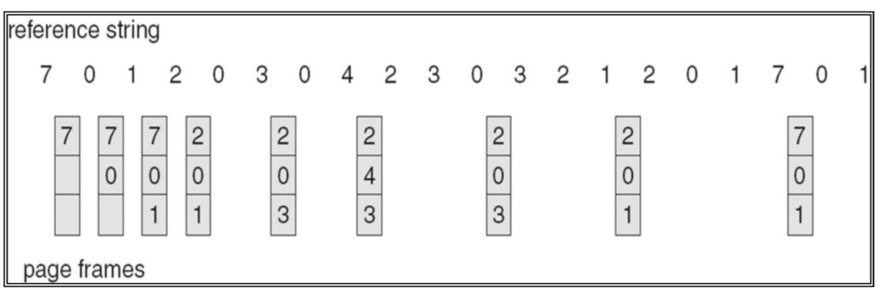
7 0 1 2 0 3 0 4 2 3 0 3 2 1 2 0 1 7 0 1

- Identify how many page faults would occur if there are:
  - 3 frames
  - 4 frames

117

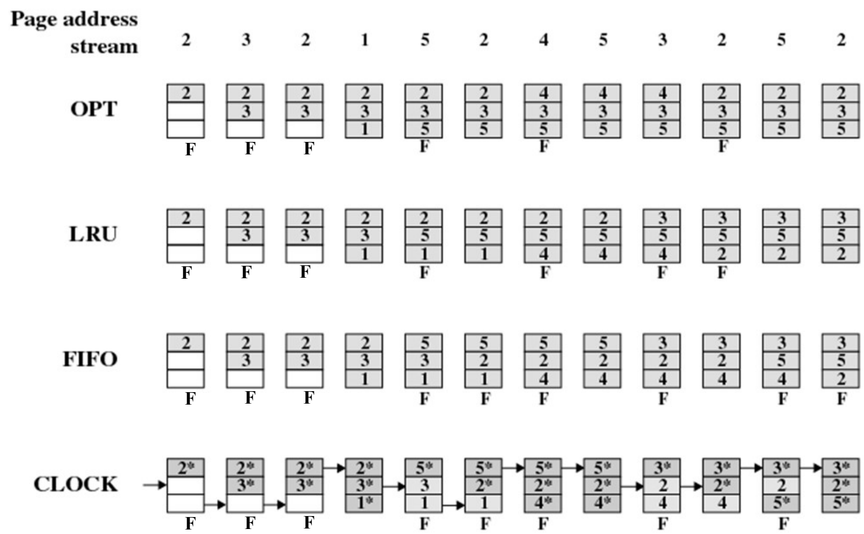
117

Optimal Page Replacement



118

118



F = page fault occurring after the frame allocation is initially filled

Figure 8.15 Behavior of Four Page-Replacement Algorithms

119

Example

- Given a reference string:
  4 7 0 7 1 0 3 4 1 2 1 2 7 3 5
- For each of these page replacement algorithms:
  - FIFO
  - Optimal
  - LRU
- Identify how many page faults would occur if there are:
  - 4 frames
  - 3 frames

120

120

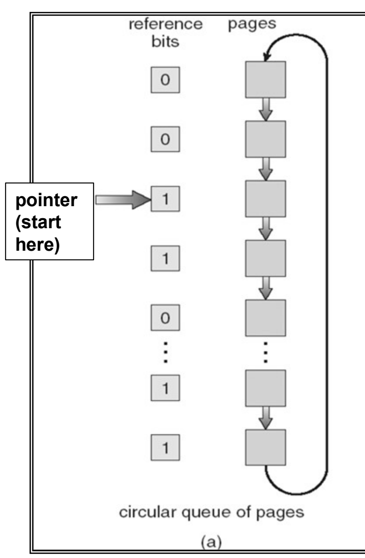
LRU Approximation Algorithms

- Reference bit
  - With each page associate a bit, initially = 0
  - When page is referenced bit set to 1
  - Replace the one which is 0 (if one exists). We do not know the order, however.
- Second chance / Clock replacement
  - Need reference bit
  - If page to be replaced (in clock order) has reference bit = 1 then:
    - set reference bit 0
    - leave page in memory
    - replace next page (in clock order), subject to same rules

121

121

Second-Chance (Clock) Page-Replacement Algorithm

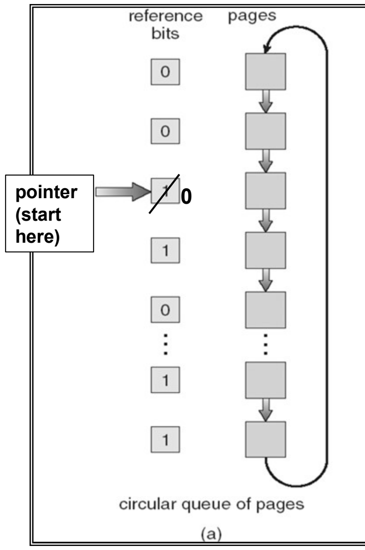


If bit = 1, then change to 0
If bit = 0, then replace page

122

122

Second-Chance (Clock) Page-Replacement Algorithm

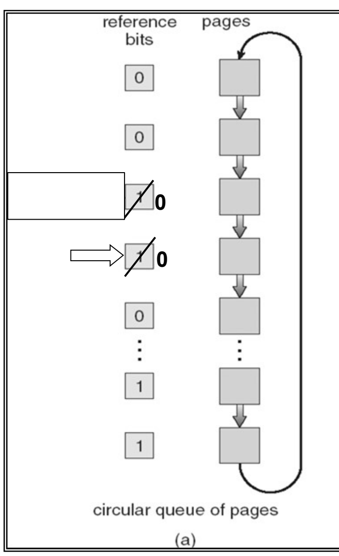


If bit = 1, then change to 0
If bit = 0, then replace page

123

123

Second-Chance (Clock) Page-Replacement Algorithm

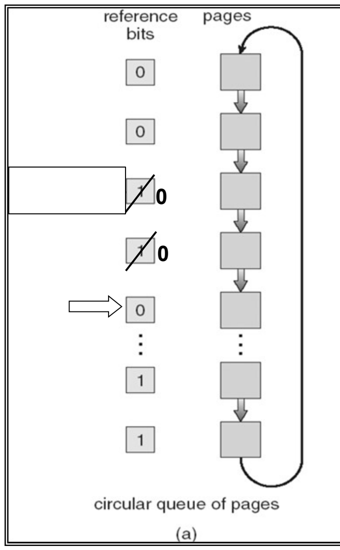


If bit = 1, then change to 0
If bit = 0, then replace page

124

124

Second-Chance (Clock) Page-Replacement Algorithm



If bit = 1, then change to 0
If bit = 0, then replace page

125

Counting Algorithms

- Keep a counter of the number of references that have been made to each page
- **LFU Algorithm**: replaces page with smallest count
- **MFU Algorithm**: based on the argument that the page with the smallest count was probably just brought in and has yet to be used

126

Comparison of Placement Algorithms

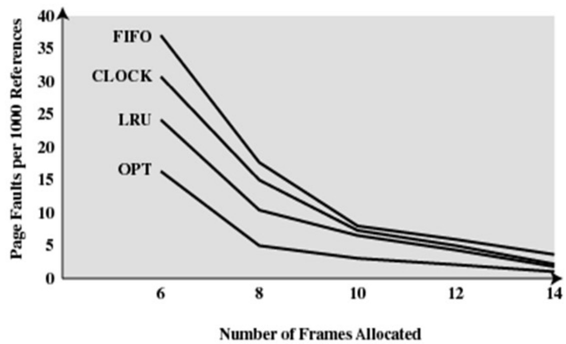


Figure 8.17 Comparison of Fixed-Allocation, Local Page Replacement Algorithms

127

127

Global vs. Local Allocation

- **Global replacement** – process selects a replacement frame from the set of all frames; one process can take a frame from another
- **Local replacement** – each process selects from only its own set of allocated frames

128

128

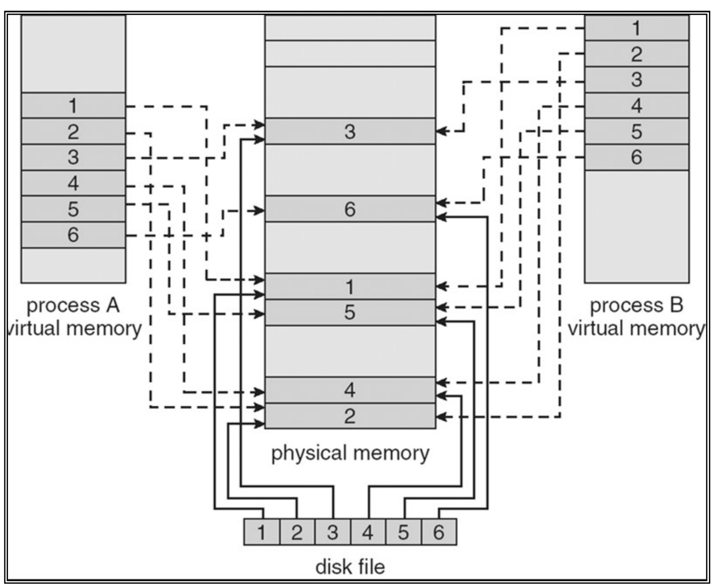
Memory-Mapped Files

- Memory-mapped file I/O allows file I/O to be treated as routine memory access by **mapping** a disk block to a page in memory
- A file is initially read using demand paging. A page-sized portion of the file is read from the file system into a physical page. Subsequent reads/writes to/from the file are treated as ordinary memory accesses.
- Simplifies file access by treating file I/O through memory rather than **read() write()** system calls
- Also allows several processes to map the same file allowing the pages in memory to be shared

129

129

Memory Mapped Files



130

130

Memory-Mapped Files in Java

```
import java.io.*;
import java.nio.*;
import java.nio.channels.*;
public class MemoryMapReadOnly
{
    // Assume the page size is 4 KB
    public static final int PAGE SIZE = 4096;
    public static void main(String args[]) throws IOException {
        RandomAccessFile inFile = new
    RandomAccessFile(args[0],"r");
        FileChannel in = inFile.getChannel();
        MappedByteBuffer mappedBuffer =
          in.map(FileChannel.MapMode.READ ONLY, 0, in.size());
        long numPages = in.size() / (long)PAGE SIZE;
        if (in.size() % PAGE SIZE > 0)
                ++numPages;
```

131

131

Memory-Mapped Files in Java (cont)

```
        // we will "touch" the first byte of every page
        int position = 0;
        for (long i = 0; i < numPages; i++) {
                byte item = mappedBuffer.get(position);
                position += PAGE SIZE;
        }
        in.close();
        inFile.close();
    }
}
```

- The API for the map() method is as follows:

map(mode, position, size)

132

132

Other Issues -- Prepaging

- Prepaging
  - To reduce the large number of page faults that occurs at process startup
  - Prepage all or some of the pages a process will need, before they are referenced
  - But if prepaged pages are unused, I/O and memory was wasted
  - Assume s pages are prepaged and α of the pages is used
    - Is cost of $s * \alpha$ save pages faults > or < than the cost of prepaging
      $s * (1- \alpha)$ unnecessary pages?
    - α near zero $\Rightarrow$ prepaging loses

133

133

Other Issues – Page Size

- Page size selection must take into consideration:
  - fragmentation
  - table size
  - I/O overhead
  - locality

134

134

Other Issues – Program Structure

- Program structure
  - Int[128,128] data;
  - Each row is stored in one page
  - Program 1

```
for (j = 0; j <128; j++)
    for (i = 0; i < 128; i++)
        data[i,j] = 0;
```

128 x 128 = 16,384 page faults

  - Program 2

```
for (i = 0; i < 128; i++)
    for (j = 0; j < 128; j++)
        data[i,j] = 0;
```

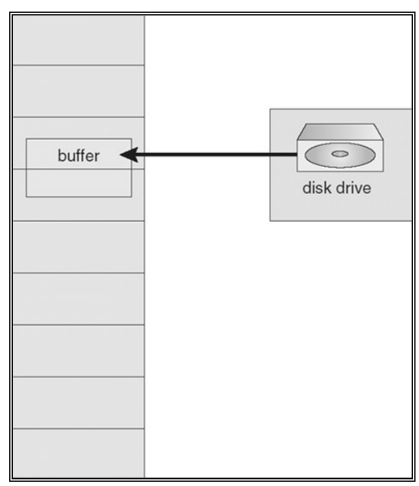
128 page faults

135

135

Other Issues – I/O interlock

- **I/O Interlock** – Pages must sometimes be locked into memory
- Consider I/O. Pages that are used for copying a file from a device must be locked from being selected for eviction by a page replacement algorithm.



Reason Why Frames Used for I/O Must Be in Memory

136

136

Operating System Examples

- Windows XP
- Solaris

137

137

Windows XP

- Uses demand paging with **clustering**. Clustering brings in pages surrounding the faulting page.
- Processes are assigned **working set minimum** and **working set maximum**
- Working set minimum is the minimum number of pages the process is guaranteed to have in memory
- A process may be assigned as many pages up to its working set maximum
- When the amount of free memory in the system falls below a threshold, **automatic working set trimming** is performed to restore the amount of free memory
- Working set trimming removes pages from processes that have pages in excess of their working set minimum

138

138

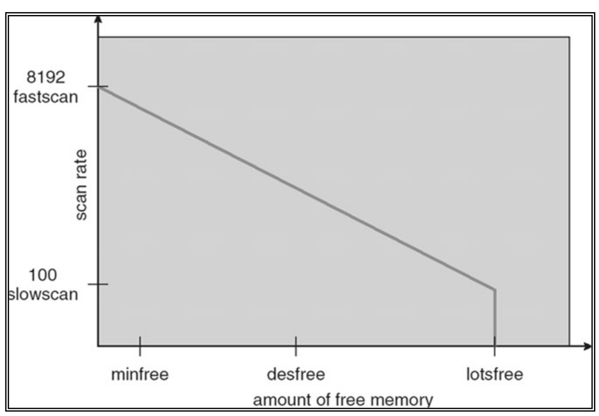
Solaris

- Maintains a list of free pages to assign faulting processes
- *Lotsfree* – threshold parameter (amount of free memory) to begin paging
- *Desfree* – threshold parameter to increasing paging
- *Minfree* – threshold parameter to being swapping
- Paging is performed by *pageout* process
- Pageout scans pages using modified clock algorithm
- *Scanrate* is the rate at which pages are scanned. This ranges from *slowscan* to *fastscan*
- Pageout is called more frequently depending upon the amount of free memory available

139

139

Solaris 2 Page Scanner



140

140