Understanding Operating Systems Fifth Edition

Chapter 3
Memory Management: Virtual Memory

Learning Objectives

After completing this chapter, you should be able to describe:

- The basic functionality of the memory allocation methods covered in this chapter: paged, demand paging, segmented, and segmented/demand paged memory allocation
- The influence that these page allocation methods have had on virtual memory

Learning Objectives (cont'd.)

- The difference between a first-in first-out page replacement policy, a least-recently-used page replacement policy, and a clock page replacement policy
- The mechanics of paging and how a memory allocation scheme determines which pages should be swapped out of memory

Learning Objectives (cont'd.)

- The concept of the working set and how it is used in memory allocation schemes
- The impact that virtual memory had on multiprogramming
- Cache memory and its role in improving system response time

Introduction

- Evolution of virtual memory
  - Paged, demand paging, segmented, segmented/demand paging
  - Foundation for current virtual memory methods
- Improvement areas
  - Continuous program storage
  - Placement of entire program in memory during execution
  - Fragmentation
  - Overhead due to relocation

Introduction (cont'd.)

- Page replacement policies
  - First-In First-Out
  - Least Recently Used
    - Clock replacement and bit-shifting
  - Mechanics of paging
  - The working set
- Virtual memory
  - Concepts and advantages
- Cache memory
  - Concepts and advantages

Paged Memory Allocation

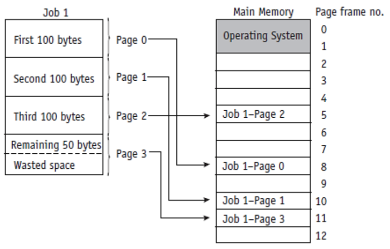
- Divides each incoming job into pages of equal size
- Best condition
  - Page size = Memory block size (page frames) = Size of disk section (sector, block)
    - Sizes depend on operating system and disk sector size
- Memory manager tasks prior to program execution
  - Determines number of pages in program
  - Locates enough empty page frames in main memory
  - Loads all program pages into page frames

Paged Memory Allocation (cont'd.)

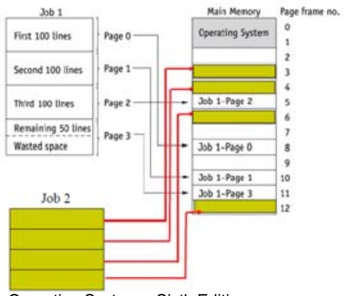
- Program: stored in noncontiguous page frames
  - Advantages: more efficient memory use; compaction scheme eliminated (no external fragmentation)
  - New problem: keeping track of job's pages (increased operating system overhead)

Paged Memory Allocation (cont'd.)

(figure 3.1)
In this example, each page frame can hold 100 bytes. This job, at 350 bytes long, is divided among four page frames with internal fragmentation in the last page frame.
© Cengage Learning 2014



If another job arrives



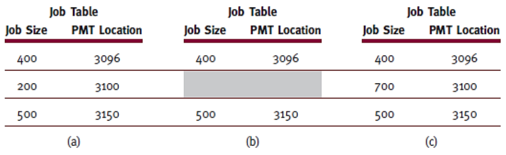
Paged Memory Allocation (cont'd.)

- Internal fragmentation: job's last page frame only
- Entire program: required in memory during its execution
- Three tables for tracking pages: Job Table (JT), Page Map Table (PMT), and Memory Map Table (MMT)
  - Stored in main memory: operating system area

Paged Memory Allocation (cont'd.)

- Three tables for tracking pages
  - **Job Table (JT)**
    - Size of job
    - Memory location where its PMT is stored
  - **Page Map Table (PMT)**
    - Page number
    - Corresponding page frame memory address
  - **Memory Map Table (MMT)**
    - Location for each page frame
    - Free/busy status

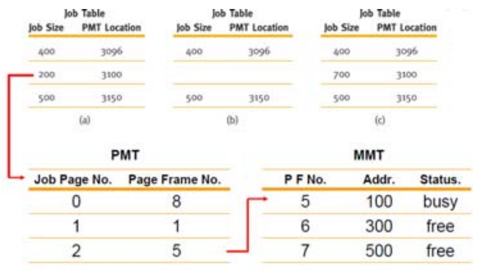
Paged Memory Allocation (cont'd.)



Job Table		Job Table		Job Table	
Job Size	PMT Location	Job Size	PMT Location	Job Size	PMT Location
400	3096	400	3096	400	3096
200	3100			700	3100
500	3150	500	3150	500	3150
(a)		(b)		(c)	

(table 3.1)
This section of the Job Table initially has one entry for each job (a). When the second job ends (b), its entry in the table is released and then replaced by the entry for the next job (c).
© Cengage Learning 2014

Job Table, PMT and MMT



Job Table		Job Table		Job Table	
Job Size	PMT Location	Job Size	PMT Location	Job Size	PMT Location
400	3096	400	3096	400	3096
200	3100			700	3100
500	3150	500	3150	500	3150
(a)		(b)		(c)	

PMT

Job Page No.	Page Frame No.
0	8
1	1
2	5

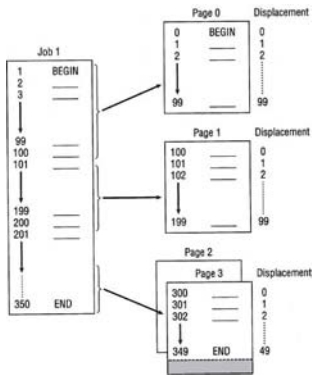
MMT

P F No.	Addr.	Status.
5	100	busy
6	300	free
7	500	free

Paged Memory Allocation (cont'd.)

- **Displacement** (offset) of a line
  - Line distance from beginning of its page
  - Locates line within its page frame
  - Relative value
- Determining page number and displacement of a line
  - Divide job space address by the page size
  - **Page number**: integer quotient from the division
  - **Displacement**: remainder from the division

(figure 3.2)
This job is 350 bytes long and is divided into four pages of 100 bytes each that are loaded into four page frames in memory.
© Cengage Learning 2014



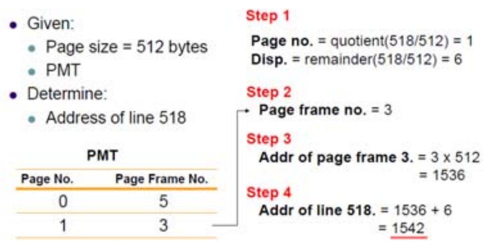
Determining Displacement and Page Number

- **Example:**
  - Page size = 100 (lines)
  - Address of line 214:
    - Page number = quotient(214/100) = 2
    - Displacement = remainder(214/100) = 14

Paged Memory Allocation (cont'd.)

- Instruction: determining exact location in memory
  - Step1: Determine page number/displacement of line
  - Step 2: Refer to the job's PMT
    - Determine page frame containing required page
  - Step 3: Obtain beginning address of page frame
    - Multiply page frame number by page frame size
  - Step 4: Add the displacement (calculated in first step) to starting address of the page frame
- Address resolution (address translation)
  - Job space address (logical) → physical address (absolute)

Address Resolution: Example



- Given:
  - Page size = 512 bytes
  - PMT
- Determine:
  - Address of line 518

PMT

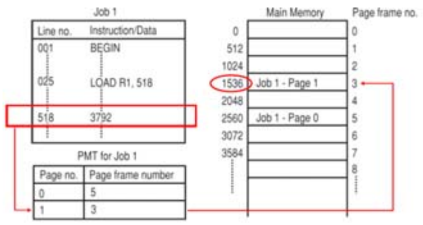
Page No.	Page Frame No.
0	5
1	3

Step 1
Page no. = quotient(518/512) = 1
Disp. = remainder(518/512) = 6

Step 2
Page frame no. = 3

Step 3
Addr of page frame 3. = 3 x 512 = 1536

Step 4
Addr of line 518. = 1536 + 6 = 1542



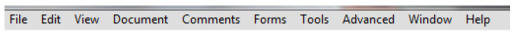
Paged Memory Allocation (cont'd.)

- Advantages
  - Allows job allocation in noncontiguous memory
    - Efficient memory use
- Disadvantages
  - Increased overhead from address resolution
  - Internal fragmentation in last page
  - Must store entire job in memory location
- Page size selection is crucial
  - Too small: generates very long PMTs
  - Too large: excessive internal fragmentation

Demand Paging Memory Allocation

- Pages brought into memory only as needed
  - Removes restriction: entire program in memory
  - Requires high-speed page access
- Exploits programming techniques
  - Modules written sequentially
    - All pages not necessary needed simultaneously
  - Examples
    - User-written error handling modules
    - Mutually exclusive modules
    - Certain program options: mutually exclusive or not accessible

- When using one module the others are usually not required.

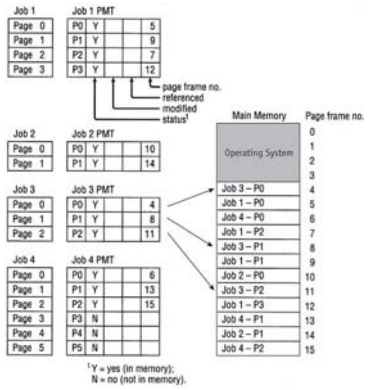


Demand Paging (cont'd.)

- Allowed for wide availability of **virtual memory** concept
  - Provides appearance of almost infinite or nonfinite physical memory
  - Jobs run with less main memory than required in paged memory allocation scheme
  - Requires high-speed direct access storage device
    - Works directly with CPU
  - **Swapping**: how and when pages passed between memory and secondary storage
    - Depends on predefined policies

Demand Paging (cont'd.)

- Memory Manager requires three tables
- **Job Table**
- **Page Map Table**:
  - First field: page requested already in memory?
  - Second field: page contents modified?
  - Third field: page referenced recently?
  - Fourth field: frame number
- **Memory Map Table**



(figure 3.5)
Demand paging requires that the Page Map Table for each job keep track of each page as it is loaded or removed from main memory. Each PMT tracks the status of the page, whether it has been modified, whether it has been recently referenced, and the page frame number for each page currently in main memory. (Note: For this illustration, the Page Map Tables have been simplified. See Table 3.3 for more detail.
© Cengage Learning 2014

Demand Paging (cont'd.)

- **Swapping Process**
  - Exchanges resident memory page with secondary storage page
  - Involves
    - Copying resident page to disk (if it was modified)
    - Writing new page into the empty page frame
  - Requires close interaction between:
    - Hardware components
    - Software algorithms
    - Policy schemes

Demand Paging Memory Allocation (cont'd.)

- Hardware components:
  - Generate the address: required page
  - Find the page number
  - Determine page status: already in memory
- Page fault: failure to find page in memory
- Page fault handler: part of operating system
  - Determines if empty page frames in memory
    - Yes: requested page copied from secondary storage
    - No: swapping (dependent on the predefined policy)

Hardware Instruction Processing

```
1. Start processing instructions
2. Generate data address
3. Compute page number
4. If page is in memory, then
       get data and finish instruction
       advance to next instruction
       return to step 1
   Else
       generate page interrupt
       call page fault handler
   End If
```

The cost of a page fault

- Let
  - $\boldsymbol{T_m}$ be the main memory access time
  - $\boldsymbol{T_d}$ the disk access time
  - f the page fault rate
  - $\boldsymbol{T_a}$ the average access time of the VM

$$T_a = (1 - f)\ \boldsymbol{T_m} + f\,(\boldsymbol{T_m} + \boldsymbol{T_d}) = \boldsymbol{T_m} + f\,\boldsymbol{T_d}$$

Example

- Assume T_m = 70 ns and T_d = 7 ms

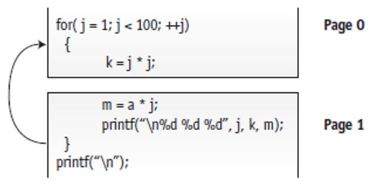
f	T_a
10^{-3}	= 70ns + 7ms/ 10^3 = 7,070 ns
10^{-4}	= 70ns + 7ms/ 10^4 = 770 ns
10^{-5}	= 70ns + 7ms/ 10^5 = 140 ns
10^{-6}	= 70ns + 7ms/ 10^6 = 77ns

Conclusion

- Demand paging (virtual memory) works best when page fault rate is less than a page fault per 100,000 instructions

Demand Paging Memory Allocation (cont'd.)

- Tables updated when page swap occurs
  - PMT for both jobs (page swapped out; page swapped in) and the MMT
- Thrashing
  - Excessive page swapping: inefficient operation
  - Main memory pages: removed frequently; called back soon thereafter
  - Occurs across jobs
    - Large number of jobs: limited free pages
  - Occurs within a job
    - Loops crossing page boundaries


```
for( j = 1; j < 100; ++j)
 {
        k = j * j;
        m = a * j;
        printf("\n%d %d %d", j, k, m);
 }
printf("\n");
```

(figure 3.6) Asume only one empty page available
An example of demand paging that causes a page swap each time the loop is executed and results in thrashing. If only a single page frame is available, this program will have one page fault each time the loop is executed.
© Cengage Learning 2014

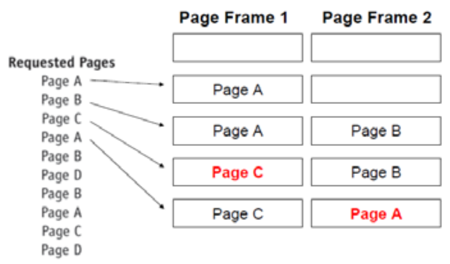
Demand Paging (cont'd.)

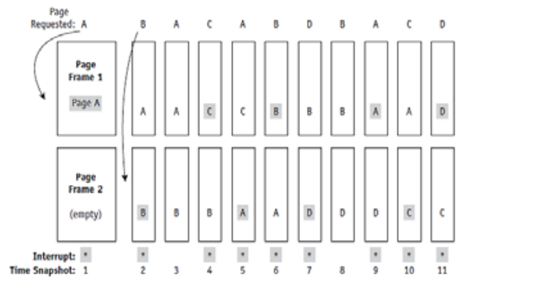
- Advantages
  - Job no longer constrained by the size of physical memory (concept of virtual memory)
  - Utilizes memory more efficiently than previous schemes
  - Faster response
- Disadvantages
  - Increased overhead caused by tables and page interrupts

Page Replacement Policies and Concepts

- Policy to select page removal
  - Crucial to system efficiency
- Page replacement polices
  - **First-In First-Out (FIFO)** policy
    - Best page to remove is one in memory longest
  - **Least Recently Used (LRU)** policy
    - Best page to remove is least recently accessed
- Mechanics of paging concepts
- The working set concept

FIFO



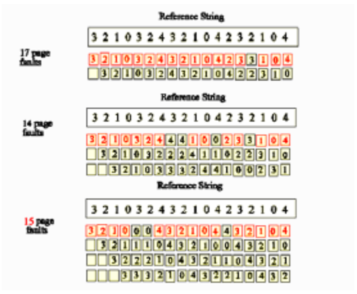

Page Requested: A B A C A B D B A C D
Page Frame 1
Page A
Page Frame 2
(empty)
Interrupt:
Time Snapshot: 1 2 3 4 5 6 7 8 9 10 11

(figure 3.8)
Using a FIFO policy, this page trace analysis shows how each page requested is swapped into the two available page frames. When the program is ready to be processed, all four pages are in secondary storage. When the program calls a page that isn't already in memory, a page interrupt is issued, as shown by the gray boxes and asterisks. This program resulted in nine page interrupts.
© Cengage Learning 2014

First-In First-Out

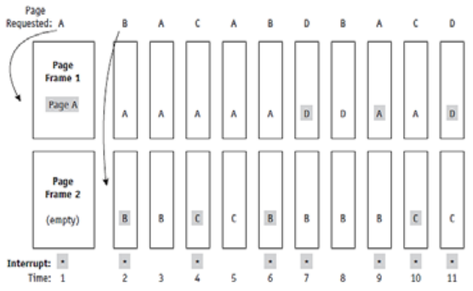
- Removes page in memory the longest
- **Efficiency**
  - Failure rate: Ratio of page interrupts to page requests
  - FIFO example: not so good
    - Failure rate is 9/11 or 82%
    - Success Rate 2/11 or 18%
- **FIFO anomaly**
  - More memory does not lead to better performance

FIFO or Belady's Anomaly



Least Recently Used

- Removes page least recently accessed
- Efficiency
  - Causes either decrease in or same number of interrupts
  - Slightly better (compared to FIFO): 8/11 or 73%
  - Increasing main memory will cause either a decrease in or the same number of page interrupts
  - Does not experience FIFO anomaly


Page Requested: A B A C A B D B A C D
Page Frame 1
Page A
Page Frame 2
(empty)
Interrupt:
Time: 1 2 3 4 5 6 7 8 9 10 11

(figure 3.9)
Memory management using an LRU page removal policy for the program shown in Figure 3.8. Throughout the program, 11 page requests are issued, but they cause only 8 page interrupts.
© Cengage Learning 2014

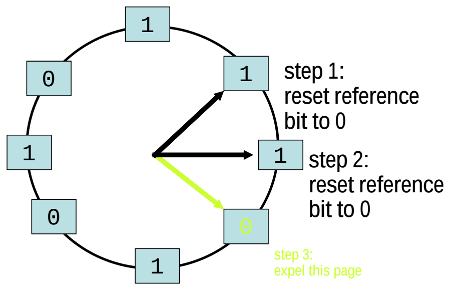
Least Recently Used (cont'd.)

- Two variations
  - Clock replacement technique
    - Circular queue: pointer steps through active pages'
    - Paced according to the computer's clock cycle
  - Bit-shifting technique
    - Uses 8-bit reference byte and bit-shifting technique
    - Tracks usage of each page currently in memory

Clock Page Replacement

- Implementation
  - A circular queue and a pointer
  - Reference bits
- Replacement
  - If its reference bit is zero, page is targeted for removal
  - If its reference bit is one, set to zero and move to the next page.

Clock Replacement (Multics)


1
0
1
1
1
0
1
0
step 1: reset reference bit to 0
step 2: reset reference bit to 0
step 3: expel this page

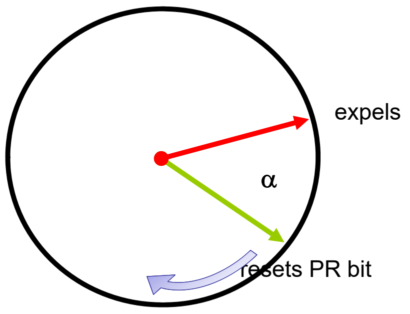
Variations of Clock Replacement

- When memory is overused, hand of clock moves too fast to find pages to be expelled
  - Too many resets
  - Too many context switches
- Berkeley UNIX limited CPU overhead of policy to 10% of CPU time
  - No more than 300 page scans/second

Evolution of the policy

- In mid 1980s memory sizes started to grow. Hand clock was taking too much time to scan the whole memory.
- By the late 80's a ***two-hand policy*** was introduced:
  - First hand resets simulated PR bit
  - Second hand follows first at constant angle and expels all pages whose PR bit = 0

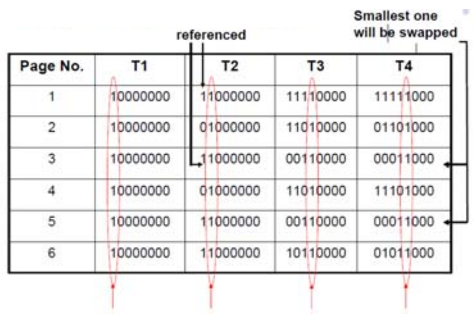
Two Hand Policy



Bit-Shifting Technique

- When page is first copied to memory, leftmost bit of its reference byte is zero. And all others set to zero
- At each specific time interval all bits are shifted to right
- Each time the page is referenced leftmost bit is set to 1
- When page fault occurs the one with the smallest value is expelled.

Bit-Shifting

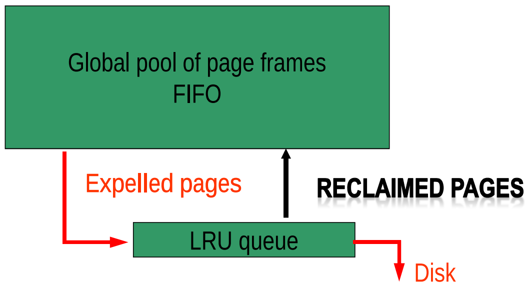

referenced
Smallest one will be swapped

Page No.	T1	T2	T3	T4
1	10000000	11000000	11110000	11111000
2	10000000	01000000	11010000	01101000
3	10000000	11000000	00110000	00011000
4	10000000	01000000	11010000	11101000
5	10000000	11000000	00110000	00011000
6	10000000	11000000	10110000	01011000

Mach Policy (I, 1980s)

- Mach divides its main memory into a *pool of page frames* shared by all processes and one ***global queue*** from which pages can be reclaimed.
- Pool of page frames are managed according to Global ***FIFO policy.***
- Expelled pages go to the end of a ***global LRU queue*** where they wait before being actually expelled from main memory
  - Can be ***rescued*** if they were ***expelled by error***
  - FIFO policy makes many errors

Mach Policy (II)



The Mechanics of Paging for LRU

- Page swapping
  - Memory manager requires specific information: Page Map Table

Page No.	Status Bit	Modified Bit	Referenced Bit	Page Frame No.
0	1	1	1	5
1	1	0	0	9
2	1	0	0	7
3	1	0	1	12

(table 3.3)
Page Map Table for Job 1 shown in Figure 3.5.
A 1 = Yes and 0 = No.
© Cengage Learning 2014

The Mechanics of Paging (cont'd.)

- Page map table bit meaning
  - Status bit
    - Indicates if page currently in memory
  - Referenced bit
    - Indicates if page referenced recently
    - Used by LRU to determine page to swap
  - Modified bit
    - Indicates if page contents altered
    - Used to determine if page must be rewritten to secondary storage when swapped out
- Four combinations of modified and referenced bits

Status Bit		Modified Bit		Referenced Bit	
Value	Meaning	Value	Meaning	Value	Meaning
0	*not* in memory	0	*not* modified	0	*not* called
1	*resides* in memory	1	*was* modified	1	*was* called

(table 3.4)
The meaning of these bits used in the Page Map Table.
© Cengage Learning 2014

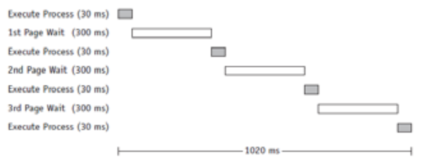
	Modified?	Referenced?	What it Means
Case 1	0	0	Not modified AND not referenced
Case 2	0	1	Not modified BUT was referenced
Case 3	1	0	Was modified BUT not referenced [Impossible?]
Case 4	1	1	Was modified AND was referenced

(table 3.5)
Four possible combinations of modified and referenced bits and the meaning of each.
© Cengage Learning 2014

The Working Set

- Set of pages residing in memory accessed directly without incurring a page fault
  - Improves performance of demand page scheme
- Requires concept of "locality of reference"
  - Occurs in well-structured programs
    - Only small fraction of pages needed during program execution
  - Working set changes as job moves
    - Initialize job
    - Repeated calculations
    - Output devices
    - Finalize and close the job

The Working Set (cont'd.)



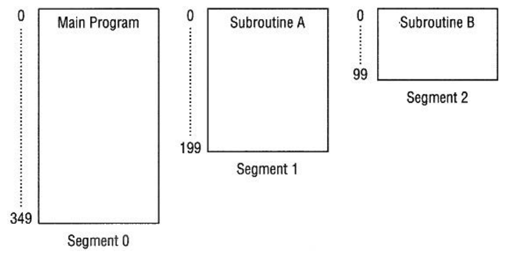
(figure 3.12)
Time line showing the amount of time required to process page faults for a single program. The program in this example takes 120 milliseconds (ms) to execute but an additional 900 ms to load the necessary pages into memory. Therefore, job turnaround is 1020 ms.

Limitation of Working Set

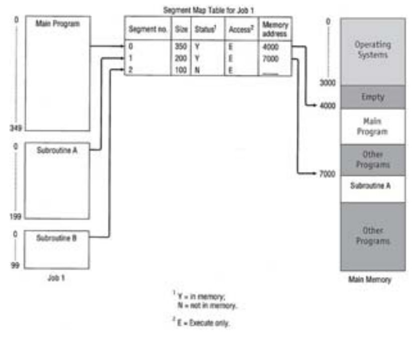
- The very high cost of the hardware required to detect which pages belong to the working set of a process and which pages should be expelled.
- Time sharing systems considerations
  - Must track every working set's size and identity
- System decides
  - Number of pages comprising working set
  - Maximum number of pages allowed for a working set

Segmented Memory Allocation

- Each job divided into several segments
  - Segments are different sizes
  - One for each module containing related functions
- Reduces page faults
  - Segment's loops not split over two or more pages
- Main memory no longer divided into page frames
  - Now allocated dynamically
- Program's structural modules determine segments
  - Each segment numbered when compiled/assembled
  - Segment Map Table (SMT) generated



(figure 3.14)
Segmented memory allocation. Job 1 includes a main program and two subroutines. It is a single job that is structurally divided into three segments of different sizes.
© Cengage Learning 2014



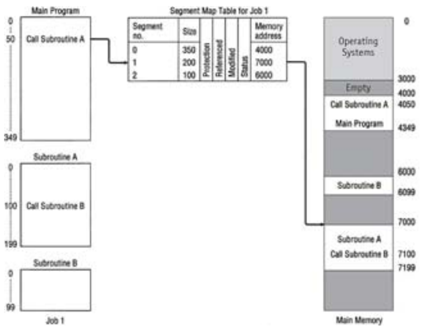
Segment no.	Size	Status[1]	Access[2]	Memory address
0	350	Y	E	4000
1	200	Y	E	7000
2	100	N	E	

[1] Y = in memory; N = not in memory.

[2] E = Execute only.

(figure 3.15)
The Segment Map Table tracks each segment for this job. Notice that Subroutine B has not yet been loaded into memory.
© Cengage Learning 2014

(figure 3.16)
During execution, the main program calls Subroutine A, which triggers the SMT to look up its location in memory.
© Cengage Learning 2014



Segment no.	Size	Protection	Referenced	Modified	Status	Memory address
0	350					4000
1	200					7000
2	100					6000

Segmented Memory Allocation (cont'd.)

- Memory Manager tracks segments using tables
  - **Job Table**
    - Lists every job in process (one for whole system)
  - **Segment Map Table**
    - Lists details about each segment (one for each job)
  - **Memory Map Table**
    - Monitors allocation of main memory (one for whole system)
- Instructions with segments ordered sequentially
- Segments not necessarily stored contiguously

Segmented Memory Allocation (cont'd.)

- Addressing scheme requirement
  - Segment number and displacement
- Advantages
  - Internal fragmentation is removed
  - Memory allocated dynamically
- Disadvantages
  - Difficulty managing variable-length segments in secondary storage
  - External fragmentation

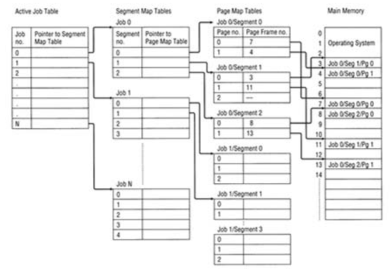
Segmented/Demand Paged Memory Allocation

- Subdivides segments into pages of equal size
  - Smaller than most segments
  - More easily manipulated than whole segments
  - Logical benefits of segmentation
  - Physical benefits of paging
- Segmentation problems removed
  - Compaction, external fragmentation, secondary storage handling
- Addressing scheme requirements
  - Segment number, page number within that segment, and displacement within that page

Segmented/Demand Paged Memory Allocation (cont'd.)

- Scheme requires four tables
  - **Job Table**
    - Lists every job in process (one for the whole system)
  - **Segment Map Table**
    - Lists details about each segment (one for each job)
  - **Page Map Table**
    - Lists details about every page (one for each segment)
  - **Memory Map Table**
    - Monitors allocation of page frames in main memory (one for the whole system)

Segmented/Demand Paged Memory Allocation (cont'd.)



(figure 3.16)

How the Job Table, Segment Map Table, Page Map Table, and main memory interact in a segment/paging scheme.

Virtual Memory

- Allows program execution even if not stored entirely in memory
- Requires cooperation between:
  - Memory Manager: tracks each page or segment
  - Processor hardware: issues the interrupt and resolves the virtual address
- Advantages
  - Job size not restricted to size of main memory
  - Allows an unlimited amount of multiprogramming

Virtual Memory (cont'd.)

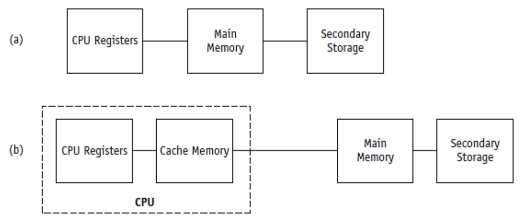
- Advantages (cont'd.)
  - Allows the sharing of code and data
  - Facilitates dynamic linking of program segments
  - Job size is no longer restricted
  - Allowed multiprogramming
  - Memory used more efficiently
- Disadvantages
  - Increased processor hardware costs
  - Increased overhead for handling paging interrupts
  - Increased software complexity to prevent thrashing

Virtual Memory with Paging	Virtual Memory with Segmentation
Allows internal fragmentation within page frames	Doesn't allow internal fragmentation
Doesn't allow external fragmentation	Allows external fragmentation
Programs are divided into equal-sized pages	Programs are divided into unequal-sized segments that contain logical groupings of code
The absolute address is calculated using page number and displacement	The absolute address is calculated using segment number and displacement
Requires Page Map Table (PMT)	Requires Segment Map Table (SMT)

(table 3.6)
Comparison of the advantages and disadvantages of virtual memory with paging and segmentation.
© Cengage Learning 2014

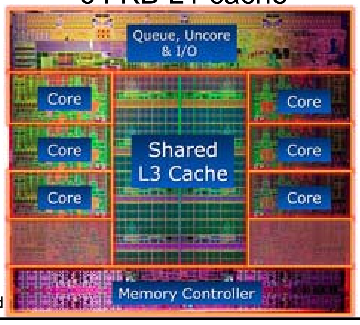
Cache Memory

- Small high-speed intermediate memory unit
- Performance of computer system increased
  - Memory access time significantly reduced
  - Faster processor access compared to main memory
  - Stores frequently used data and instructions
- Two levels of cache (even 3 in some processors)
  - L2: Connected to CPU; contains copy of bus data
  - L1: Pair built into CPU; stores instructions and data
- Data/instructions move from main memory to cache
  - Uses methods similar to paging algorithms



(figure 3.19)
Comparison of (a) the traditional path used by early computers between main memory and the CPU and (b) the path used by modern computers to connect the main memory and the CPU via cache memory.
© Cengage Learning 2014

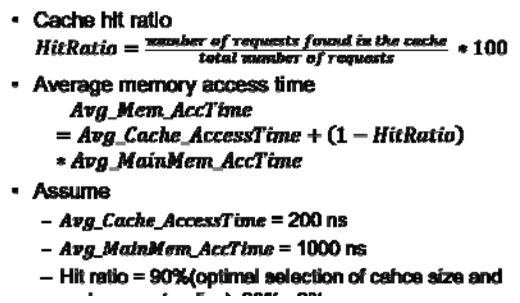
I7 3960X 15 mb L3 cache
256 KB L2 cache
64 KB L1 cache



Cache Memory (cont'd.)

- Four cache memory design factors
  - Cache size, block size, block replacement algorithm (or without algorithm) , and rewrite policy
- An optimal selection of cache and replacement algorithm necessary
  - May lead to 80-90% of all requests in cache
- Efficiency measures
  - **Cache hit ratio (h)**
    - Percentage of total memory request found in cache
  - **Miss ratio (1-h)**
  - **Average memory access time**
    - AvgCacheAccessTime + (1-h) * AvgMemACCTime

Cache Memory (cont'd.)



- Cache hit ratio

$$HitRatio = \frac{number\ of\ requests\ found\ in\ the\ cache}{total\ number\ of\ requests} * 100$$

- Average memory access time

$$Avg\_Mem\_AccTime = Avg\_Cache\_AccessTime + (1 - HitRatio) * Avg\_MainMem\_AccTime$$

- Assume
  - $Avg\_Cache\_AccessTime$ = 200 ns
  - $Avg\_MainMem\_AccTime$ = 1000 ns
  - Hit ratio = 90%(optimal selection of cahce size and

Cache Memory Organization

Fully Associative Mapped Cache

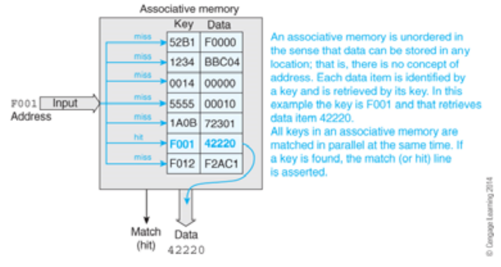
We would like a cache that places no restrictions on what data it can contain; that is, data in the cache can come from anywhere within the main store.

Such a cache uses *associative memory* that can store data anywhere in it because data is accessed by its *value* and not its *address* (location).

Figure below illustrates the concept of an *associative memory*.

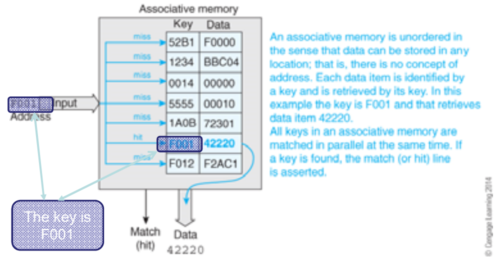
Each entry has two values, a **key** and a data element; for example, the top line contains the **key** 52B1 and the data F0000.

FIGURE 9.7 Principle of the associative memory


Associative memory
Key
Data
miss
52B1
F0000
1234
BBC04
0014
00000
5555
00010
1A0B
72301
hit
F001
42220
F012
F2AC1
F001
Input
Address
Match (hit)
Data
42220
An associative memory is unordered in the sense that data can be stored in any location; that is, there is no concept of address. Each data item is identified by a key and is retrieved by its key. In this example the key is F001 and that retrieves data item 42220.
All keys in an associative memory are matched in parallel at the same time. If a key is found, the match (or hit) line is asserted.
© Cengage Learning 2014

Suppose the computer applies the key F001. This key is applied to all locations in the memory *simultaneously*. Because a match takes place with this key, the memory responds by indicating a match and supplying the value 42220 at its data terminals.

FIGURE 9.7 Principle of the associative memory


Associative memory
Key
Data
miss
52B1
F0000
1234
BBC04
0014
00000
5555
00010
1A0B
72301
hit
F001
42220
F012
F2AC1
F001
Input
Address
The key is F001
Match (hit)
Data
42220
An associative memory is unordered in the sense that data can be stored in any location; that is, there is no concept of address. Each data item is identified by a key and is retrieved by its key. In this example the key is F001 and that retrieves data item 42220.
All keys in an associative memory are matched in parallel at the same time. If a key is found, the match (or hit) line is asserted.
© Cengage Learning 2014

True associative memory requires that we access all elements in parallel to locate the line with the matching key.

This requires parallel access. If we have a million locations, we need to perform a million comparisons in parallel.

Current technology does not permit this (other than for very small associative memories).

Consequently, fully-associative memory cannot be economically constructed.

Furthermore, associative memories require data replacement algorithms because when the cache is full, it is necessary to determine which old entry is ejected when new data is accepted.

Direct Mapped Cache

The easiest way organize a cache memory employs *direct mapping* that relies on a simple algorithm to map data block *i* from the main memory into data block *i* in the cache.

In a direct mapped cache, the lines are arranged into units called *sets,* where the size of a set is the same size as the cache.

For example, a computer with a 16 MB memory and a 64 KB cache would divide the memory into 16 MB/64 KB = 256 sets.

To illustrate how direct-mapped cache works, we'll create a memory with 32 words (bytes) accessed by a 5-bit address that has a cache holding 8 words.

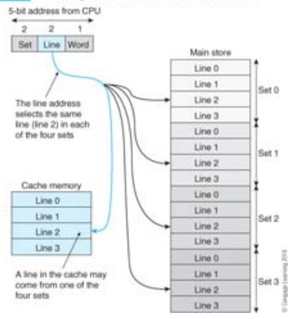
The line size is two words.

The number of sets is memory size/cache size = 32/8 = 4 sets.

A 5-bit address is s_1,s_0,l_1,l_0,w where the *s* bits define the set, the *l* bits define the line and the *w* bit defines the word.

Figure below demonstrates how the word currently addressed by the processor is accessed in memory via its set address, its line address, and its word address.

FIGURE 9.10 Accessing cache and main store with a 5-bit address



When the processor generates an address, the appropriate line in the cache is accessed.

If the address is 01**10**0, line 2 is accessed. There are four lines numbered two—a line 2 in set 0, a line 2 in set 1, a line 2 in set 2, and a line 2 in set 3.

FIGURE 9.10 Accessing cache and main store with a 5-bit address

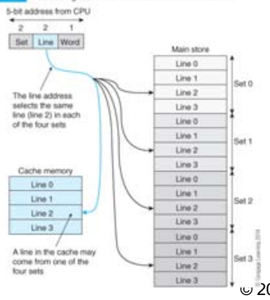
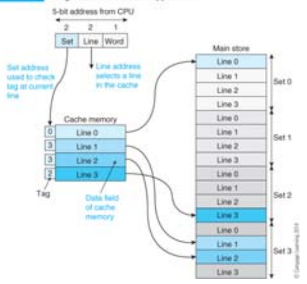

5-bit address from CPU
2 2 1
Set Line Word
Main store
Line 0
Line 1
Line 2
Line 3
Set 0
Set 1
Set 2
Set 3
The line address selects the same line (line 2) in each of the four sets
Cache memory
A line in the cache may come from one of the four sets

Figure 9.11 demonstrates how the ambiguity between lines is resolved by a direct mapped cache.

Each line in the cache memory has a tag or label that identifies which set that particular line belongs to.

FIGURE 9.11 Organization of a direct-mapped cache


5-bit address from CPU
2 2 1
Set Line Word
Set address used to check tag at current line
Line address selects a line in the cache
Cache memory
Tag
Data field of cache memory
Main store
Set 0
Set 1
Set 2
Set 3

When the processor accesses a memory location whose line address is 3, the tag belonging to line 3 in the cache is sent to a comparator. At the same time the set field from the processor is also sent to the comparator. If they are the same, the line in the cache is the requested line and a hit occurs. If they are not the same, a miss occurs and the cache must be updated.

FIGURE 9.11 Organization of a direct-mapped cache

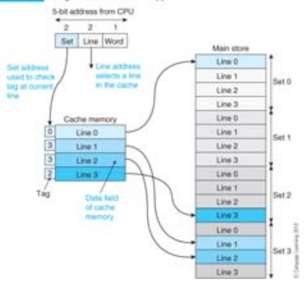
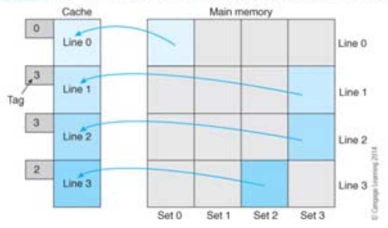

5-bit address from CPU
2 2 1
Set Line Word
Set address used to check tag at current line
Line address selects a line in the cache
Cache memory
Tag
Data field of cache memory
Main store
Set 0
Set 1
Set 2
Set 3

Figure below provides a way of viewing a direct-mapped cache where the main store is depicted as a matrix of dimension *set* x *line*.

FIGURE 9.12 Alternative view of the arrangement of data in a direct-mapped cache



Advantages of Direct Mapped Cache

The direct mapped cache requires no complex line replacement algorithm.

If line *x* in set *y* is accessed and a miss takes place, line *x* from set *y* in the main store is loaded into the frame for line *x* in the cache memory.

No decision concerning which line from the cache is to be rejected has to be made when a new line is to be loaded.

An advantage of direct-mapped cache is its inherent parallelism.

Since the cache memory holding the data and the cache tag RAM are independent, they can both be accessed simultaneously.

Disadvantages of Direct Mapped Cache

The disadvantage of direct-mapped cache is its sensitivity to the location of the data to be cached.

We can relate this to the domestic address book that has, say, one slot for each letter of the alphabet.

If you already have a friend whose surname begins with *S*, you have a problem the next time you meet someone whose name also begins with *S*.

It's annoying because the *Q* and *X* slots are entirely empty.

Because only one line with the number *x* may be in the cache at any instant, accessing data from a different set but with the same line number will always flush the current occupant of line *x* in the cache.

Figure 9.14 illustrates the operation of very simple hypothetical direct-mapped cache in a system with a 16-word main store and an 8-word direct-mapped cache.

Only accesses to instructions are included to simplify the diagram. This cache can hold lines from one of two sets.
We've labeled cache lines 0 to 7 on the left in black.

On the right we've put labels 8 to 15 in blue to demonstrate where lines 8 to 15 from memory locations are cached. The line size is equal to the wordlength and we run the following code.

```
        LDR  r1,(r3)    ;Load r1 from memory location pointed at by r3
        LDR  r2,(r4)    ;Load r2 from memory location pointed at by r4
        BL   Adder      ;Call a subroutine
        B    XYZ        ;

Adder   ADD  r1,r2,r1   ;Add r1 to r2
        MOV  pc,lr      ;Return
```

FIGURE 9.14 Snapshot of a direct-mapped cache while running a program

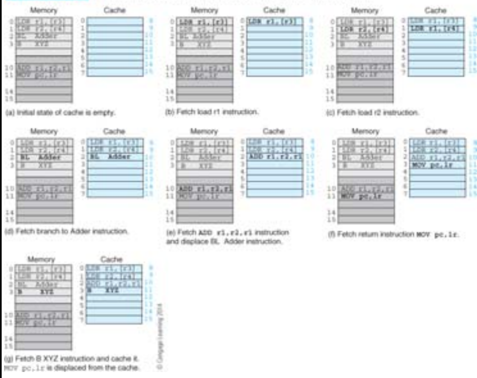

Memory
Cache
(a) Initial state of cache is empty.
(b) Fetch load r1 instruction.
(c) Fetch load r2 instruction.
(d) Fetch branch to Adder instruction.
(e) Fetch ADD r1,r2,r1 instruction and displace BL Adder instruction.
(f) Fetch return instruction MOV pc,lr.
(g) Fetch B XYZ instruction and cache it. MOV pc,lr is displaced from the cache.
© Cengage Learning 2014

Figure 9.14 shows only instruction fetch cycles.

Figure 9.14(a) shows the initial state of the system.

Figures 9.14(b) to (d) show the fetching of the first three instructions, each of which is loaded into a consecutive cache location.

When the subroutine is called in Figure 9.14(d), a branch is made to the instruction at location 10.

In this direct-mapped cache, line 10 is the same as line 2.

Consequently, in Figure 9.14(e) the ADD overwrites the B instruction in line 2 of the cache.

This is called a *conflict miss* because it occurs when data can't be loaded into a cache because its target location is already occupied.

In Figure 9.14(f) the MOV **pc**,lr instruction in line 11 is loaded into line 3 of the cache.

Finally, in Figure 9.14(g) the return is made and the B XYZ instruction in line 3 is loaded in line 3 of the cache, displacing the previous cached value.

Figure 9.14 demonstrates that even in a trivial system, elements in a direct mapped cache can be easily displaced.

If this fragment of code were running in a loop, the repeated displacement of elements in the cache would degrade the performance.

Set-associative Cache

The direct-mapped cache we've just described is easy to implement and doesn't require a line(page)-replacement algorithm.

However, it, doesn't allow two lines with the same number from different sets to be cached at the same time.

The fully associative cache places no restriction on where data can be located, but it requires a means of choosing which line(page) to eject once the cache is full. Moreover, any reasonably large associative cache would be too expensive to construct.

The *set-associative* cache combines the best features of both these types of cache and is not expensive to construct Consequently, it is the form of cache found in all computers.

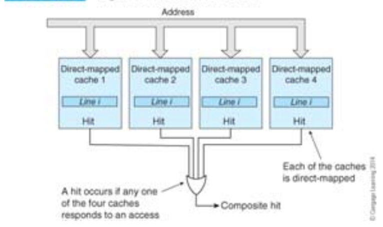
A direct mapped cache has only one location for each line *i*. If you operate two direct mapped caches in *parallel*, line *i* can go in either cache. If you have *n* direct mapped caches operating in parallel, line *i* can go in one of *i* locations. That is an *n*-way set-associative cache.

In an *n*-way set-associative cache there are *n* possible cache locations that a given line can be loaded into.

Typically, *n* is in the range 2 to 8.

Figure 9.15 illustrates the structure of a four-way set-associative cache that consists of four direct-mapped caches operated in parallel.

FIGURE 9.15 Organization of set-associative cache



Example – Cache Size

A 4-way set associative cache uses 64-bit words. Each cache line is composed of 4 words. There are 8,192 lines. How big is the cache?

1. The cache has lines of four 64-bit words; that is, 32 bytes/line.
2. There are 8,192 lines giving 8,192 x 32 = 2^{18} bytes per direct-mapped cache (256 KBs).
3. The associatively is four which means there are four direct-mapped caches in parallel, giving 4 x 256K = 1Mbyte of cache memory.

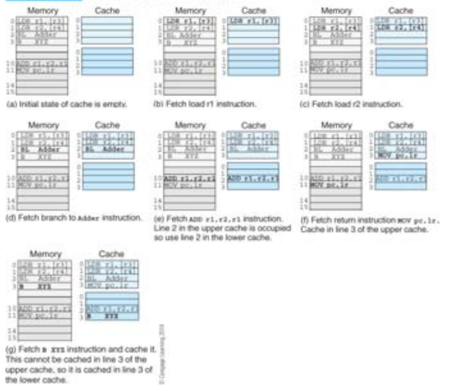
Figure 9.16 repeats the previous example with a set-associative cache that has 4 lines/cache, or 8 lines in total. A line may be cached in the upper (light blue) or lower (dark blue) direct-mapped cache.

Everything is the same until 9.16(e) when ADD **r1**,r2,r1 at address 10 is mapped onto line 2 (set size 4) currently occupied by BL Adder.

The corresponding location in the *second* cache in the associative pair is free and, therefore, the instruction can be cached in location 2 of the lower cache without ejecting line 2 from the upper cache.

In 9.16(f) the MOV **pc**,lr has a line 3 address and is cached in the upper cache. When the B XYZ instruction in line 3 of the main memory is executed, line 3 in the upper cache is taken and it is placed in line 3 in the lower cache.

FIGURE 9.16 Behavior of a set-associative cache


Memory
Cache
LDR r1,[r3]
LDR r2,[r4]
BL Adder
B XYZ
ADD r1,r2,r1
MOV pc,lr
(a) Initial state of cache is empty.
(b) Fetch load r1 instruction.
(c) Fetch load r2 instruction.
(d) Fetch branch to Adder instruction.
(e) Fetch ADD r1,r2,r1 instruction. Line 2 in the upper cache is occupied so use line 2 in the lower cache.
(f) Fetch return instruction MOV pc,lr. Cache in line 3 of the upper cache.
(g) Fetch B XYZ instruction and cache it. This cannot be cached in line 3 of the upper cache, so it is cached in line 3 of the lower cache.
© Cengage Learning 2014

Associativity

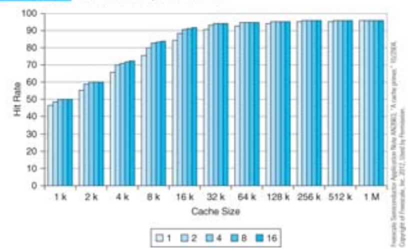
Table from Interprice integrated (IDT) demonstrates the effect of cache organization on the miss ratio. The miss ratio has been normalized by dividing it by the miss ratio of a direct-mapped cache, to demonstrate the results relative to a direct mapped-cache. A four-way set associative cache is about 30% better than a direct-mapped cache. Increasing the associatively has little further improvement on the cache's performance.

Cache organization	Normalized miss ratio
Direct-mapped	1.0
2-way set associative	0.78
4-way set associative	0.70
8-way set associative	0.67
Fully associative	0.66

Figure 9.17 from a Freescale Semiconductor application note demonstrates the relationship between associativity and hit rate for varying cache sizes for a GCC complier.

Depending on the size of the program once the caches reach to a certain amoun the effect of associativity becomes insignificant.

FIGURE 9.17 Set-associativity and cache size



Categories of miss

- When calculating the efficiency of a cache system, we are interested in the hit rate because it's the hits that make the cache effective.
- When designing cache systems we are interested in the miss ratio because there is only one source of hits (the data was in the cache), whereas there are several sources of misses. We can improve cache performance by asking, "Why wasn't the data that resulted in the miss not already in the cache?“

Types of miss

- Cache misses are divided into three classes,
  - compulsory,
  - capacity
  - conflict.
- The compulsory miss cannot be avoided. A compulsory miss occurs because of the inevitable miss on the first access to a block of data. Some processors do avoid the compulsory miss by anticipating an access to data and bringing it into cache before it is required. This is a form of pre-fetching mechanism.

Types of miss

- Another cause of a miss is the capacity miss.
- In this case a miss takes place because the working set (i.e., the lines that make up the current program) is larger than the cache and all the required data cannot reside in the cache.
- If the program is sufficiently large, there comes a point when the cache is full and the next access causes a capacity miss.
- Now the system has to load new data into the cache and eject old data to make room.

Types of miss

- A third form of cache miss is the conflict miss.
- This is the most wasteful type of miss because it happens when the cache is not yet full, but the new data has to be rejected because of the side effects of cache organization.
- A conflict miss occurs in an m-way associative cache when all m associative pages already contain a line i and a new line i is to be cached. Conflict misses account for between 20% and 40% of all misses in direct mapped systems.

Summary

- Paged memory allocation
  - Efficient use of memory
  - Allocate jobs in noncontiguous memory locations
  - Problems
    - Increased overhead
    - Internal fragmentation
- Demand paging scheme
  - Eliminates physical memory size constraint
  - LRU provides slightly better efficiency (compared to FIFO)
- Segmented memory allocation scheme
  - Solves internal fragmentation problem

Summary (cont'd.)

- Segmented/demand paged memory
  - Problems solved
    - Compaction, external fragmentation, secondary storage handling
- Virtual memory
  - Programs execute if not stored entirely in memory
  - Job's size no longer restricted to main memory size
- Cache memory
  - CPU can execute instruction faster

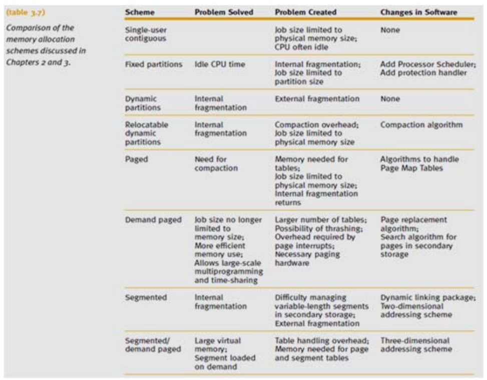
Scheme	Problem Solved	Problem Created	Key Software Changes
Single-user contiguous	Not applicable	Job size limited to physical memory size; CPU often idle	Not applicable
Fixed partitions	Idle CPU time	Internal fragmentation; job size limited to partition size	Add Processor Scheduler; add protection handler
Dynamic partitions	Internal fragmentation	External fragmentation	Algorithms to manage partitions
Relocatable dynamic partitions	External fragmentation	Compaction overhead; job size limited to physical memory size	Algorithms for compaction
Paged	Need for compaction	Memory needed for tables; Job size limited to physical memory size; internal fragmentation returns	Algorithms to manage tables

(table 3.7)
The big picture. Comparison of the memory allocation schemes discussed in Chapters 2 and 3.
© Cengage Learning 2014

Scheme	Problem Solved	Problem Created	Key Software Changes
Demand paged	Job size limited to memory size; inefficient memory use	Large number of tables; possibility of thrashing; overhead required by page interrupts; paging hardware added	Algorithm to replace pages; algorithm to search for pages in secondary storage
Segmented	Internal fragmentation	Difficulty managing variable-length segments in secondary storage; external fragmentation	Dynamic linking package; two-dimensional addressing scheme
Segmented/ demand paged	Segments not loaded on demand	Table handling overhead; memory needed for page and segment tables	Three-dimensional addressing scheme

(table 3.7) (cont'd.)
The big picture. Comparison of the memory allocation schemes discussed in Chapters 2 and 3.
© Cengage Learning 2014

Summary (cont'd.)



(table 3.7)

Comparison of the memory allocation schemes discussed in Chapters 2 and 3.

Scheme	Problem Solved	Problem Created	Changes in Software
Single-user contiguous		Job size limited to physical memory size; CPU often idle	None
Fixed partitions	Idle CPU time	Internal fragmentation; Job size limited to partition size	Add Processor Scheduler; Add protection handler
Dynamic partitions	Internal fragmentation	External fragmentation	None
Relocatable dynamic partitions	Internal fragmentation	Compaction overhead; Job size limited to physical memory size	Compaction algorithm
Paged	Need for compaction	Memory needed for tables; Job size limited to physical memory size; Internal fragmentation returns	Algorithms to handle Page Map Tables
Demand paged	Job size no longer limited to memory size; More efficient memory use; Allows large-scale multiprogramming and time-sharing	Larger number of tables; Possibility of thrashing; Overhead required by page interrupts; Necessary paging hardware	Page replacement algorithm; Search algorithm for pages in secondary storage
Segmented	Internal fragmentation	Difficulty managing variable-length segments in secondary storage; External fragmentation	Dynamic linking package; Two-dimensional addressing scheme
Segmented/ demand paged	Large virtual memory; Segment loaded on demand	Table handling overhead; Memory needed for page and segment tables	Three-dimensional addressing scheme