Operating System
Chapter 5: Memory Management

Prepared By:
Amit K. Shrivastava
Asst. Professor
Nepal College Of Information Technology

Introduction:

- Memory is central to the operation of a modern computer system. Memory consists of a large array of words or bytes, each with its own address.
- Programs and data must be in main storage in order to be run or referenced directly. Secondary storage – most commonly disk, drum, and tape - provides massive, inexpensive capacity for the abundance of programs and data that must be kept readily available for processing.

Storage Organization:

- By storage organization we mean the manner in which the main storage is viewed. For example – Do we place only a single user in the main storage, or do we place several users in it at the same time? If several user programs are in main storage at the same time, do we give each of them the same amount of space, or do we divide main storage in to portions, called partitions of different sizes? Etc.

Storage Management:

- Regardless of what storage organization scheme we adopt for a particular system, we must decide what strategies to use to obtain optimal performance. Storage management strategies determine how a particular storage organization performs under various policies.
- For example – when do we get a new program to place in the memory? Do we get it when the system specifically asks for it, or do we attempt to anticipate the system's request? Where in the main storage do we place the next program to be run? Etc.
- Systems have been implemented using each of these storage management strategies.

Storage Hierarchy:

In 1960 storage hierarchy is extended by one more level, i.e the cache which is a high-speed storage that is much faster than main storage

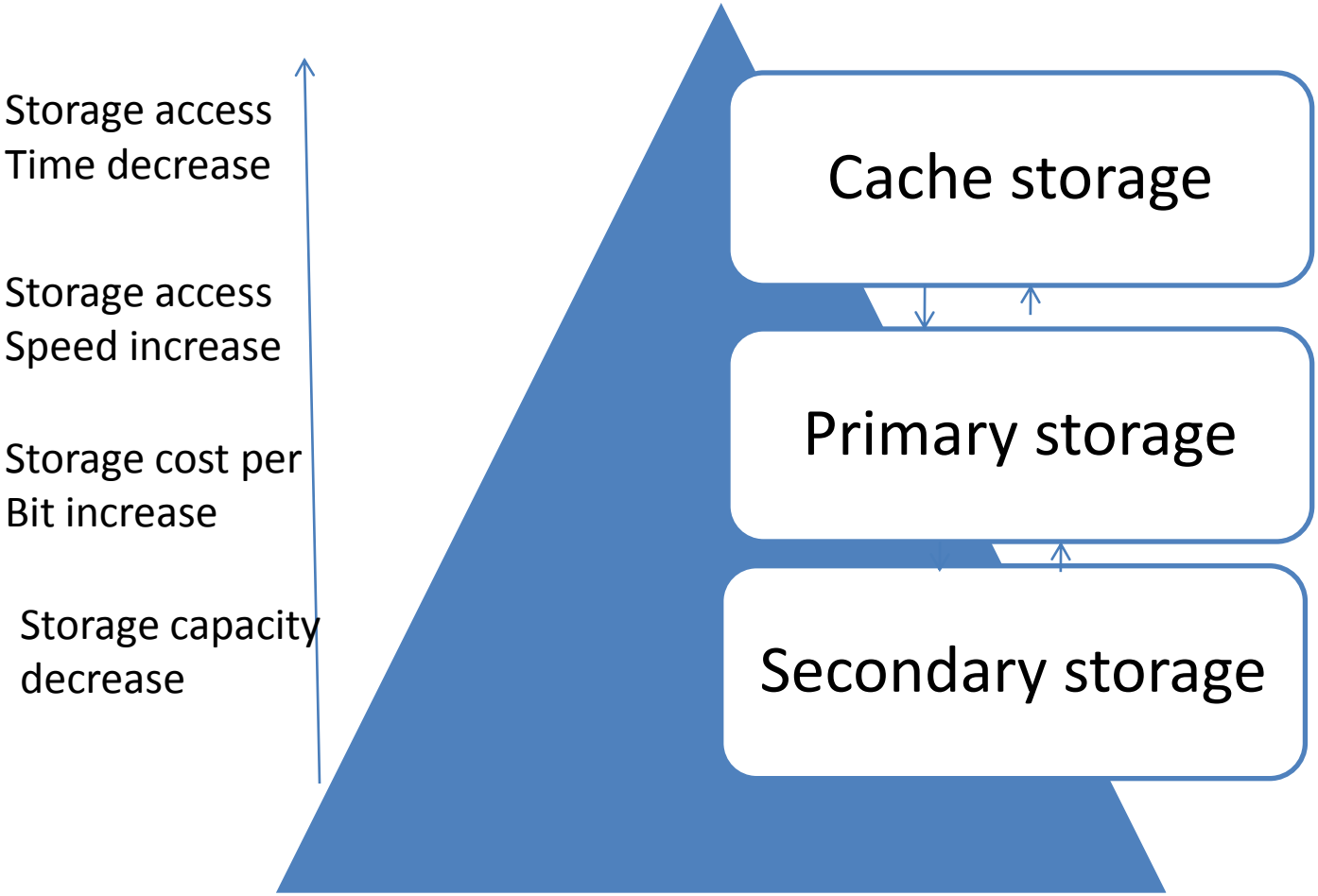


Fig: Hierarchical storage organization

Storage Management Strategies:

- Storage Management strategies are generated for obtaining the best possible use of the main storage. They are divided into following categories:
- 1. Fetch Strategies
  - a) Demand
  - b) Anticipatory

  2. Placement Strategies

  3. Replacement Strategies

- Fetch Strategies are concerned with when to obtain the next piece of program or data for transfer to main storage from secondary storage.
- Placement Strategies are concerned with determining where in main storage to place an incoming program. E.g. first-fit, best-fit, and worst-fit
- Replacement Strategies are concerned with determining which piece of program or data to displace to make room for incoming programs.

Contiguous Vs Noncontiguous Storage Allocation:

- The earliest computing system required contiguous allocation - each program had to occupy a single contiguous block of stotage location.
- In noncontiguous storage allocation, a program is divided into several blocks or segments that may be placed throughout main storage in pieces not necessarily adjacent to one another.
- It is more difficult for an operating system to control noncontiguous storage allocation. The benefit is that if main storage has many small holes available instead of a single large hole, then the operating system can often load and execute a program that would otherwise need to wait.

Logical and Physical Memory:

- An address generated by the CPU is commonly referred to as a **logical address,** whereas an address seen by the memory unit-that is, the one loaded into the **memory-address register** of the memory-is commonly referred to as a **physical address.**
- Logical and physical addresses are the same in compile-time and load-time address-binding schemes; logical (virtual) and physical addresses differ in execution-time address-binding scheme.

Fragmentation:

- **External Fragmentation –** total memory space exists **to** satisfy a request, but it is not contiguous.
- **Internal Fragmentation –** allocated memory may be slightly larger than requested memory; this size difference is memory internal to a partition, but not being used.
- Reduce external fragmentation by compaction
  - ✦ Shuffle memory contents to place all free memory together in one large block.
  - ✦ Compaction is possible *only if relocation is dynamic, and is* done at execution time.
  - ✦ I/O problem
    - ✔ Latch job in memory while it is involved in I/O.
    - ✔ Do I/O only into OS buffers.

Fixed Partition Multiprogramming

- In a multiprogramming environment, several programs reside in primary memory at a time and the CPU passes its control rapidly between these programs. One way to support multiprogramming is to divide the main memory into several partitions each of which is allocated to a single process. Depending upon how and when partitions are created, there may be two types of memory partitioning: (1) Static(fixed) and (2) Dynamic(variable).
- Static partitioning implies that the division of memory into number of partitions and its size is made in the beginning (during the system generation process) and remain fixed thereafter.
- The basic approach here is to divide memory into several fixed size partitions where each partition will accommodate only one program for execution. The number of programs (i.e. degree of multiprogramming) residing in memory will be bound by the number of partition When a program terminates, that partition is free for another program waiting in a queue.

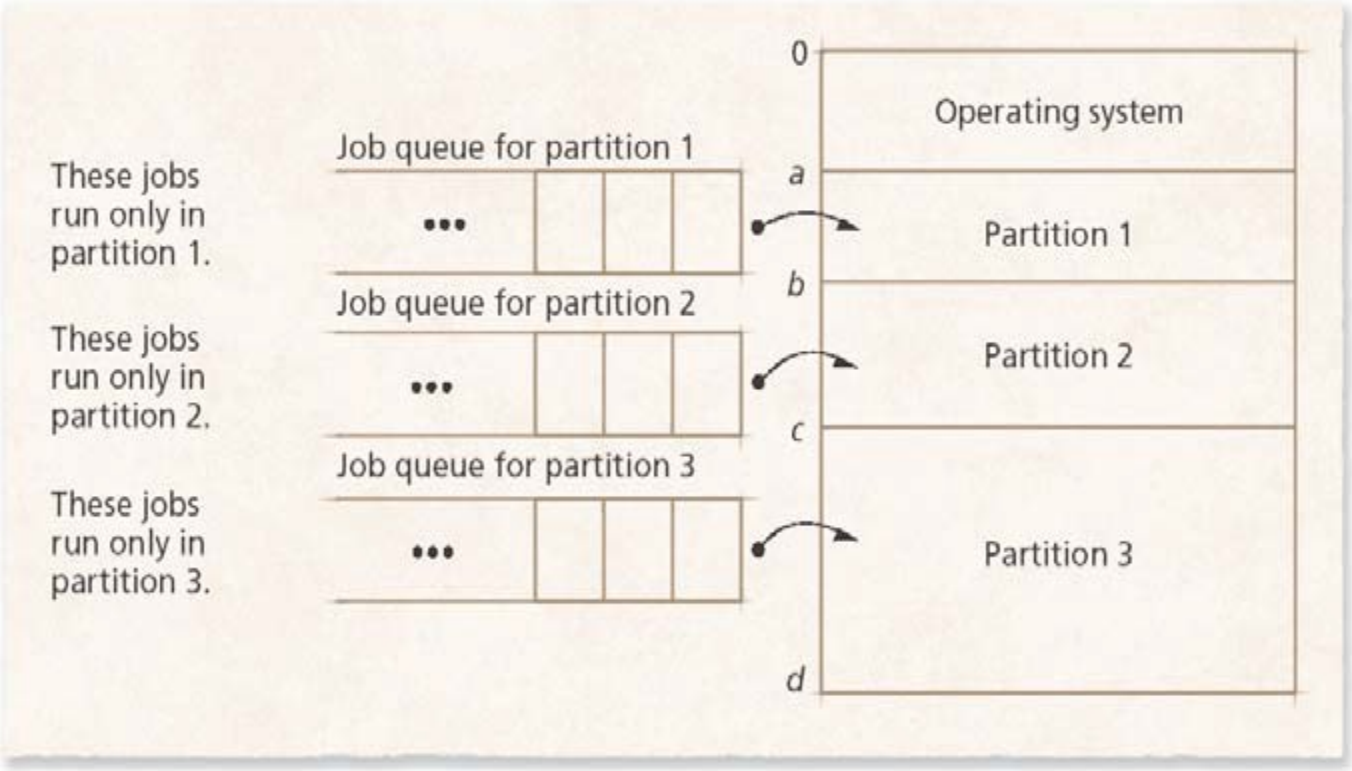


Figure : Fixed-partition multiprogramming with absolute translation and loading.

- Drawbacks to fixed partitions
  - Early implementations used absolute addresses
    - If the requested partition was full, code could not load
    - Later resolved by relocating compilers

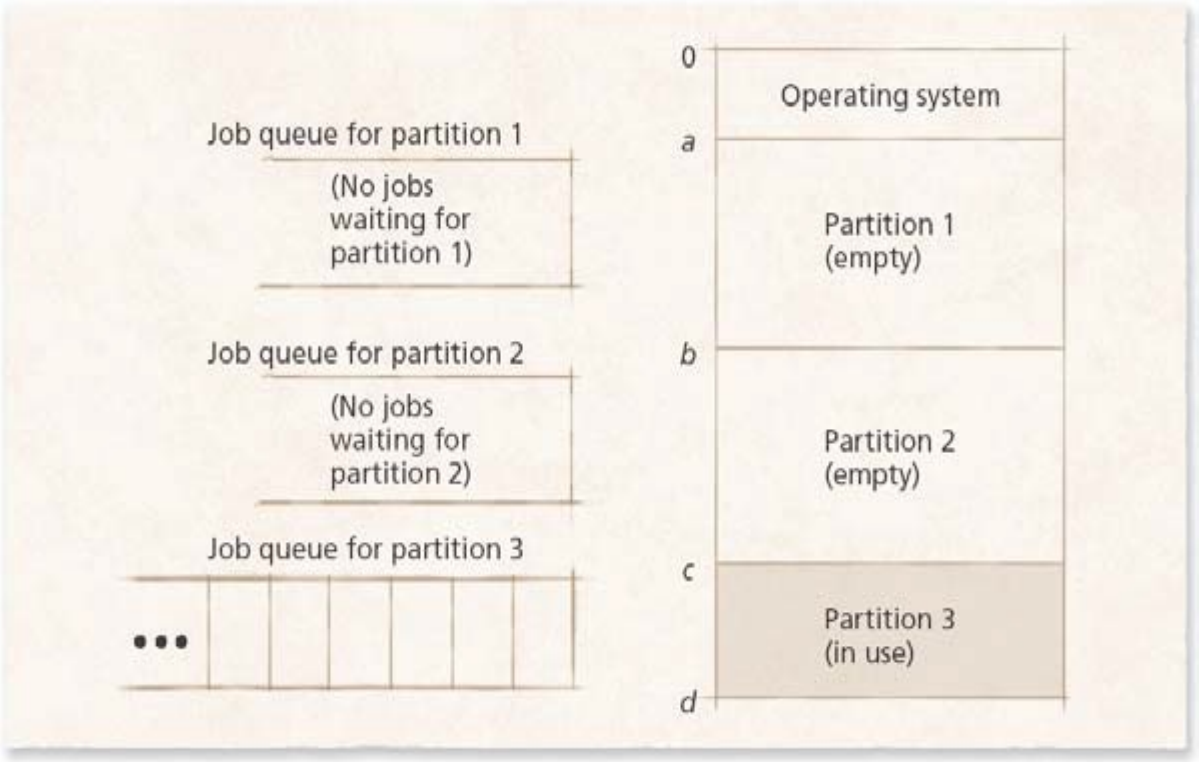


Figure : Memory waste under fixed-partition multiprogramming with absolute translation and loading.

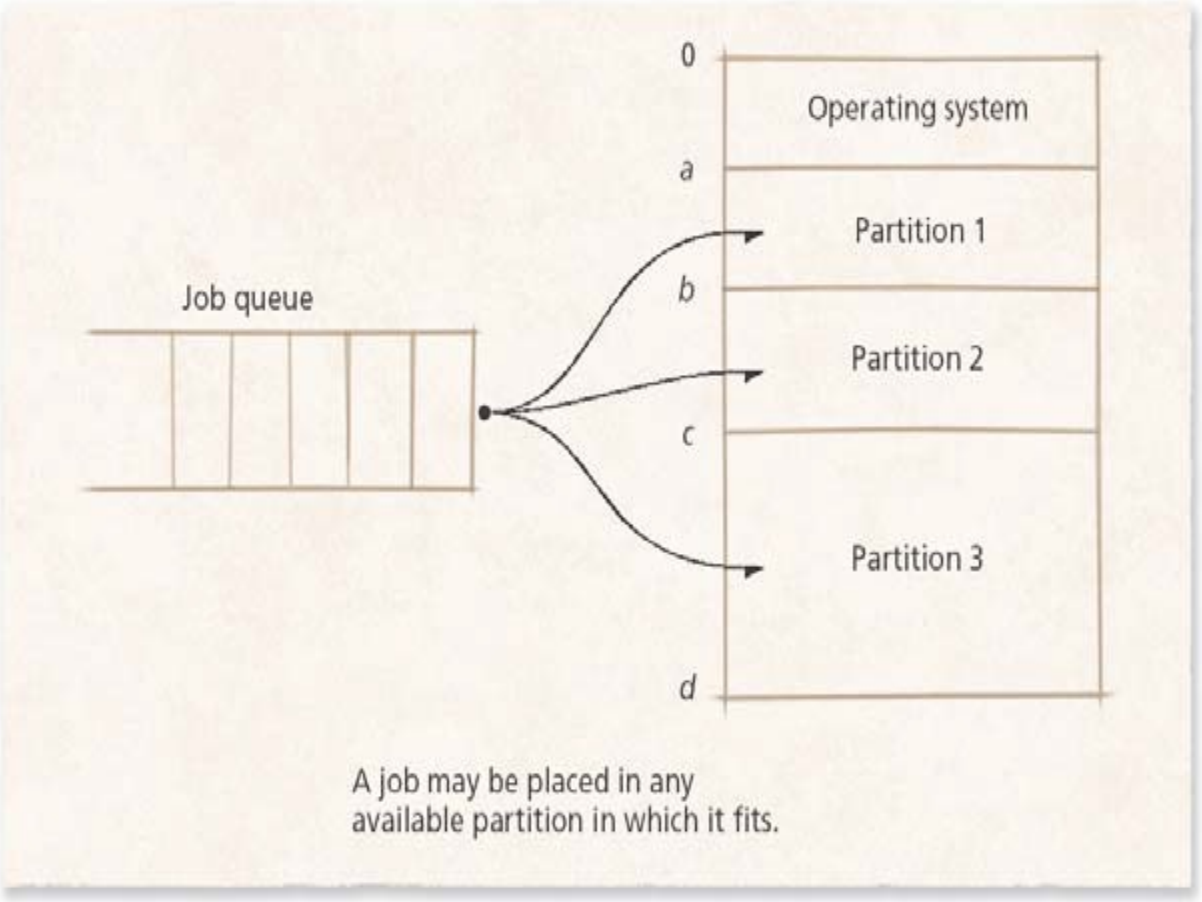


Figure : Fixed-partition multiprogramming with relocatable translation and loading.

Variable-Partition Multiprogramming

- System designers found fixed partitions too restrictive
  - Internal fragmentation
  - Potential for processes to be too big to fit anywhere
  - Variable partitions designed as replacement

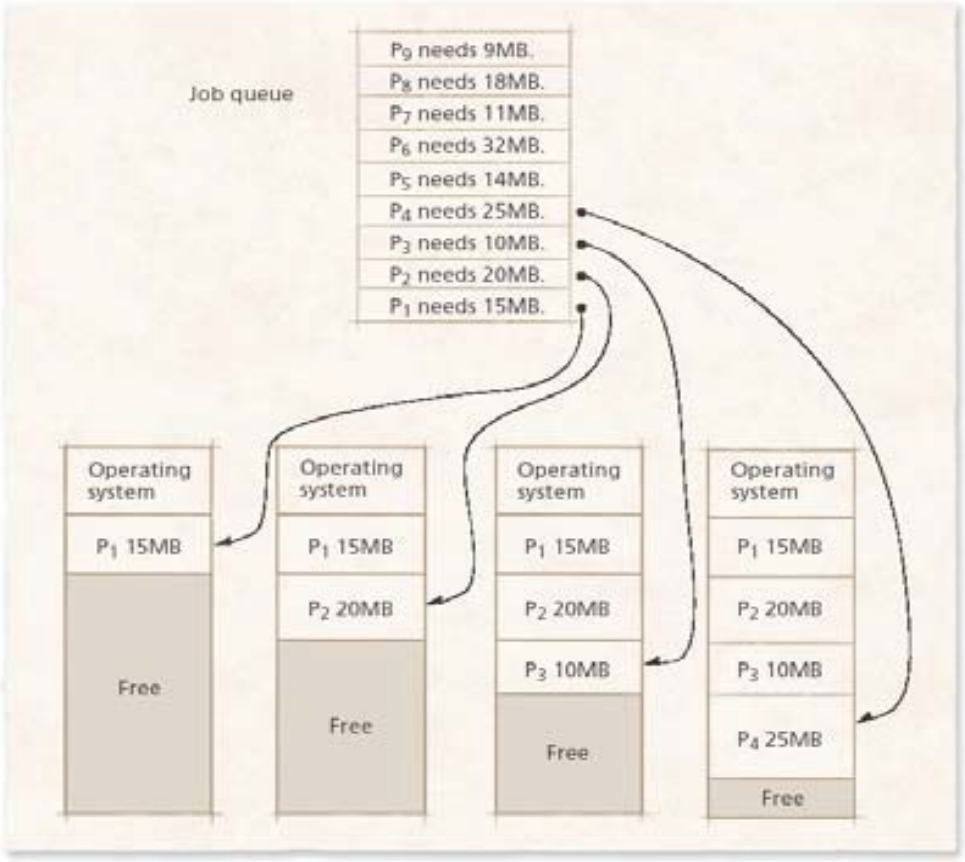


Figure : Initial partition assignments in variable-partition programming.

Variable-Partition Characteristics

- Jobs placed where they fit
  - – No space wasted initially
  - – Internal fragmentation impossible
    - Partitions are exactly the size they need to be
  - – External fragmentation can occur when processes removed
    - Leave holes too small for new processes
    - Eventually no holes large enough for new processes

Variable-Partition Characteristics

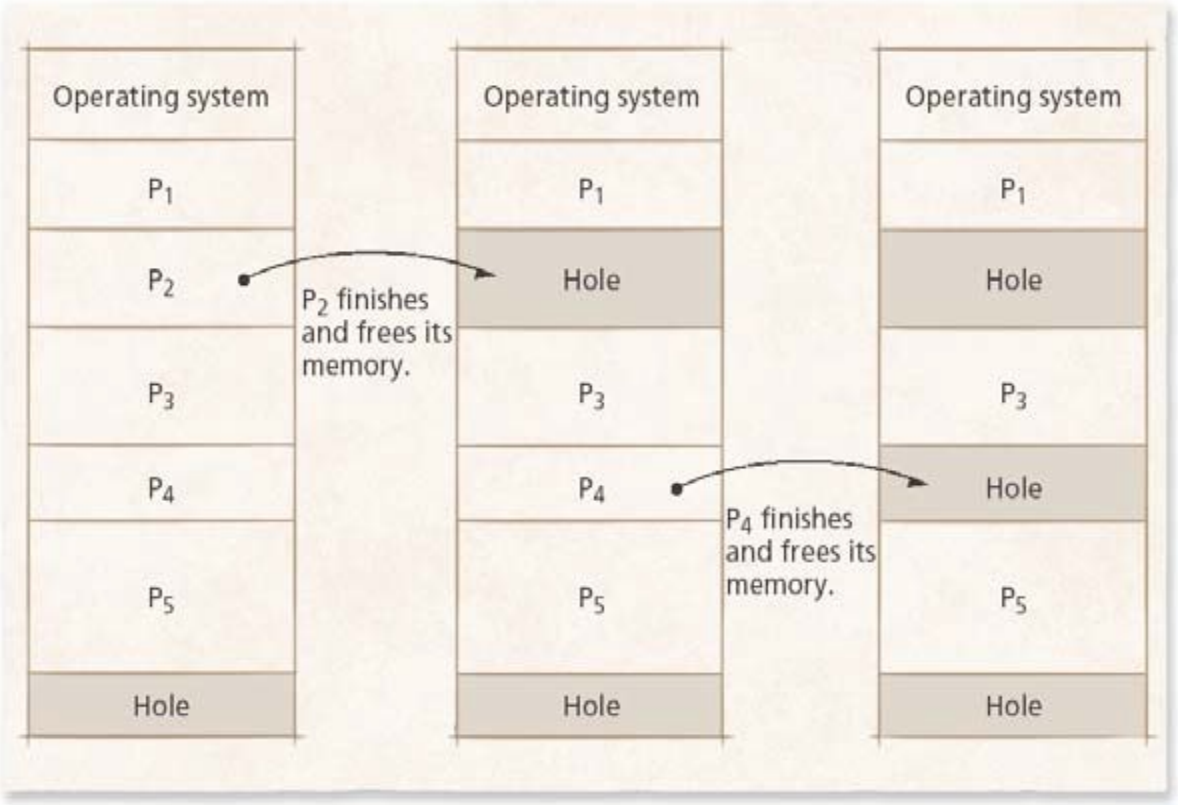


Figure : Memory "holes" in variable-partition multiprogramming.

Variable-Partition Characteristics

- Several ways to combat external fragmentation
  - Coalescing
    - Combine adjacent free blocks into one large block
    - Often not enough to reclaim significant amount of memory
  - Compaction
    - Sometimes called garbage collection (not to be confused with GC in object-oriented languages)
    - Rearranges memory into a single contiguous block free space and a single contiguous block of occupied space
    - Makes all free space available
    - Significant overhead

Variable-Partition Characteristics

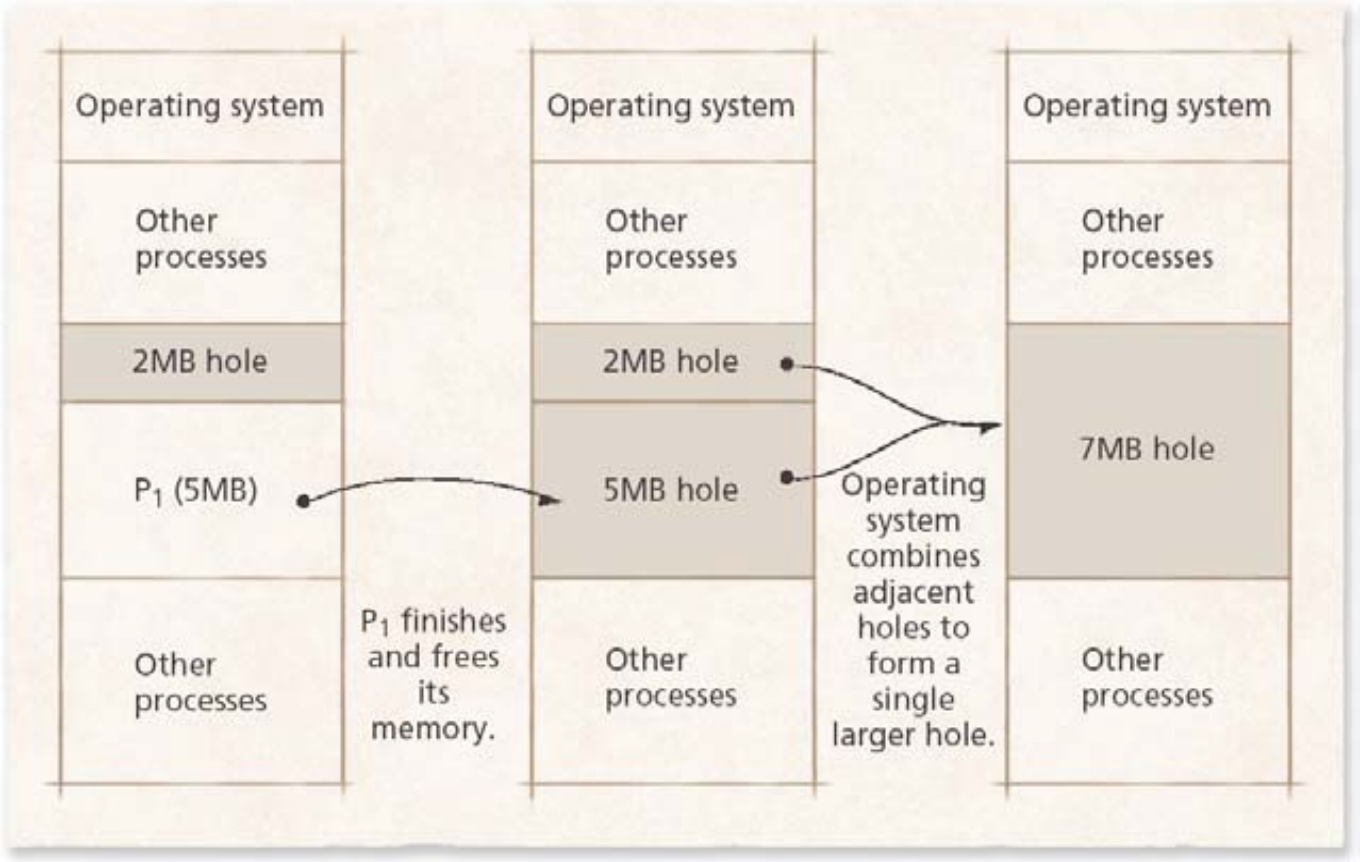


Figure : Coalescing memory "holes" in variable-partition multiprogramming.

Variable-Partition Characteristics

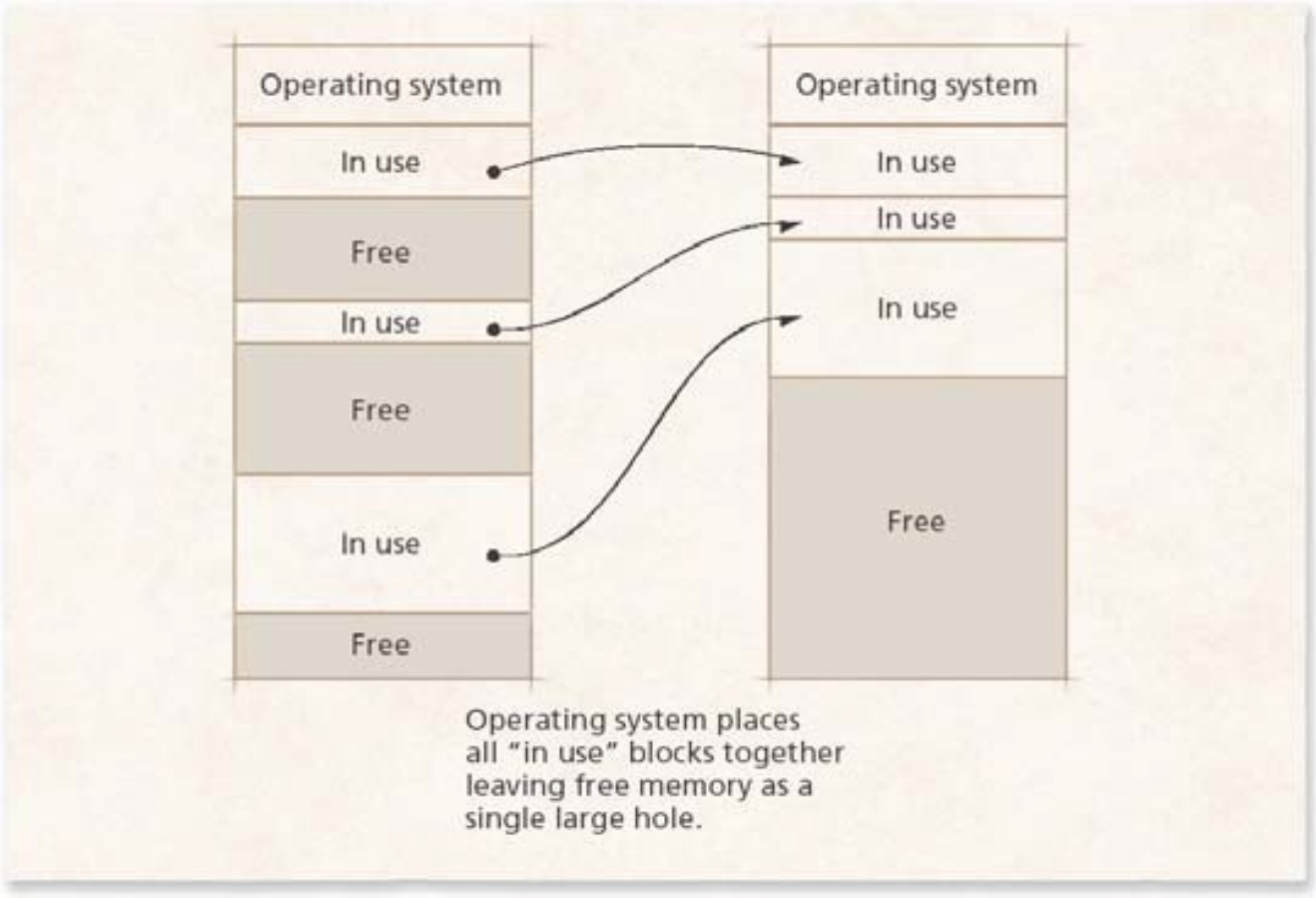


Figure : Memory compaction in variable-partition multiprogramming

Relocation and Protection

- Relocation

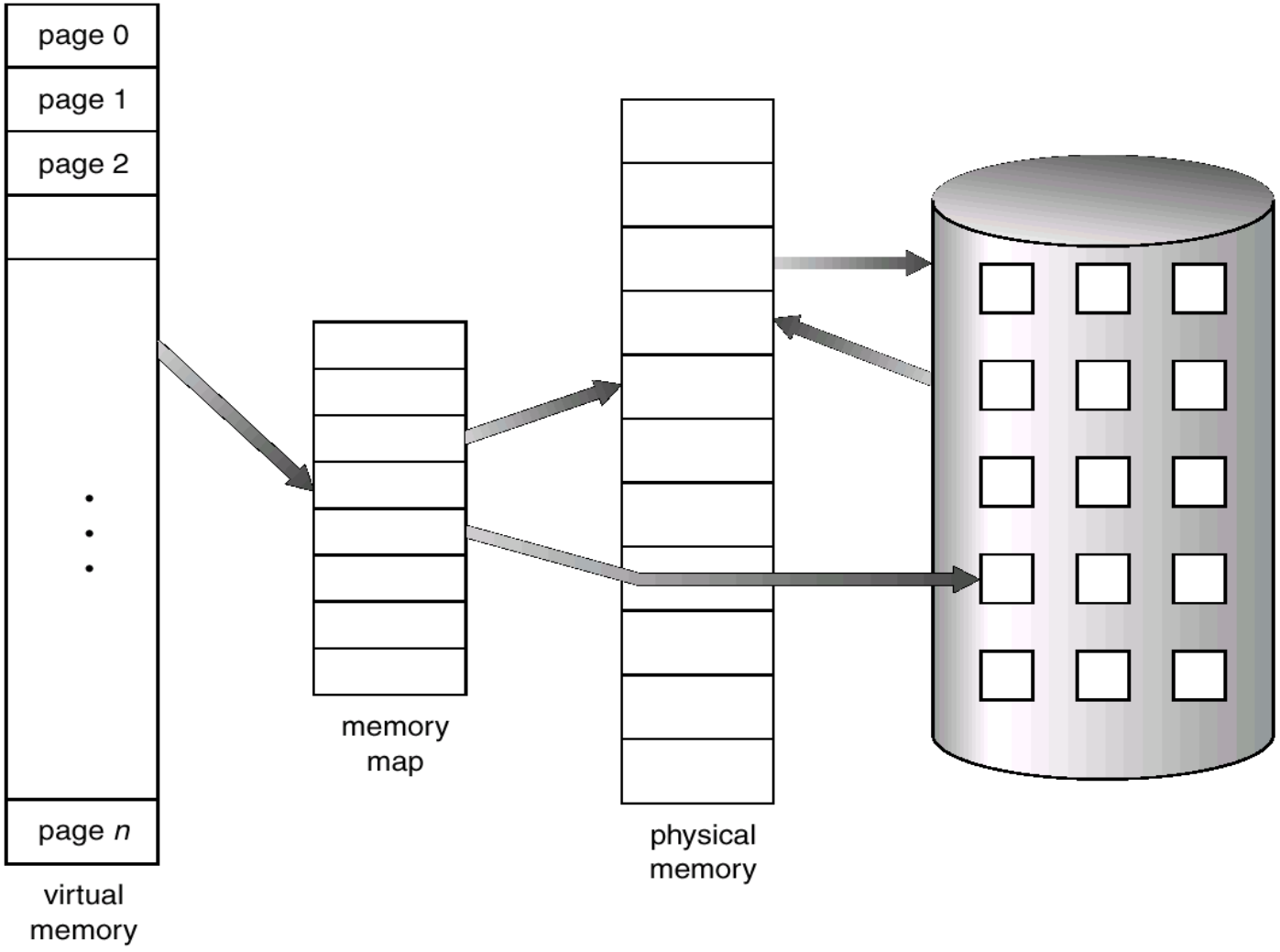
  In systems with virtual memory, programs in memory must be able to reside in different parts of the memory at different times. This is because when the program is swapped back into memory after being swapped out for a while it can not always be placed in the same location. The virtual memory management unit must also deal with concurrency. Memory management in the operating system should therefore be able to relocate programs in memory and handle memory references and addresses in the code of the program so that they always point to the right location in memory.
- Protection

  Processes should not be able to reference the memory for another process without permission. This is called memory protection, and prevents malicious or malfunctioning code in one program from interfering with the operation of other running programs.

Virtual Memory

- Introduction:
  - Solves problem of limited memory space
  - Creates the illusion that more memory exists than is available in system
  - Two types of addresses in virtual memory systems
    - Virtual addresses
      - Referenced by processes
    - Physical addresses
      - Describes locations in main memory
  - Memory management unit (MMU)
    - Translates virtual addresses to physical address

Virtual Memory That is Larger Than Physical Memory



Virtual Memory: Basic Concepts

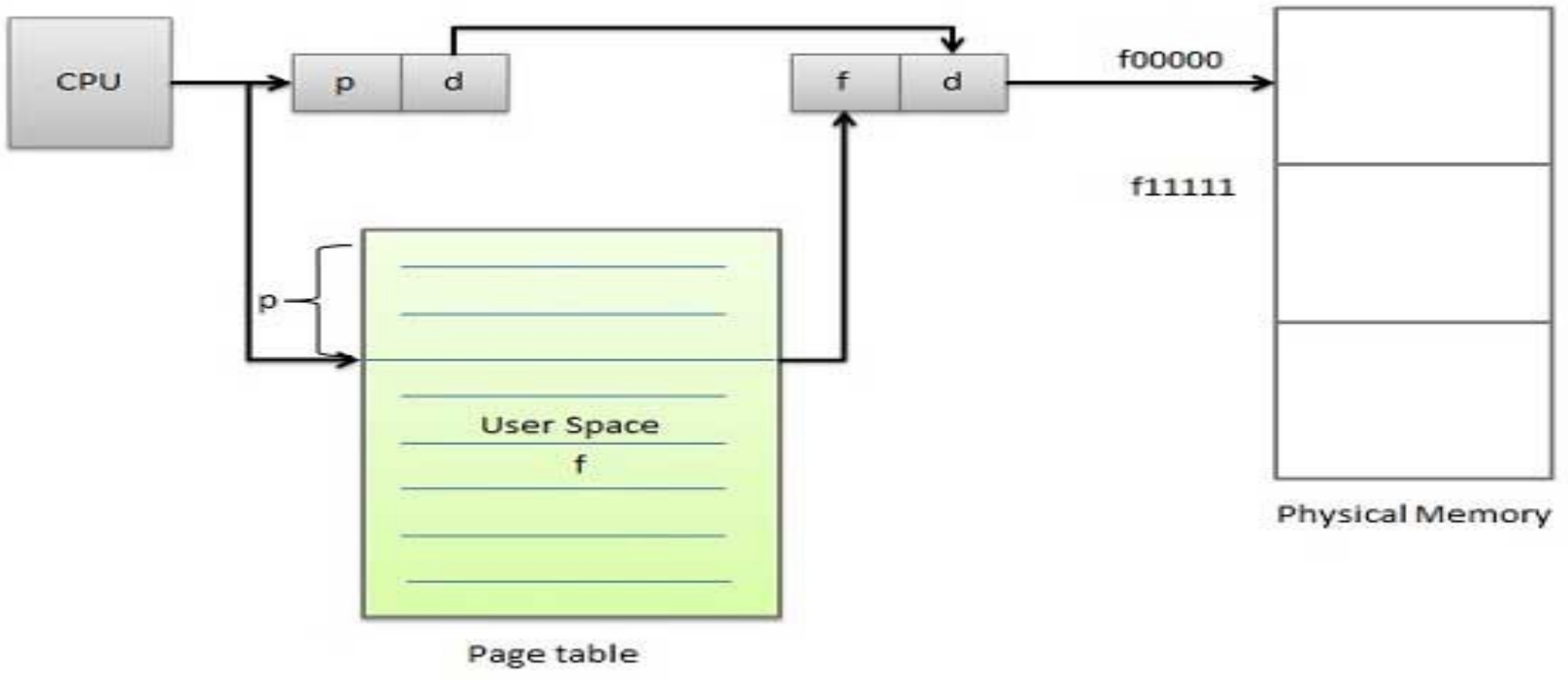
- Virtual address space, *V*
  - Range of virtual addresses that a process may reference
- Real address space, *R*
  - Range of physical addresses available on a particular computer system
- Dynamic address translation (DAT) mechanism
  - Converts virtual addresses to physical addresses during program execution
- *|V| is often much greater than |R|*
  - OS must store parts of *V for each process outside of main* memory
- – Two-level storage
  - OS shuttles portions of *V between main memory (and caches) and* secondary storage

Paging:

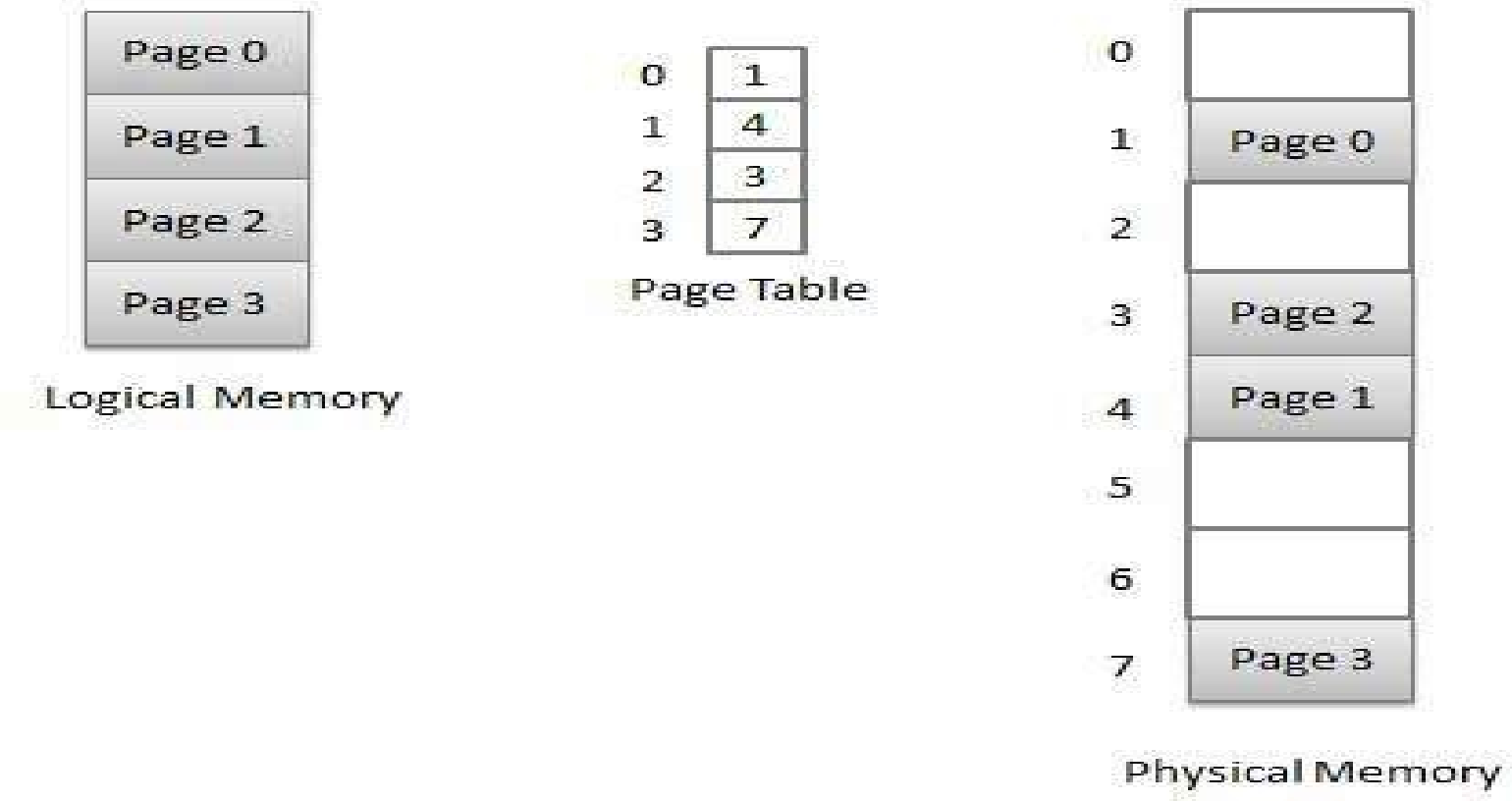
- Logical address space of a process can be noncontiguous; process is allocated physical memory whenever the latter is available.
- Divide physical memory into fixed-sized blocks called **frames**
- (size is power of 2, between 512 bytes and 8192 bytes).
- Divide logical memory into blocks of same size called **pages.**
- Keep track of all free frames.
- To run a program of size *n pages, need to find n free frames* and load program.
- Set up a page table to translate logical to physical addresses.
- Internal fragmentation

Paging(contd..)

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



Following figure show the paging table architecture



Page Table Structure

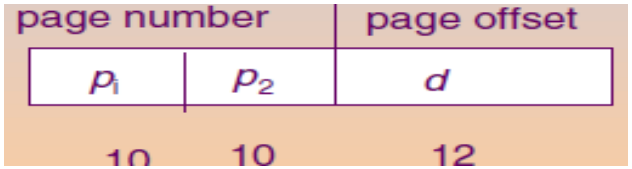
- Hierarchical Paging
- Hashed Page Tables
- Inverted Page Tables

Hierarchical Page Tables

- Break up the logical address space into multiple page tables.
- A simple technique is a two-level page table.

Two-Level Paging Example

- A logical address (on 32-bit machine with 4K page size) is divided into:
  - ✦ a page number consisting of 20 bits.
  - ✦ a page offset consisting of 12 bits.
- Since the page table is paged, the page number is further divided into:
  - ✦ a 10-bit page number.
  - ✦ a 10-bit page offset.
- Thus, a logical address is as follows:

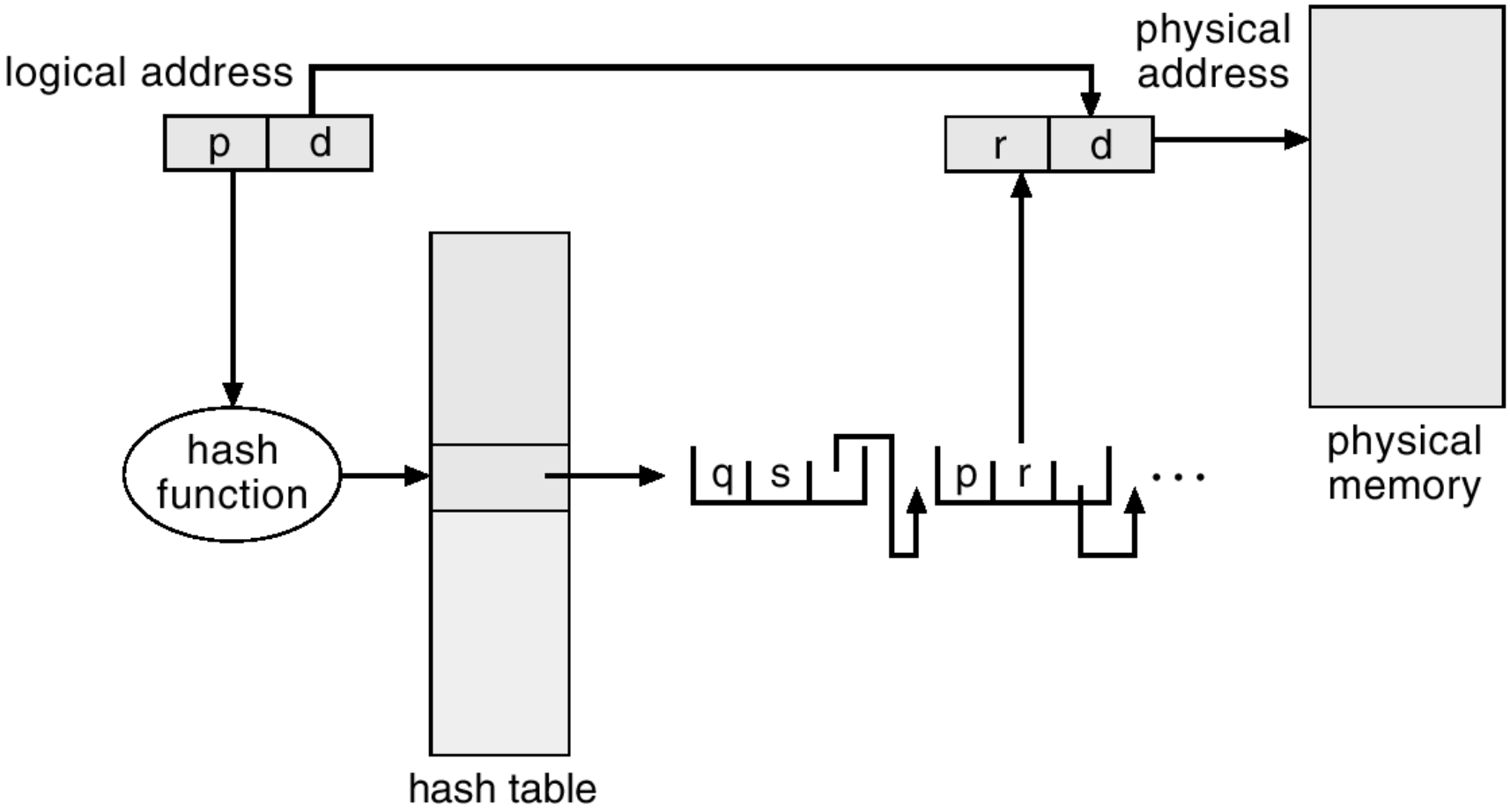


- where *pi is an index into the outer page table, and p2 is the* displacement within the page of the outer page table

Hashed Page Tables

- Common in address spaces > 32 bits.
- The virtual page number is hashed into a page table. This page table contains a chain of elements hashing to the same location.
- Virtual page numbers are compared in this chain searching for a match. If a match is found, the corresponding physical frame is extracted.

Hashed Page Table

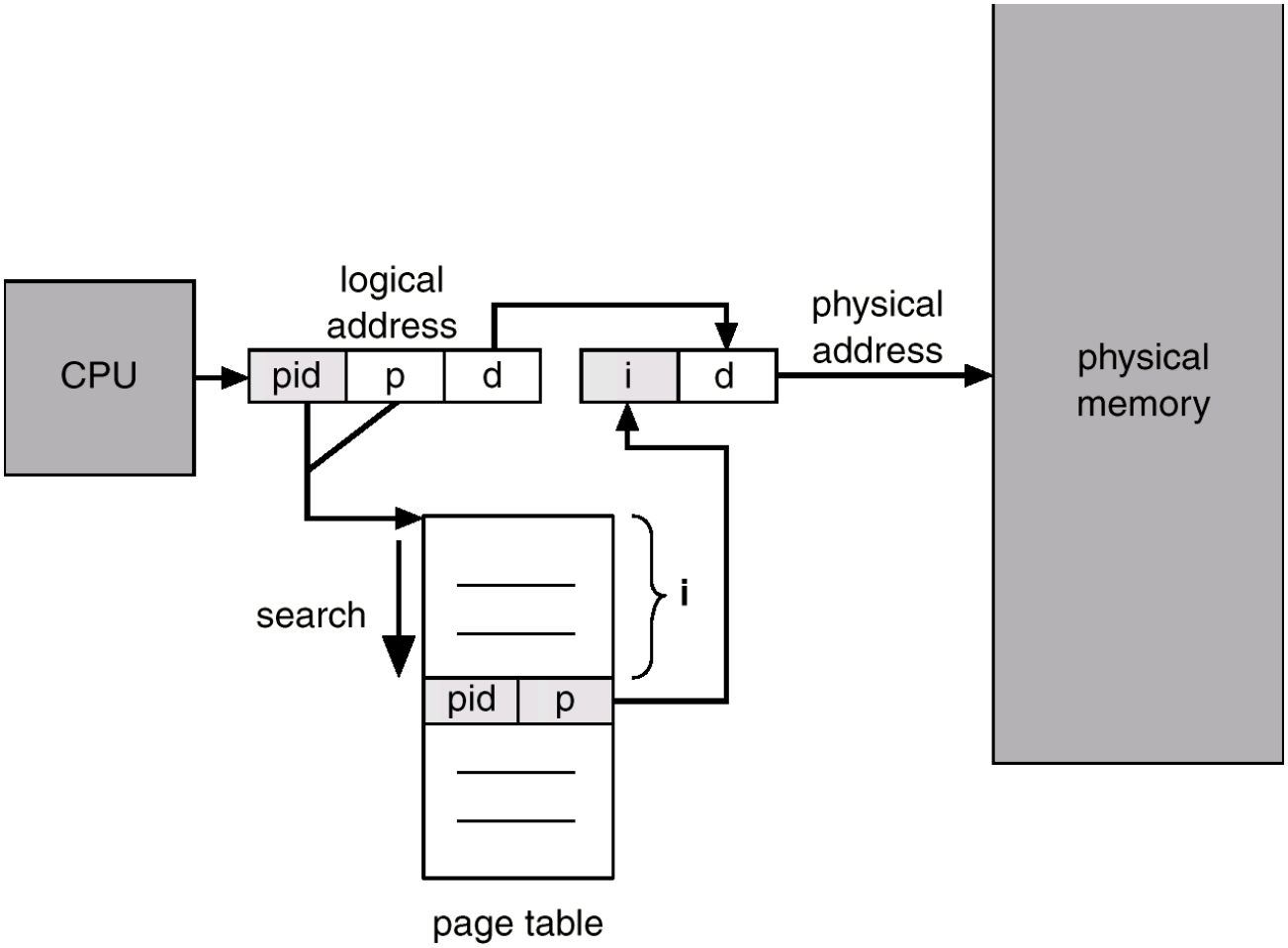


Inverted Page Table

- One entry for each real page of memory.
- Entry consists of the virtual address of the page stored in

  that real memory location, with information about the process that owns that page.
- Decreases memory needed to store each page table, but increases time needed to search the table when a page reference occurs.
- Use hash table to limit the search to one — or at most a few — page-table entries.

Inverted Page Table Architecture



TLB(Translation Look aside Buffers)

- A translation look aside buffer(TLB) is a memory cache that stores recent translations of virtual memory to physical addresses for faster retrieval.
- Each entry in the TLB consists of two parts: a key (or tag) and a value. When the associative memory is presented with an item, the item is compared with all keys simultaneously. If the item is found, the corresponding value field is returned. The search is fast; a TLB lookup in modern hardware is part of the instruction pipeline, essentially adding no performance penalty. To be able to execute the search within a pipeline step, however, the TLB must be kept small. It is typically between 32 and 1,024 entries in size.
- The TLB is used with page tables in the following way. The TLB contains only a few of the page-table entries. When a logical address is generated by the CPU, its page number is presented to the TLB. If the page number is found, its frame number is immediately available and is used to access memory. As just mentioned, these steps are executed as part of the instruction pipeline within the CPU, adding no performance penalty compared with a system that does not implement paging.

TLB(Translation Look aside Buffers(contd…)

- If the page number is not in the TLB (known as a **TLB miss), a memory** reference to the page table must be made. Depending on the CPU, this may be done automatically in hardware or via an interrupt to the operating system. When the frame number is obtained, we can use it to access memory (Figure in next slide). In addition, we add the page number and frame number to the TLB, so that they will be found quickly on the next reference. If the TLB is already full of entries, an existing entry must be selected for replacement.

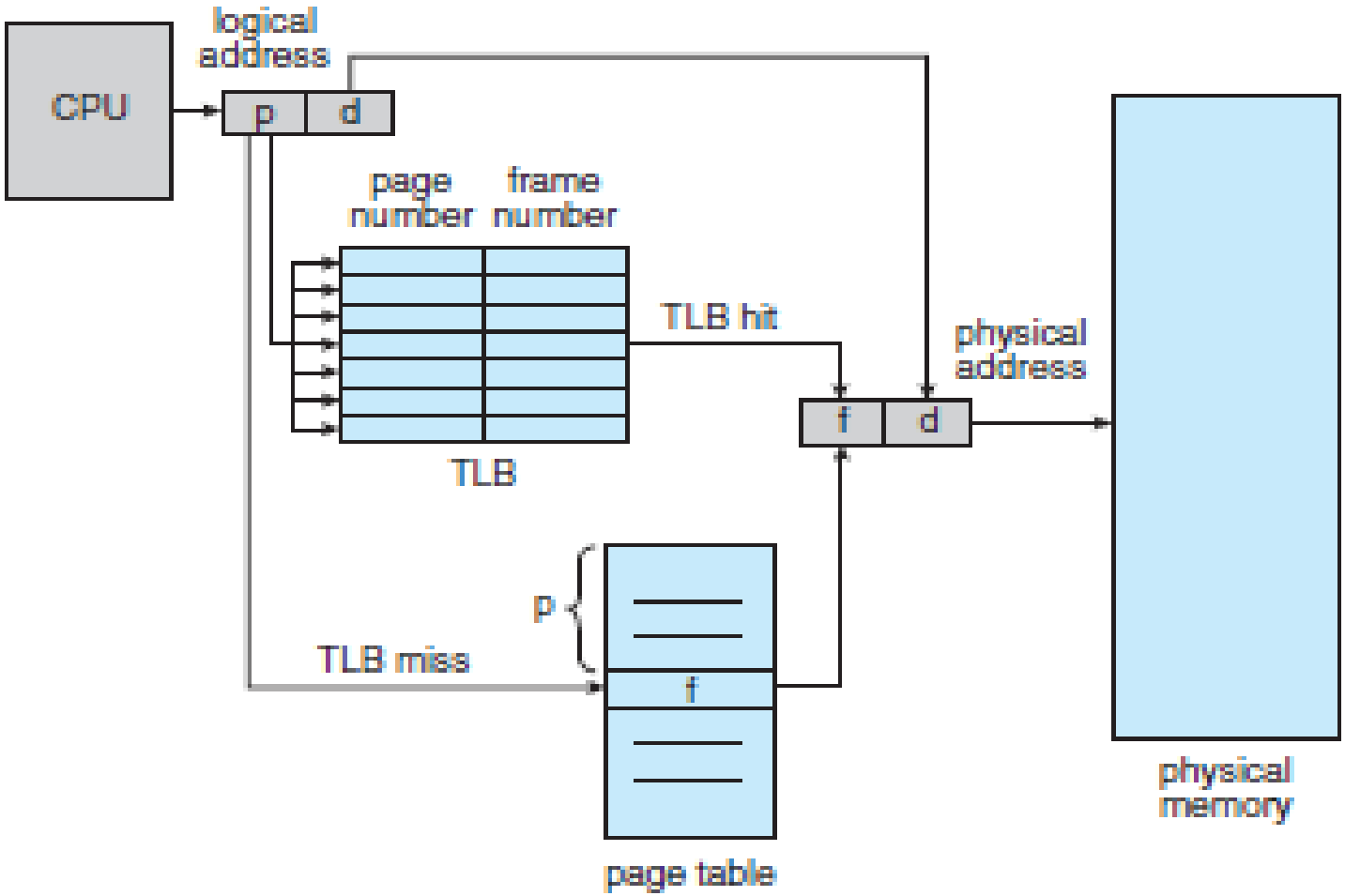
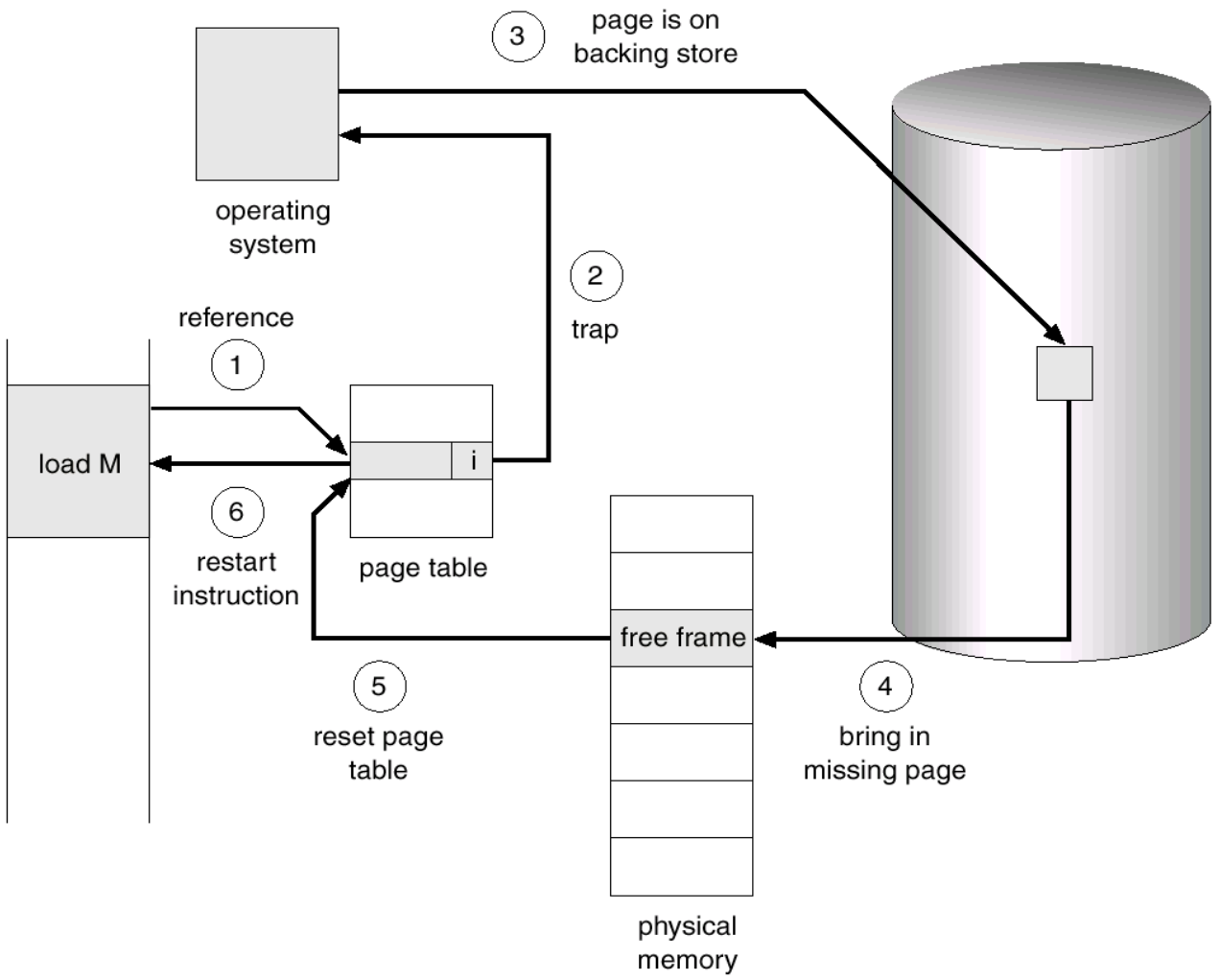


Fig: Paging hardware with TLB

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



Steps in Handling Page Fault:

- The procedure for handling this page fault is straightforward (Figure in previous slide):

1. We check an internal table (usually kept with the process control block) for this process to determine whether the reference was a valid or an invalid memory access.
2. If the reference was invalid, we terminate the process. If it was valid but we have not yet brought in that page, we now page it in.
3. We find a free frame (by taking one from the free-frame list, for example).
4. We schedule a disk operation to read the desired page into the newly allocated frame.
5. When the disk read is complete, we modify the internal table kept with the process and the page table to indicate that the page is now in memory.
6. We restart the instruction that was interrupted by the trap. The process can now access the page as though it had always been in memory.

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.

Page Replacement

- Prevent over-allocation of memory by modifying pagefault service routine to include page replacement.
- Use *modify (dirty) bit to reduce overhead of page* transfers – only modified pages are written to disk.
- Page replacement completes separation between logical memory and physical memory – large virtual memory can be provided on a smaller physical memory.

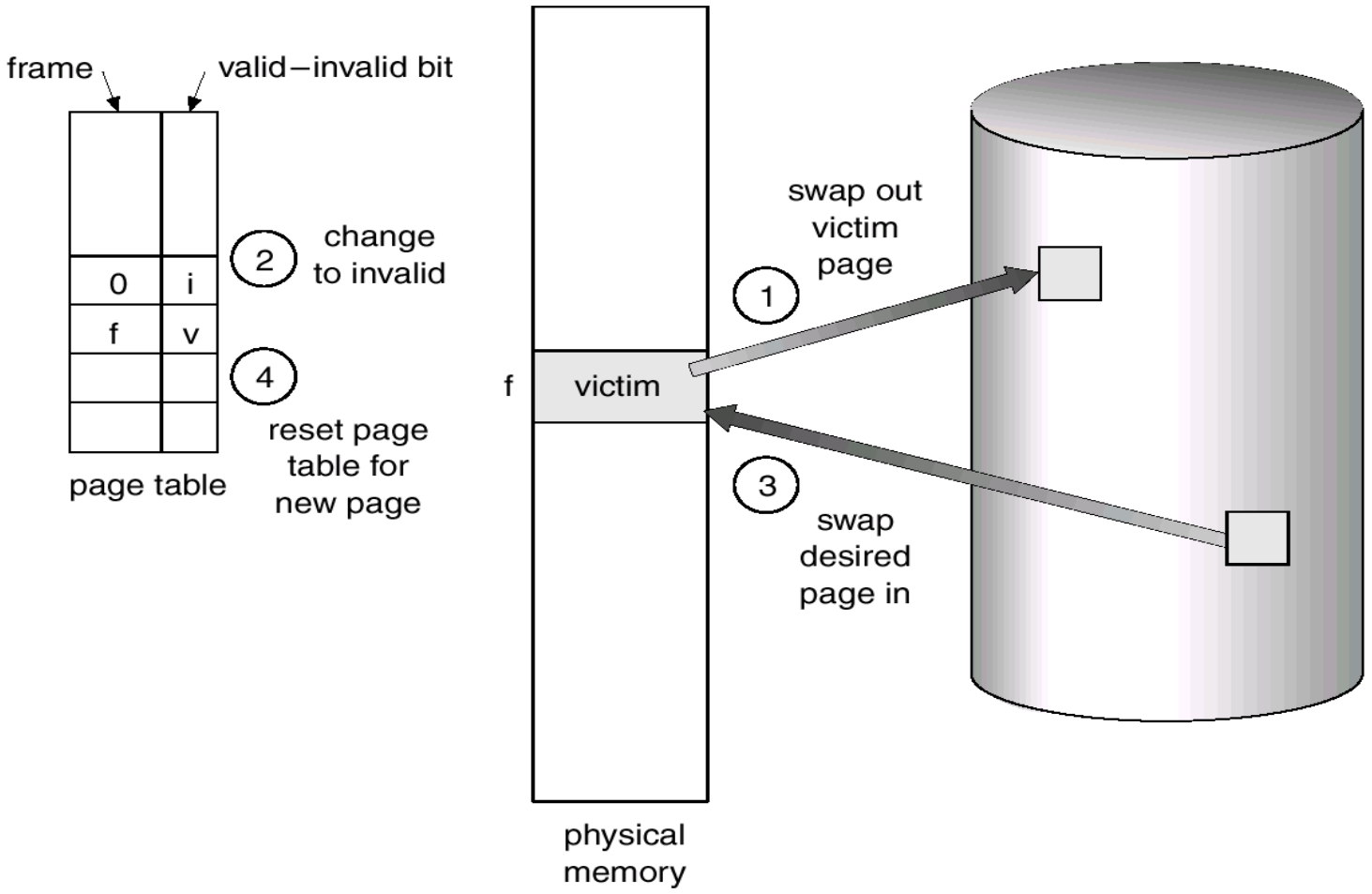
Basic Page Replacement

- Find the location of the desired page on disk.
- 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.*
- Read the desired page into the (newly) free frame.Update the page and frame tables.
- Restart the process.

Page Replacement



Page Replacement Algorithms

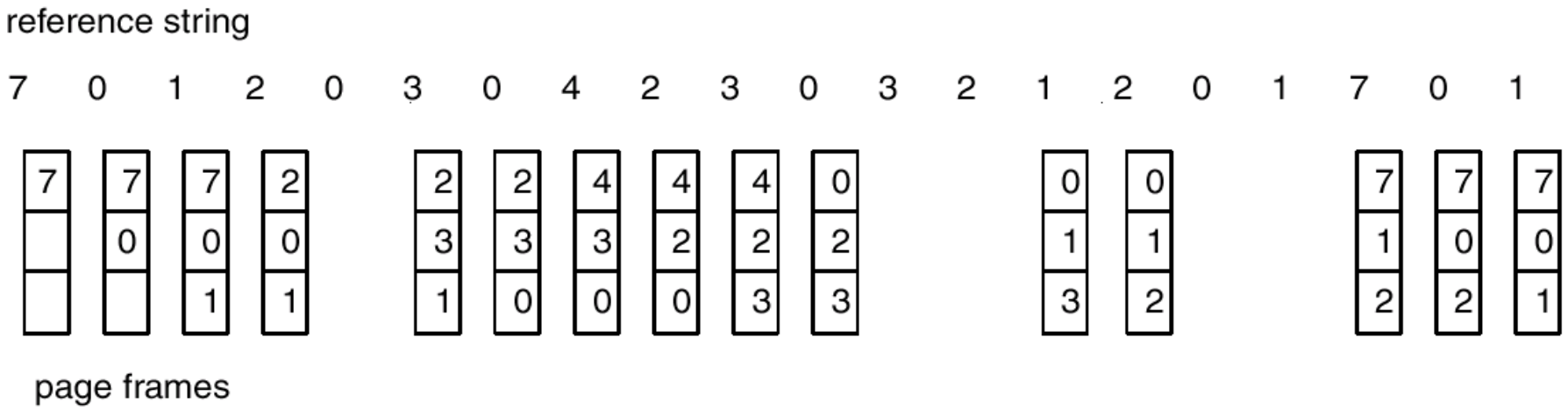
- Want lowest page-fault rate.
- Evaluate algorithm by running it on a particular string of memory references (reference string) and computing thenumber of page faults on that string.
- In all our examples, the reference string is

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

First-In-First-Out (FIFO) Page Replacement

- FIFO page replacement
  – Replace page that has been in the system the longest
  – Likely to replace heavily used pages
  – Can be implemented with relatively low overhead
  – Impractical for most systems

FIFO Page Replacement



3 frames are used (3 pages can be in memory at a time per process)

FIFO Page Replacement

- For our example reference string, our three frames are initially empty. The first three references (7,0,1) cause page faults, and are brought into these empty frames. The next reference (2) replaces page 7, because page 7 was brought in first. Since *0 is the next reference and 0 is already in memory, we have no fault* for this reference. The first reference to **3 results in page *0 being replaced, since*** it was the first of the three pages in memory (0, 1, and 2) to be brought in. Because of this replacement, the next reference, to 0, will fault. Page 1 is then replaced by page 0. This process continues as shown in Figure of previous slide. Every time a fault occurs, we show which pages are in our three frames. There are 15 faults altogether.

FIFO Anomaly

- Belady's (or FIFO) Anomoly

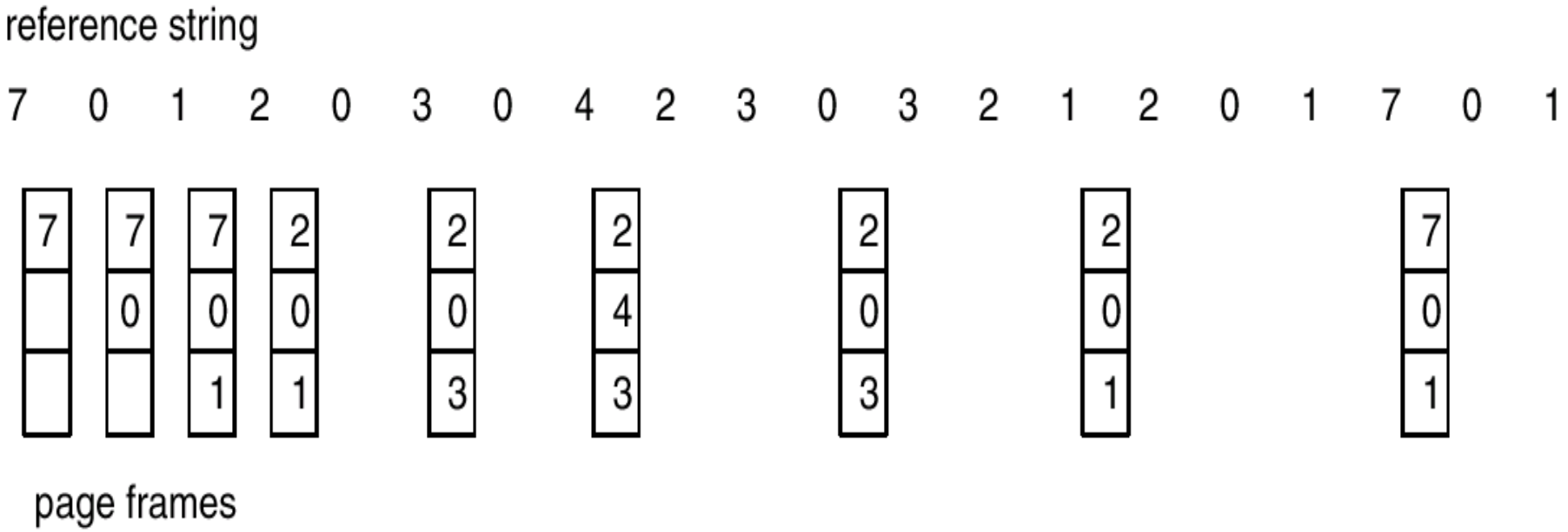
  – Certain page reference patterns actually cause more page faults when number of page frames allocated to a process is increased

FIFO Anomaly

Optimal Page Replacement

- Replace page that will not be used for longest period of time.
- Use of this page replacement algorithm gurantees the lowest possible page rate for a fixed number of frames.
- How do you know this?
- Used for measuring how well your algorithm performs.

Optimal Page Replacement



Optimal Page Replacement

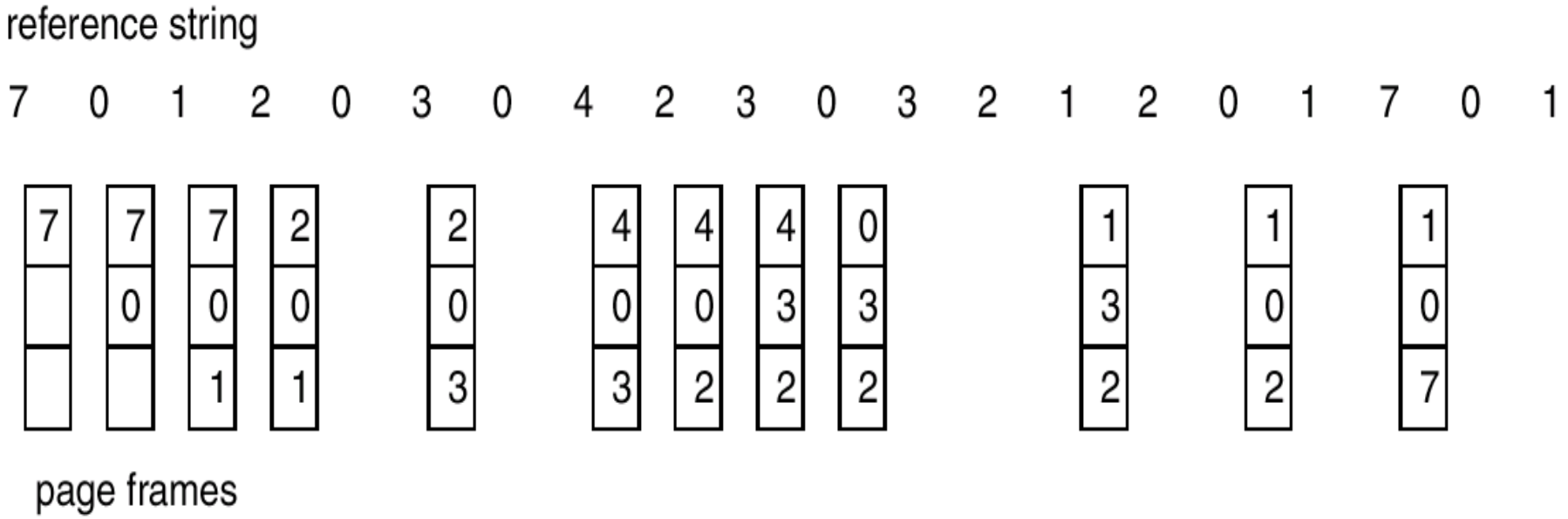
- For example, on our sample reference string, the optimal page-replacement algorithm would yield nine page faults, as shown in Figure of previous slide. The first three references cause faults that fill the three empty frames. The reference to page 2 replaces page 7, because 7 will not be used until reference 18, whereas page 0 will be used at 5, and page 1 at 14. The reference to page 3 replaces page 1, as page 1 will be the last of the three pages in memory to be referenced again. With only nine page faults, optimal replacement is much better than a FIFO algorithm, which had 15 faults.
- Unfortunately, the optimal page-replacement algorithm is difficult to implement, because it requires future knowledge of the reference string.

Least Recently Used (LRU) Algorithm

- LRU page replacement

  – Exploits temporal locality by replacing the page that has spent the longest time in memory without being referenced
- – Can provide better performance than FIFO
- – Increased system overhead
- – LRU can perform poorly if the least-recently used page is the next page to be referenced by a program that is iterating inside a loop that references several pages

LRU Page Replacement



LRU Page Replacement

- The result of applying LRU replacement to our example reference string is shown in Figure of previous slide. The LRU algorithm produces 12 faults. Notice that the first five faults are the same as the optimal replacement. When the reference to page 4 occurs, however, LRU replacement sees that, of the three frames in memory, page 2 was used least recently. The most recently used page is page 0, and just before that page **3 was used. Thus, the LRU algorithm replaces page** 2, not knowing that page 2 is about to be used. When it then faults for page 2,the LRU algorithm replaces page **3 since, of the three pages in memory {0,3,4},** page **3 is the least recently used. Despite these problems, LRU replacement with** 12 faults is still much better than FIFO replacement with 15.

LRU Algorithm (Cont.)

- 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.
- Stack implementation – keep a stack of page numbers in a double link form:
- ✦ Page referenced:

  ✔ move it to the top. So the most recently used page is always at the top of the stack and the least recently used page is always at the bottom.

  ✦ No search for replacement

Not Recently Used Page Replacement Algorithm

-Approximates LRU with little overhead by using referenced bit and modified bit to determine which page has not been used recently and can be replaced quickly

– Can be implemented on machines that lack hardware referenced bit and/or modified bit

Group	Referenced	Modified	Description
Group 1	0	0	Best choice to replace
Group 2	0	1	[Seems unrealistic]
Group 3	1	0	
Group 4	1	1	Worst choice to replace

Figure : Page types under NRU.

Second-Chance Page Replacement Algorithm

- The basic algorithm of second-chance replacement is a FIFO replacement algorithm. When a page has been selected, however, we inspect its reference bit. If the value is 0, we proceed to replace this page. If the reference bit is set to 1, however, we give that page a second chance and move on to select the next FIFO page. When a page gets a second chance, its reference bit is cleared and its arrival time is reset to the current time. Thus, a page that is given a second chance will not be replaced until all other pages are replaced (or given second chances). In addition, if a page is used often enough to keep its reference bit set, it will never be replaced.
- **Clock page replacement**

  – Similar to second chance, but arranges the pages in circular list instead of linear list

Working Set Page Replacement Algorithm

- The set of pages that a process is currently using is known as its working set. If the entire working set is in memory, the process will run without causing many faults until it moves into another execution phase. If the the available memory is too small to hold the entire working set, the process will cause many page faults and run slowly.
- To implement the working set model, it is necessary for the operating system to keep track of which pages are in working set. Having this information also immediately leads to a possible page replacement algorithm: when a page fault occurs, find a page not in the working set and evict it.

WS Clock Page Replacement Algorithm

- The basic working set algorithm is cumbersome, since the entire page table has to be scanned at each fault until a suitable candidate is located. An improved algorithm, that is based on clock algorithm but also uses the working set information is called WS Clock.
- As with clock algorithm, at each page fault the page pointed to by the hand is examined first. If R bit is set to 1, the page has been used during the current tick so it is not an ideal candidate to remove. The R bit is then set to 0, the hand advanced to next page, and algorithm repeated for that page.And if R=0 it is the candidate for removal.

Thrashing

- If the process does not have sufficient number of frames for the pages in active use, it will quickly page fault. At this point, it must replace some page. However, since all its pages are in active use, it must replace a page that will be needed again right away. Consequently, it quickly faults again, and again, and again. The process continues to fault, replacing pages for which it then faults and brings back in right away.
- This high paging activity is called **thrashing. A process is thrashing if it is** spending more time paging than executing.

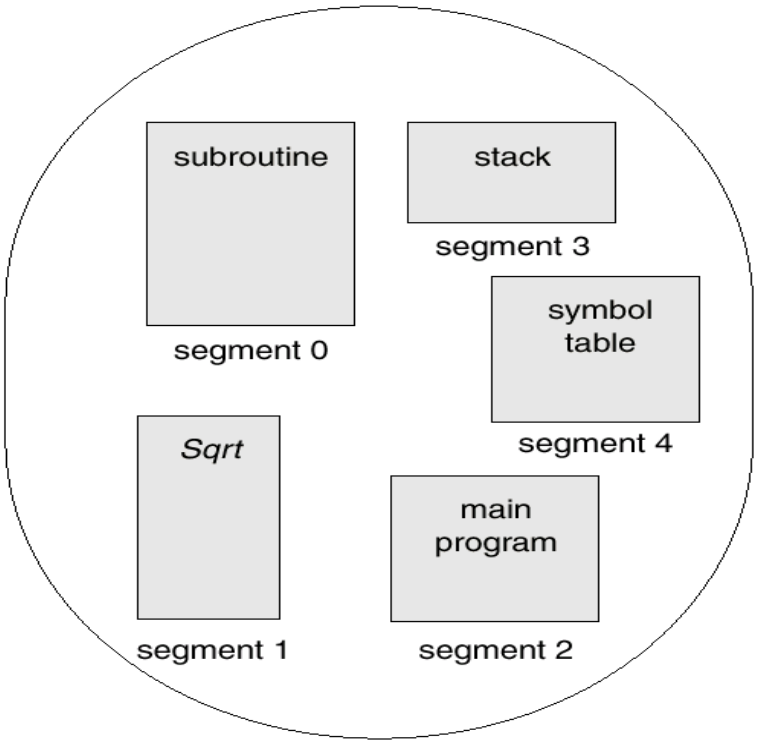
Segmentation

- **Segmentation is a** 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, local variables, global variables, common block,stack, symbol table, arrays

Segmentation Architecture

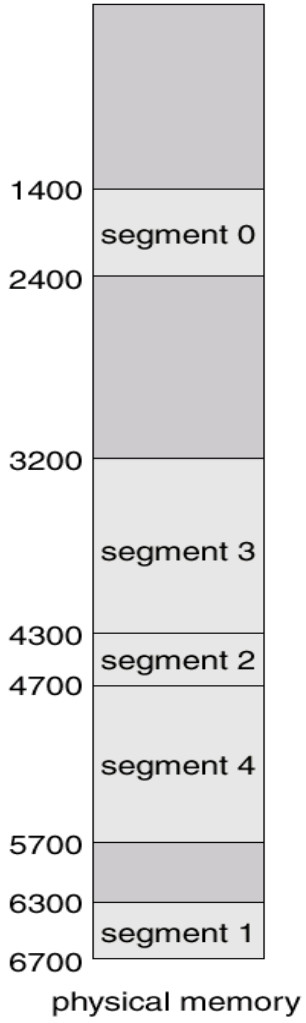
- Logical address consists of a two tuple:
  <segment-number, offset>,
- *Segment table – maps two-dimensional physical* addresses; each table entry has:
  ✦ base – contains the starting physical address where the segments reside in memory.
  ✦ *limit – specifies the length of the segment.*
- *Segment-table base register (STBR) points to the* segment table's location in memory.
- *Segment-table length register (STLR) indicates number of* segments used by a program;
  segment number *s is legal if s < STLR.*

Example of Segmentation



	limit	base
0	1000	1400
1	400	6300
2	400	4300
3	1100	3200
4	1000	4700

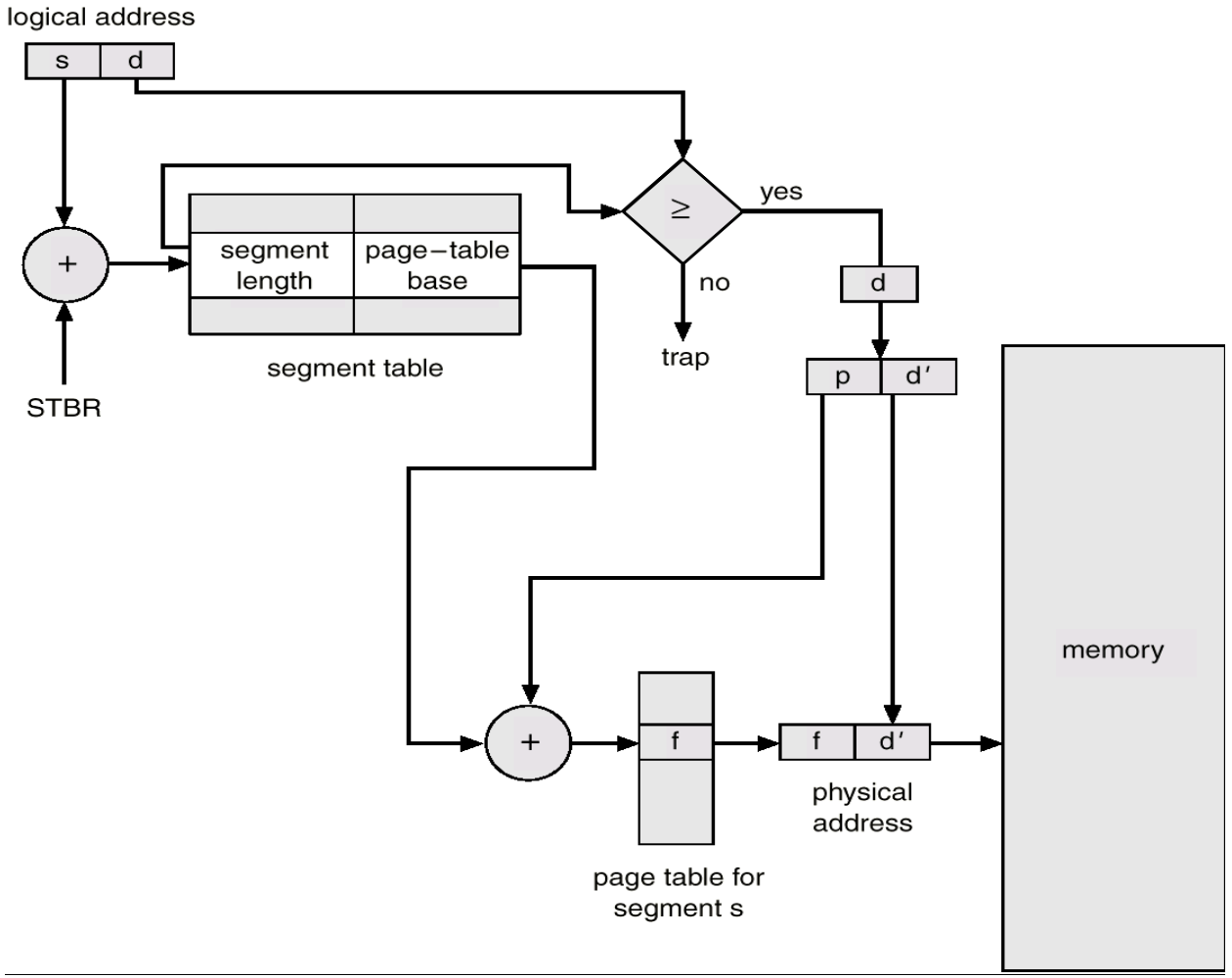
segment table



Segmentation with Paging – MULTICS

- The MULTICS system solved problems of external fragmentation and lengthy search times by paging the segments.
- Solution differs from pure segmentation in that the segment-table entry contains not the base address of the segment, but rather the base address of a *page table for* this segment.

MULTICS Address Translation Scheme



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.