Memory Management

- Memory Management
    - Memory Management strategies: Swapping, Contiguous Memory Allocation, Paging, Segmentation.
- Virtual Memory Management:
    - Demand Paging, Page Replacement, Allocation of Frames, Thrashing

Background

- Memory is central to the operation of a modern computer system.
- Memory consists of a large array of bytes, each with its own address.
- The CPU fetches instructions from memory according to the value of the program counter.

Basic Hardware

- Main memory and the registers built into the processor itself are the only general-purpose storage that the CPU can access directly.
- There are machine instructions that take memory addresses as arguments, but none that take disk addresses.
- Therefore, any instructions in execution, and any data being used by the instructions, must be in one of these direct-access storage devices.
- If the data are not in memory, they must be moved there before the CPU can operate on them.

Cache Memory

- Fast memory between the CPU and main memory, typically on the CPU chip for fast access---cache
- Information is normally kept in some storage system (such as main memory).
- As it is used, it is copied into a faster storage system—the cache—on a temporary basis.
- When we need a particular piece of information, we first check whether it is in the cache.
- If it is, we use the information directly from the cache. If it is not, we use the information from the source, putting a copy in the cache under the assumption that we will need it again soon.

- Each process has a separate memory space.
- Separate per-process memory space protects the processes from each other and is fundamental to having multiple processes loaded in memory for concurrent execution.
- To separate memory spaces, we need the ability to determine the range of legal addresses that the process may access and to ensure that the process can access only these legal addresses.
- We can provide this protection by using two registers, usually a base and a limit

- The **base register holds the smallest legal physical memory address; the limit register specifies the size of the range.**

- **For example, if the base register holds** 300040 and the limit register is 120900, then the program can legally access all addresses from 300040 through 420939 (inclusive).

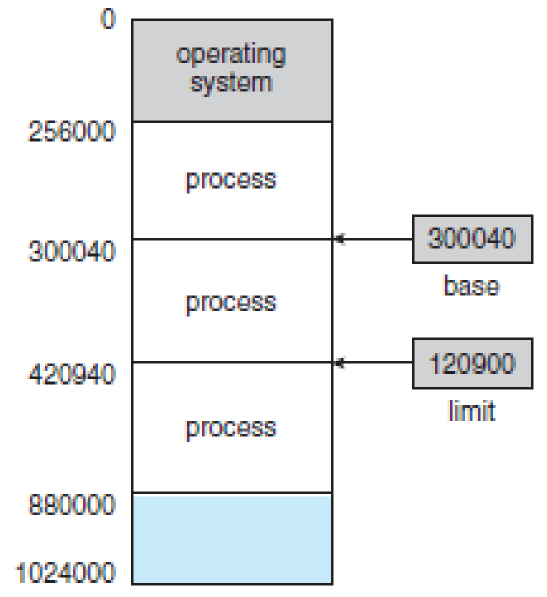


Figure 8.1 A base and a limit register define a logical address space.

- Protection of memory space is accomplished by having the CPU hardware compare every address generated in user mode with the registers.
- Any attempt by a program executing in user mode to access operating-system memory or other users' memory results in a trap to the operating system, which treats the attempt as a fatal error.
- This scheme prevents a user program from (accidentally or deliberately) modifying the code or data structures of either the operating system or other users.

- The base and limit registers can be loaded only by the operating system, which uses a special privileged instruction.
- Since privileged instructions can be executed only in kernel mode, and since only the operating system executes in kernel mode, only the operating system can load the base and limit registers.
- This scheme allows the operating system to change the value of the registers but prevents user programs from changing the registers' contents.
- The operating system, executing in kernel mode, is given unrestricted access to both operating-system memory and users' memory.

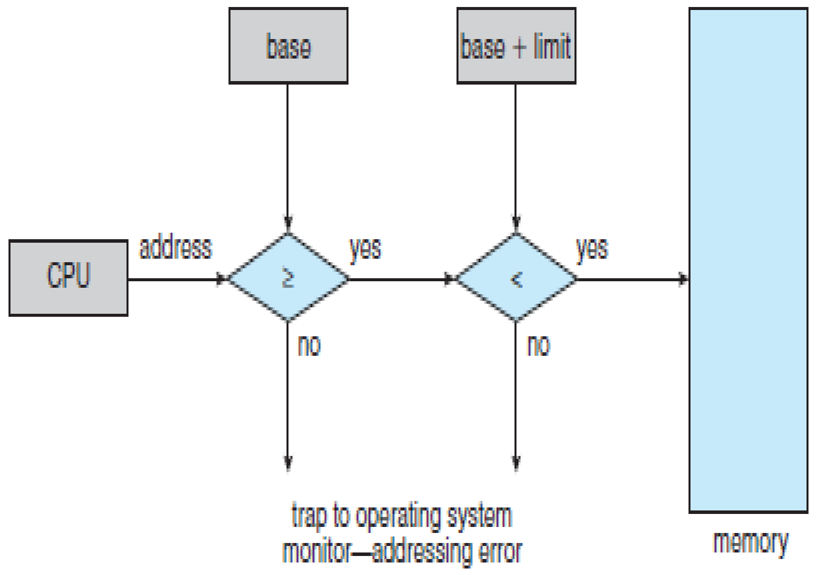


Figure 8.2 Hardware address protection with base and limit registers.

Address Binding

- A program resides on a disk as a binary executable file.
- To be executed, the program must be brought into memory and placed within a process.
- Addresses may be represented in different ways during these steps.
- Addresses in the source program are generally symbolic (such as the variable count).
- A compiler typically **binds these symbolic addresses to relocatable addresses (such as** "14 bytes from the beginning of this module").
- The linkage editor or loader in turn binds the relocatable addresses to absolute addresses (such as 74014).
- Each binding is a mapping from one address space to another.

- The binding of instructions and data to memory addresses can be done at any step along the way:
- **Compile time. If you know at compile time where the process will reside** in memory, then **absolute code can be generated.**
- **Load time. If it is not known at compile time where the process will reside** in memory, then the compiler must generate **relocatable code.**
- **Execution time. If the process can be moved during its execution from** one memory segment to another, then binding must be delayed until run time.

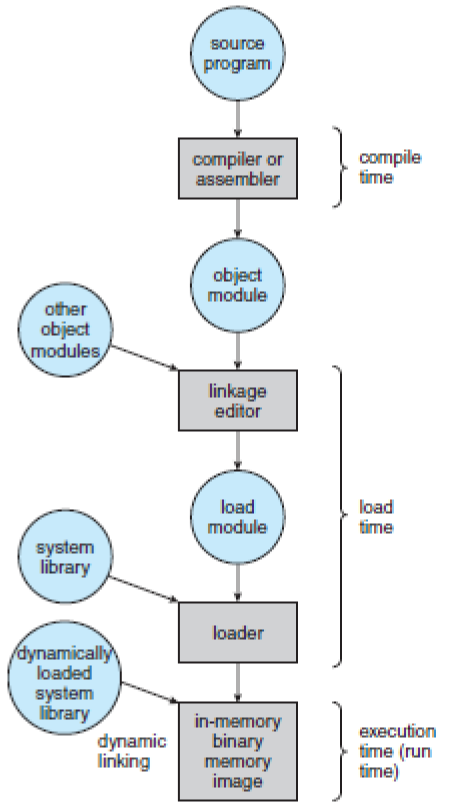


Figure 8.3 Multistep processing of a user program.

Logical Versus Physical Address Space

- 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.**

- The set of all logical addresses generated by a program is a **logical address space. The set** of all physical addresses corresponding to these logical addresses is a **physical address space.**

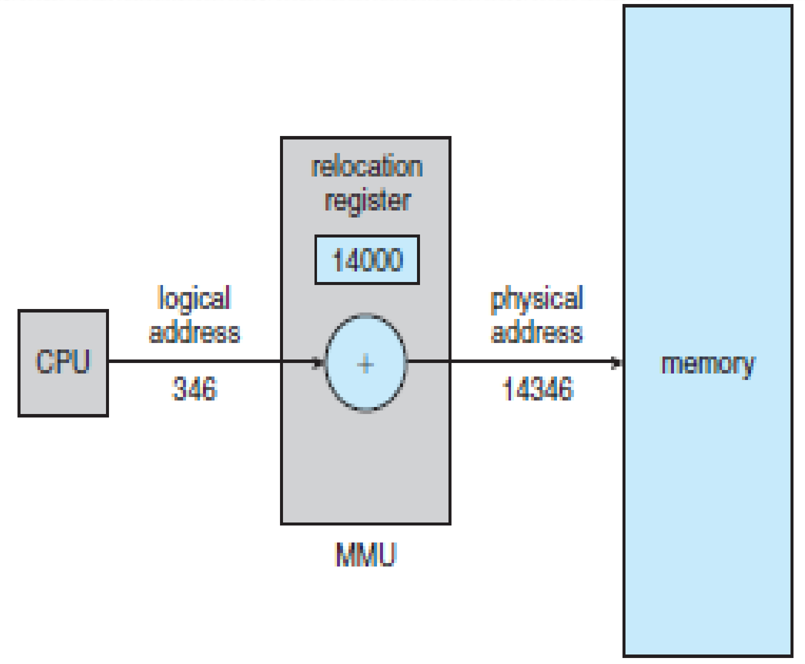


Figure 8.4 Dynamic relocation using a relocation register.

- The user program never sees the real physical addresses.
- The user program deals with logical addresses.
- The memory-mapping hardware converts logical addresses into physical addresses.

Swapping

- A process must be in memory to be executed.
- A process, however, can be **swapped temporarily out of memory to a backing store and then brought back** into memory for continued execution.
- Swapping makes it possible for the total physical address space of all processes to exceed the real physical memory of the system, thus increasing the degree of multiprogramming in a system.

Standard Swapping

- Standard swapping involves moving processes between main memory and a backing store.
- The backing store is commonly a fast disk. It must be large enough to accommodate copies of all memory images for all users, and it must provide direct access to these memory images.
- The system maintains a **ready queue** consisting of all processes whose memory images are on the backing store or in memory and are ready to run.

- Whenever the CPU scheduler decides to execute a process, it calls the dispatcher.
- The dispatcher checks to see whether the next process in the queue is in memory.
- If it is not, and if there is no free memory region, the dispatcher swaps out a process currently in memory and swaps in the desired process.
- It then reloads registers and transfers control to the selected process.
- The context-switch time in such a swapping system is fairly high.

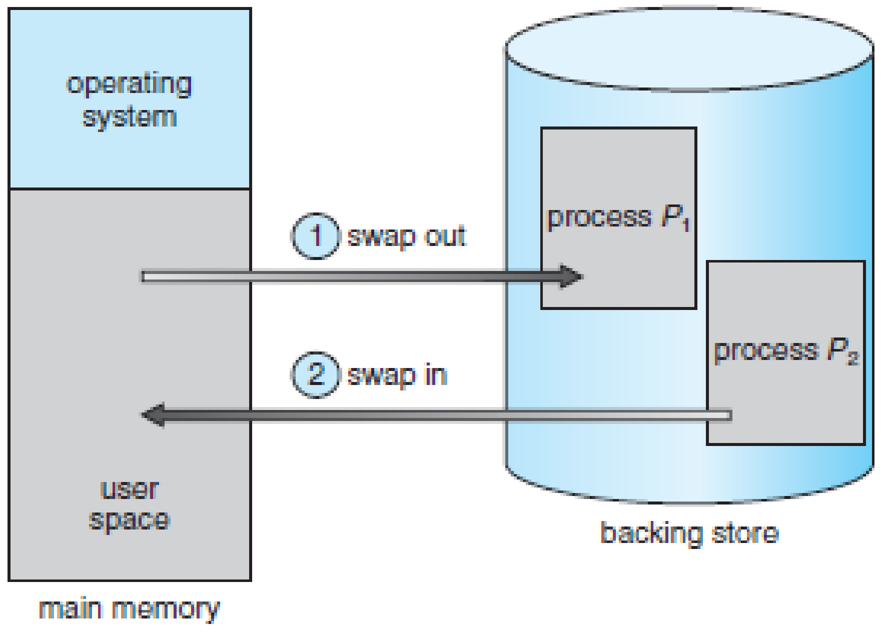


Figure 8.5 Swapping of two processes using a disk as a backing store.

Contiguous Memory Allocation

- The main memory must accommodate both the operating system and the various user processes.
- We therefore need to allocate main memory in the most efficient way possible.
- The memory is usually divided into two partitions: one for the resident operating system and one for the user processes.

- We usually want several user processes to reside in memory at the same time.
- We therefore need to consider how to allocate available memory to the processes that are in the input queue waiting to be brought into memory.
- In **contiguous memory allocation, each process is contained in a single section of** memory that is contiguous to the section containing the next process.

Memory Allocation

- One of the simplest methods for allocating memory is to divide memory into several **fixed-sized partitions.**
- **Each partition may contain exactly one process. Thus, the degree** of multiprogramming is bound by the number of partitions.
- In this **multiple partition method, when a partition is free, a process is selected from the input** queue and is loaded into the free partition.
- When the process terminates, the partition becomes available for another process.

- In the **variable-partition scheme, the operating system keeps a table** indicating which parts of memory are available and which are occupied.
- Initially, all memory is available for user processes and is considered one large block of available memory, a **hole.**
- **Eventually, as you will see, memory** contains a set of holes of various sizes.

- In general, as mentioned, the memory blocks available comprise a ***set of*** holes of various sizes scattered throughout memory.
- When a process arrives and needs memory, the system searches the set for a hole that is large enough for this process.
- If the hole is too large, it is split into two parts.
- One part is allocated to the arriving process; the other is returned to the set of holes.

- When a process terminates, it releases its block of memory, which is then placed back in the set of holes.
- If the new hole is adjacent to other holes, these adjacent holes are merged to form one larger hole.
- At this point, the system may need to check whether there are processes waiting for memory and whether this newly freed and recombined memory could satisfy the demands of any of these waiting processes.

- This procedure is a particular instance of the general **dynamic storage allocation problem, which concerns how to satisfy a request of size *n* *from a*** list of free holes.

- There are many solutions to this problem. The **first-fit, best-fit,** and **worst-fit strategies are the ones most commonly used to select a free hole** from the set of available holes.

- **First fit. Allocate the first hole that is big enough. Searching can start either** at the beginning of the set of holes or at the location where the previous first-fit search ended. We can stop searching as soon as we find a free hole that is large enough.
- **Best fit. Allocate the smallest hole that is big enough. We must search the** entire list, unless the list is ordered by size. This strategy produces the smallest leftover hole.
- **Worst fit. Allocate the largest hole. Again, we must search the entire list,** unless it is sorted by size. This strategy produces the largest leftover hole, which may be more useful than the smaller leftover hole from a best-fit approach.

Numericals

2Kb P3	2Kb	3Kb P2	3Kb

Variable size partition numerical

- P1=300KB, P2=25KB, P3=125KB, P4=50KB

Initial	**50KB**	**150KB**		**300KB**	**350KB**			**600KB**
1st fit		P2	P3		P1	P4		
Best fit		P3	25KB		P1	P2	25KB	
Worst fit		P2	P3		P1	P4		

FIXED SIZE

- P1=357KB, P2=210KB, P3=468KB, P4=491KB

INITIAL	**200KB**	**400KB**	**600KB**	**500KB**	**300KB**	**250KB**
1ST FIT		357 **43**	210 **390**	468 **32**		
INTERNAL FRAG= 43+390+32				EXTERNAL FRAG=491		
BEST FIT		357 **43**	491 **109**	468 **32**		210 **40**
INTERNAL FRAG= 43+109+32+40				EXTERNAL FRAG=0		
WORST FIT			357 **243**	210 **290**		
INTERNAL FRAG= 243+290				EXTERNAL FRAG=468+491		

- Simulations have shown that both first fit and best fit are better than worst fit in terms of decreasing time and storage utilization.
- Neither first fit nor best fit is clearly better than the other in terms of storage utilization, but first fit is generally faster.
- Best fit performs best in fixed size, worst in variable size.
- Worst fit performs best in variable size , worst in fixed size.

Fragmentation

- Both the first-fit and best-fit strategies for memory allocation suffer from external fragmentation.
- **External fragmentation exists when there is enough total memory space to satisfy a request but the available spaces are not contiguous: storage is fragmented into a large number of small holes. This fragmentation problem can be severe.**
- In the worst case, we could have a block of free (or wasted) memory between every two processes.
- If all these small pieces of memory were in one big free block instead, we might be able to run several more processes.

- Whether we are using the first-fit or best-fit strategy can affect the amount of fragmentation.
- First fit is better for some systems, whereas best fit is better for others.
- No matter which algorithm is used, however, external fragmentation will be a problem.
- Depending on the total amount of memory storage and the average process size, external fragmentation may be a minor or a major problem.

- Memory fragmentation can be internal as well as external.
- Consider a multiple-partition allocation scheme with a hole of 18,464 bytes.
- Suppose that the next process requests 18,462 bytes.
- If we allocate exactly the requested block, we are left with a hole of 2 bytes.
- The overhead to keep track of this hole will be substantially larger than the hole itself.

- The general approach to avoiding this problem is to break the physical memory into fixed-sized blocks and allocate memory in units based on block size.
- With this approach, the memory allocated to a process may be slightly larger than the requested memory.
- **The difference between these two numbers is internal fragmentation—unused memory that is internal to a partition.**

- One solution to the problem of external fragmentation is **compaction.** The goal is to shuffle the memory contents so as to place all free memory together in one large block.
- Compaction is not always possible, however.
- Another possible solution to the external-fragmentation problem is to permit the logical address space of the processes to be noncontiguous, thus allowing a process to be allocated physical memory wherever such memory is available.
- Two complementary techniques achieve this solution: **segmentation and paging**. These techniques can also be combined.
- Fragmentation is a general problem in computing that can occur wherever we must manage blocks of data.

Segmentation

- **Segmentation Hardware**
- A logical address consists of two parts: a segment number, *s, and an offset into that segment, d.*
- The segment number is used as an index to the segment table.
- The segment table is thus essentially an array of base–limit register pairs.
- Each entry in the segment table has a **segment base and a segment limit.**
- **The segment base** contains the starting physical address where the segment resides in memory, and the segment limit specifies the length of the segment.

- The offset *d of* the logical address must be between 0 and the segment limit.
- If it is not, we trap to the operating system (logical addressing attempt beyond end of segment).
- When an offset is legal, it is added to the segment base to produce the address in physical memory of the desired byte.

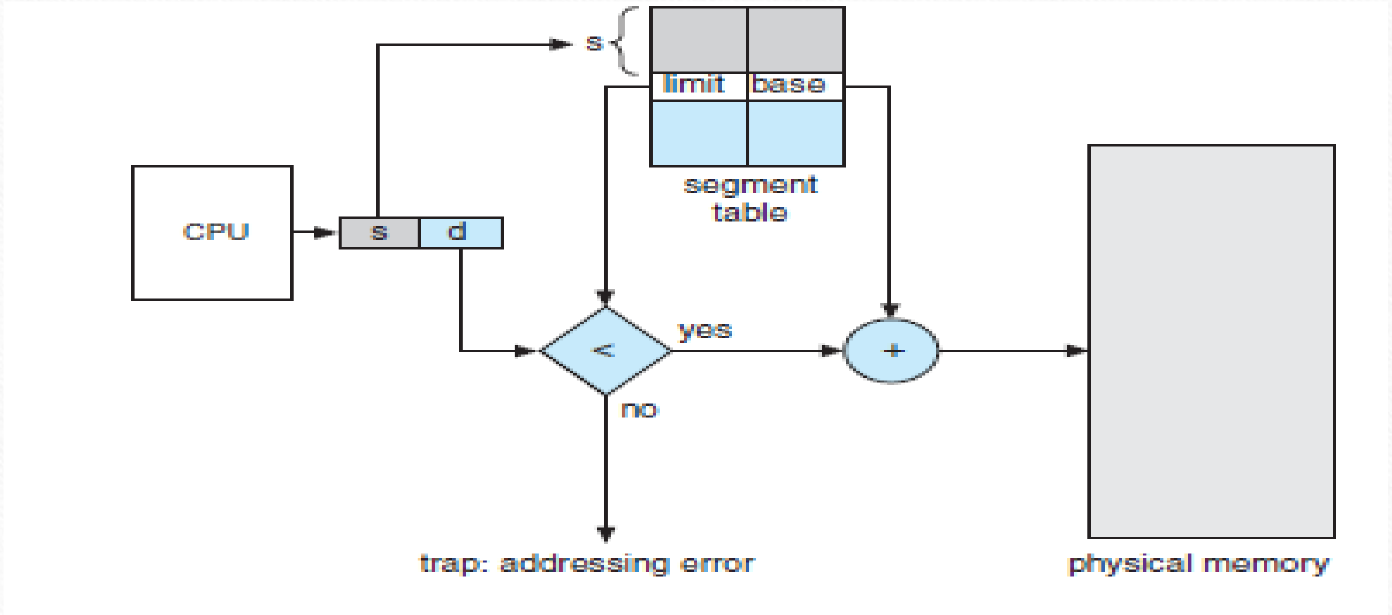
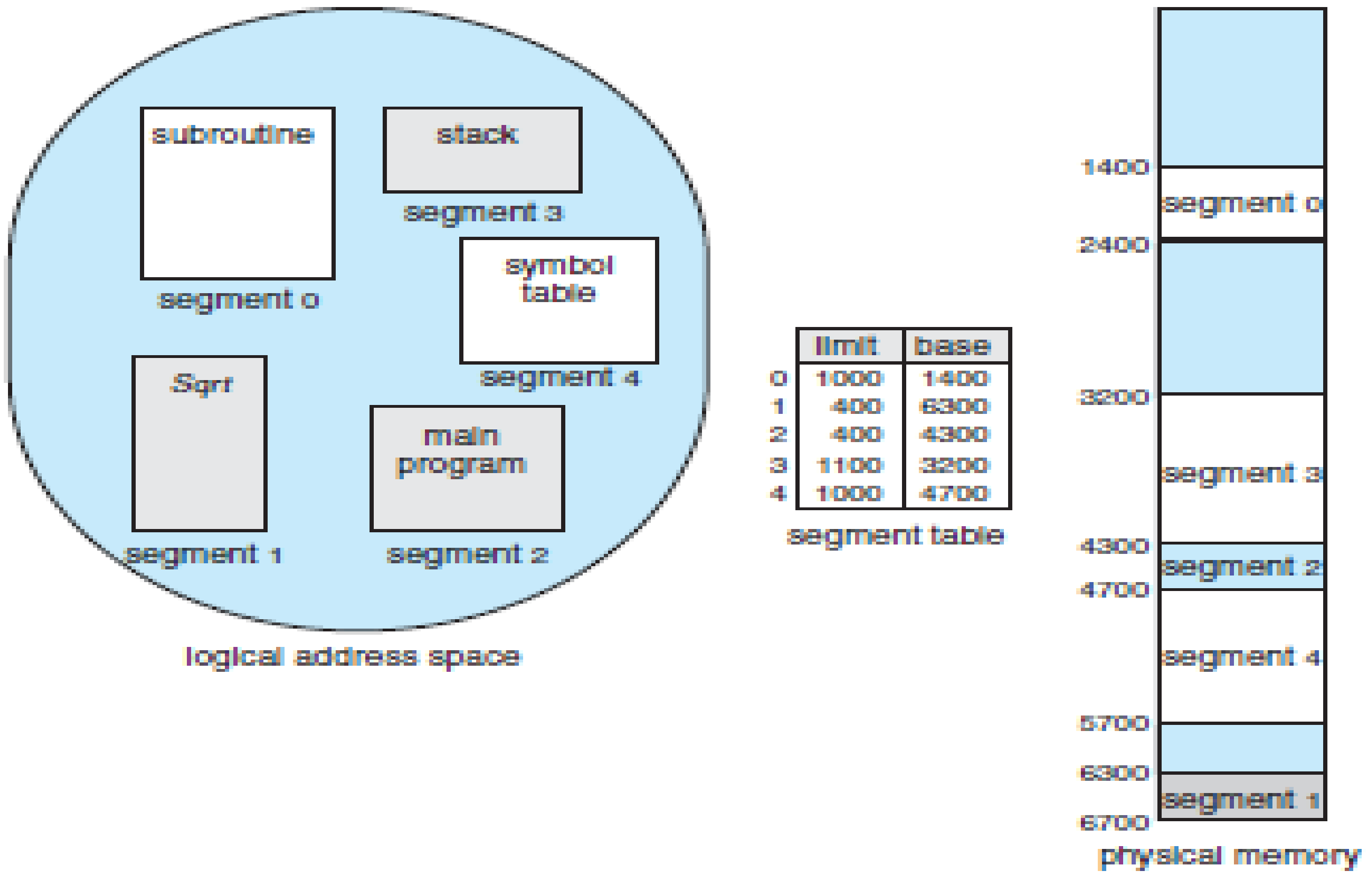


Figure 8.8 Segmentation hardware.



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

Figure 8.9 Example of segmentation.

Segments

- When writing a program, a programmer thinks of it as a main program with a set of methods, procedures, or functions.
- It may also include various data structures: objects, arrays, stacks, variables, and so on. Each of these modules or data elements is referred to by name.
- The programmer talks about “the stack,” “the math library,” and “the main program” without caring what addresses in memory these elements occupy.
- She is not concerned with whether the stack is stored before or after the Sqrt() function.
- Segments vary in length, and the length of each is intrinsically defined by its purpose in the program.
- Elements within a segment are identified by their offset from the beginning of the segment: the first statement of the program, the seventh stack frame entry in the stack, the fifth instruction of the Sqrt(), and so on.
- A logical address space is a collection of segments. Each segment has a name and a length.

Example of Segments

- Normally, when a program is compiled, the compiler automatically constructs segments reflecting the input program.
- A C compiler might create separate segments for the following:
  - The code
  - Global variables
  - The heap, from which memory is allocated
  - The stacks used by each thread
  - The standard C library
- Libraries that are linked in during compile time might be assigned separate segments. The loader would take all these segments and assign them segment numbers.

Paging

- Segmentation permits the physical address space of a process to be noncontiguous.
- **Paging is another memory-management scheme that offers this** advantage.
- However, paging avoids external fragmentation and the need for compaction, whereas segmentation does not.

- It also solves the considerable problem of fitting memory chunks of varying sizes onto the backing store.
- When code fragments or data residing in main memory need to be swapped out, space must be found on the backing store.
- The backing store has the same fragmentation problems discussed in connection with main memory, but access is much slower, so compaction is impossible.

Basic Method of Paging

- The basic method for implementing paging involves breaking physical memory into fixed-sized blocks called **frames and breaking logical memory into** blocks of the same size called **pages.**
- **When a process is to be executed, its** pages are loaded into any available memory frames from their source (a file system or the backing store).
- The backing store is divided into fixed-sized blocks that are the same size as the memory frames or clusters of multiple frames.

- Every address generated by the CPU is divided into two parts: a **page number (p) and a page offset (d).**
- **The page number is used as an index into a page table.**
- **The page table** contains the base address of each page in physical memory.
- This base address is combined with the page offset to define the physical memory address that is sent to the memory unit.

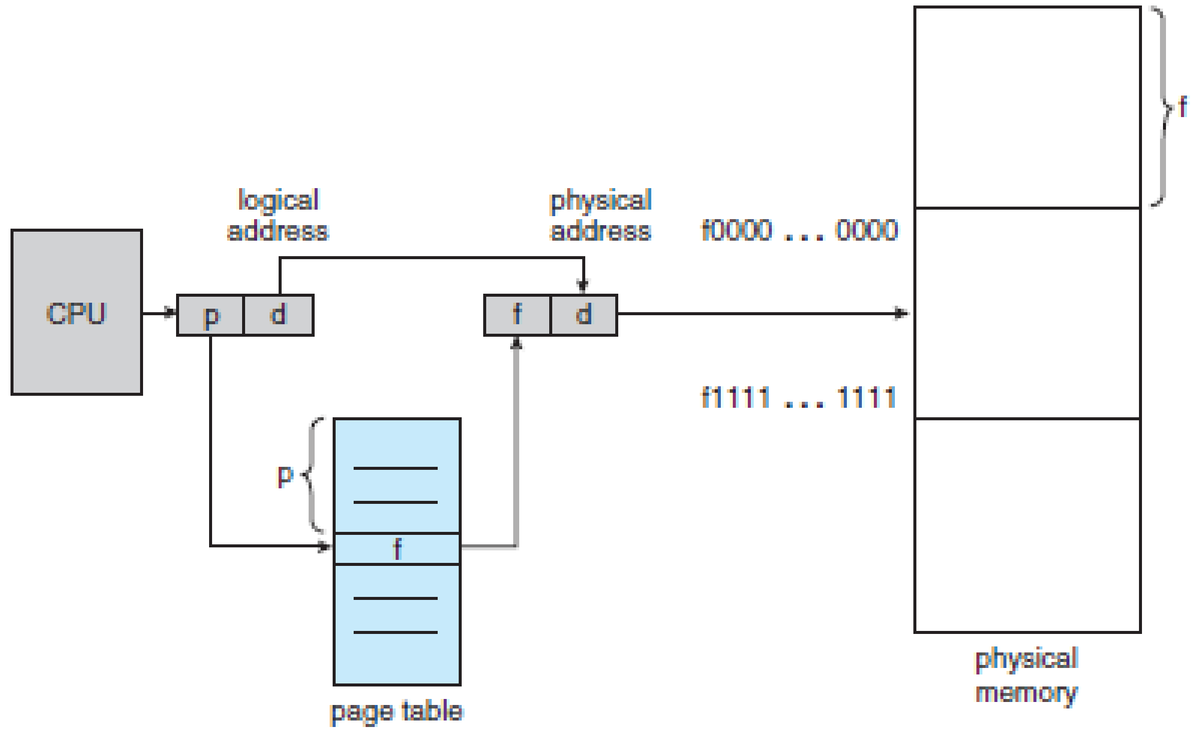


Figure 8.10 Paging hardware.

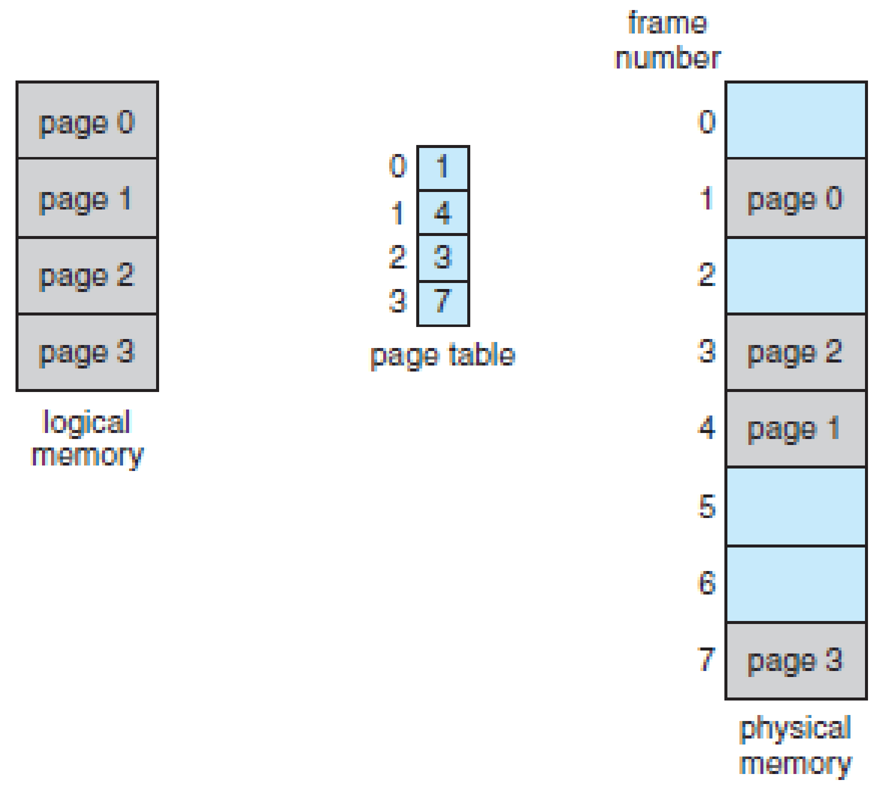
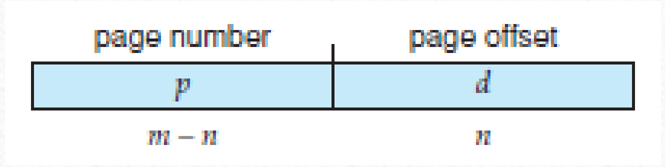


Figure 8.11 Paging model of logical and physical memory.



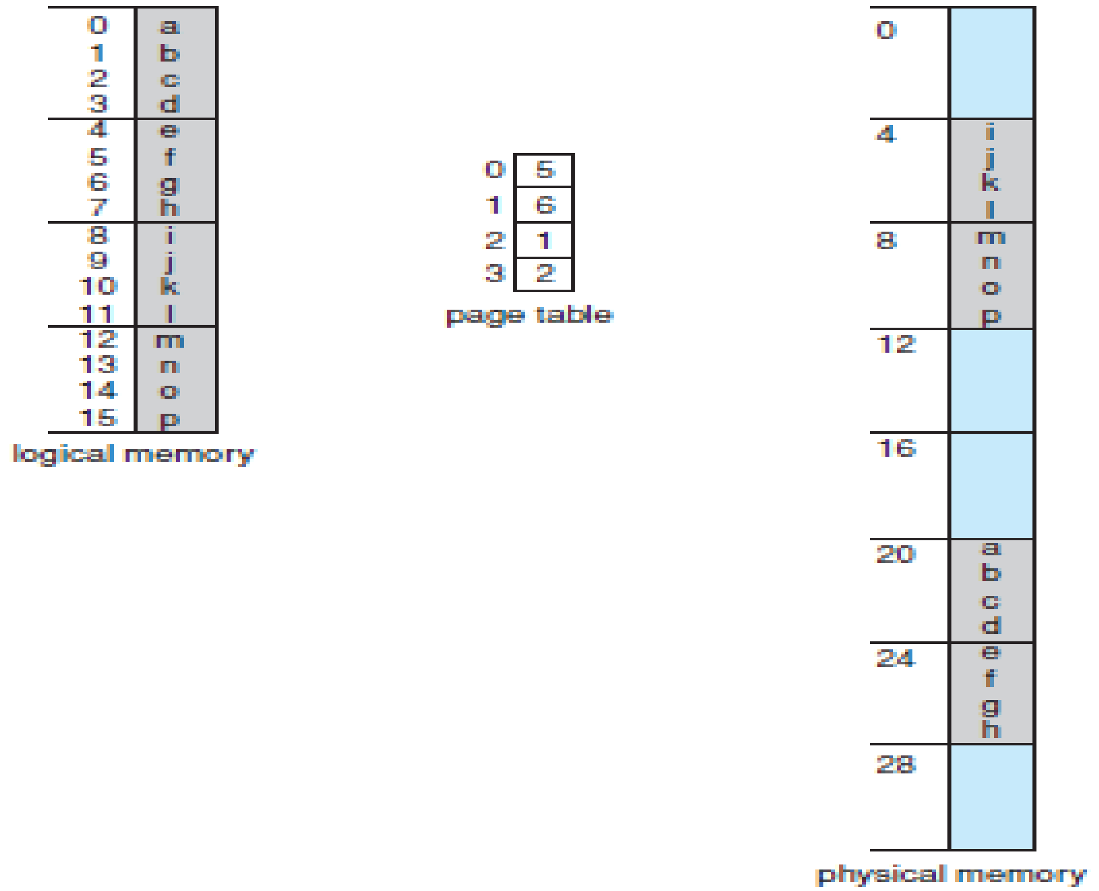


Figure 8.12 Paging example for a 32-byte memory with 4-byte pages.

- Paging itself is a form of dynamic relocation.
- Every logical address is bound by the paging hardware to some physical address.
- Using paging is similar to using a table of base (or relocation) registers, one for each frame of memory.
- When we use a paging scheme, we have no external fragmentation: any free frame can be allocated to a process that needs it. However, we may have some internal fragmentation.

Page Size

- Generally, page sizes have grown over time as processes, data sets, and main memory have become larger.
- Today, pages typically are between 4 KB and 8 KB in size, and some systems support even larger page sizes.
- Some CPUs and kernels even support multiple page sizes.
- For instance, Solaris uses page sizes of 8 KB and 4 MB, depending on the data stored by the pages.
- Researchers are now developing support for variable on-the-fly page size.
- Frequently, on a 32-bit CPU, each page-table entry is 4 bytes long, but that size can vary as well.

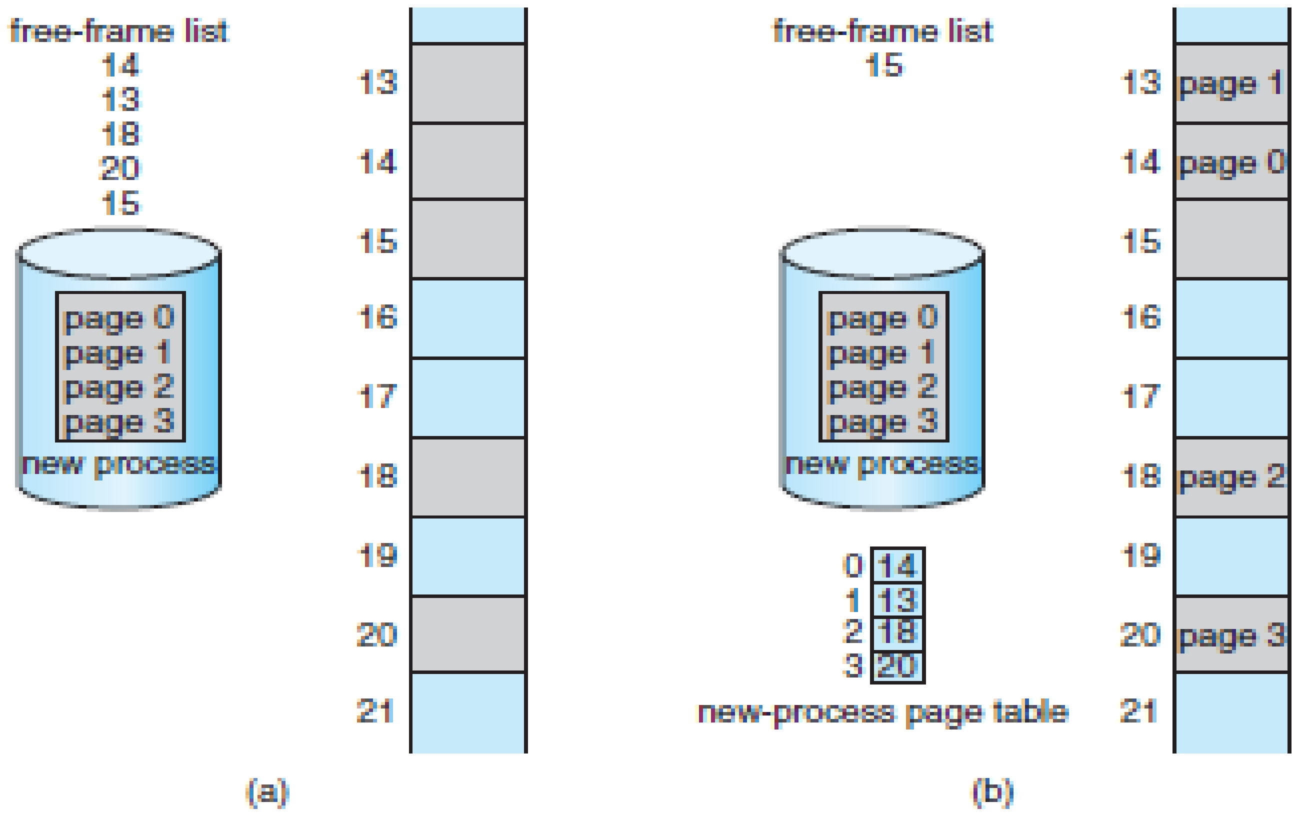
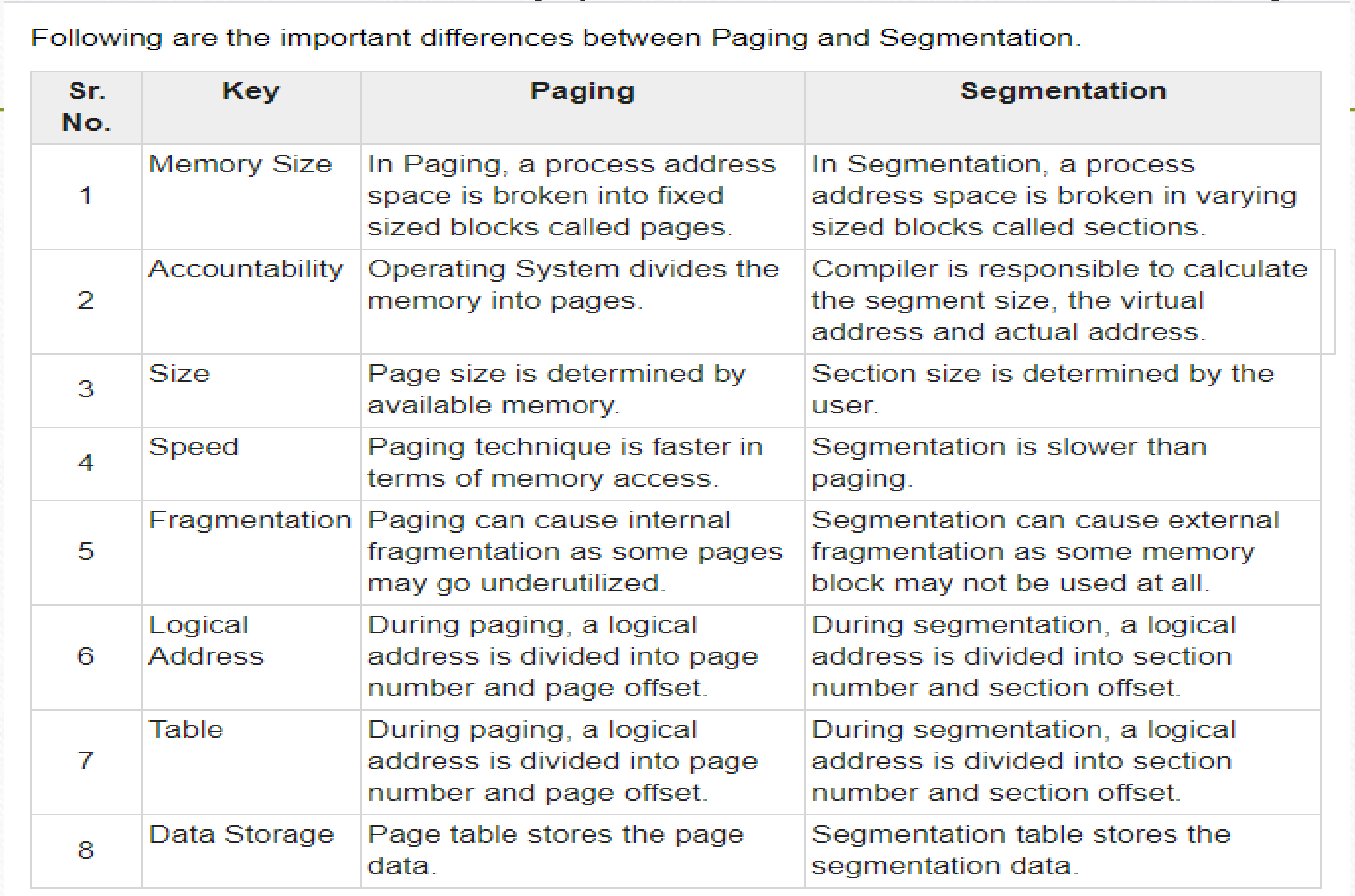


Figure 8.13 Free frames (a) before allocation and (b) after allocation.

- An important aspect of paging is the clear separation between the programmer's view of memory and the actual physical memory.
- The programmer views memory as one single space, containing only this one program. In fact, the user program is scattered throughout physical memory, which also holds other programs.
- The difference between the programmer's view of memory and the actual physical memory is reconciled by the address-translation hardware.
- The logical addresses are translated into physical addresses. This mapping is hidden from the programmer and is controlled by the operating system. Notice that the user process by definition is unable to access memory it does not own.

- Since the operating system is managing physical memory, it must be aware of the allocation details of physical memory—which frames are allocated, which frames are available, how many total frames there are, and so on.
- This information is generally kept in a data structure called a **frame table.**
- **The frame** table has one entry for each physical page frame, indicating whether the latter is free or allocated and, if it is allocated, to which page of which process or processes.

Diff between segmentation and paging



Following are the important differences between Paging and Segmentation.

Sr. No.	Key	Paging	Segmentation
1	Memory Size	In Paging, a process address space is broken into fixed sized blocks called pages.	In Segmentation, a process address space is broken in varying sized blocks called sections.
2	Accountability	Operating System divides the memory into pages.	Compiler is responsible to calculate the segment size, the virtual address and actual address.
3	Size	Page size is determined by available memory.	Section size is determined by the user.
4	Speed	Paging technique is faster in terms of memory access.	Segmentation is slower than paging.
5	Fragmentation	Paging can cause internal fragmentation as some pages may go underutilized.	Segmentation can cause external fragmentation as some memory block may not be used at all.
6	Logical Address	During paging, a logical address is divided into page number and page offset.	During segmentation, a logical address is divided into section number and section offset.
7	Table	During paging, a logical address is divided into page number and page offset.	During segmentation, a logical address is divided into section number and section offset.
8	Data Storage	Page table stores the page data.	Segmentation table stores the segmentation data.

S.NO	PAGING	SEGMENTATION
1.	In paging, program is divided into fixed or mounted size pages.	In segmentation, program is divided into variable size sections.
2.	For paging operating system is accountable.	For segmentation compiler is accountable.
3.	Page size is determined by hardware.	Here, the section size is given by the user.
4.	It is faster in the comparison of segmentation.	Segmentation is slow.
5.	Paging could result in internal fragmentation.	Segmentation could result in external fragmentation.
6.	In paging, logical address is split into page number and page offset.	Here, logical address is split into section number and section offset.
7.	Paging comprises a page table which encloses the base address of every page.	While segmentation also comprises the segment table which encloses segment number and segment offset.
8.	Page table is employed to keep up the page data.	Section Table maintains the section data.
9.	In paging, operating system must maintain a free frame list.	In segmentation, operating system maintain a list of holes in main memory.
10.	Paging is invisible to the user.	Segmentation is visible to the user.

Virtual Memory

- Virtual memory is a technique that allows the execution of processes that are not completely in memory.
- One major advantage of this scheme is that programs can be larger than physical memory.
- virtual memory abstracts main memory into an extremely large, uniform array of storage, separating logical memory as viewed by the user from physical memory.
- This technique frees programmers from the concerns of memory-storage limitations.
- Virtual memory also allows processes to share files easily and to implement shared memory.
- In addition, it provides an efficient mechanism for process creation.
- Virtual memory is not easy to implement, however, and may substantially decrease performance if it is used carelessly.

- The requirement that instructions must be in physical memory to be executed seems both necessary and reasonable; but it is also unfortunate, since it limits the size of a program to the size of physical memory.
- In fact, an examination of real programs shows us that, in many cases, the entire program is not needed.
- Even in those cases where the entire program is needed, it may not all be needed at the same time.
- Thus, running a program that is not entirely in memory would benefit both the system and the user.

implementing virtual memory through demand paging

- Consider how an executable program might be loaded from disk into memory.
- One option is to load the entire program in physical memory at program execution time.
- However, a problem with this approach is that we may not initially ***need the entire program in memory.***
- An alternative strategy is to load pages only as they are needed.
- This technique is known as **demand paging and is commonly used in virtual memory systems.**
- With demand-paged virtual memory, pages are loaded only when they are demanded during program execution.
- Pages that are never accessed are thus never loaded into physical memory.

- A demand-paging system is similar to a paging system with swapping where processes reside in secondary memory (usually a disk).
- When we want to execute a process, we swap it into memory.
- Rather than swapping the entire process into memory, though, we use a **lazy swapper.**
- A lazy swapper never swaps a page into memory unless that page will be needed.
- In the context of a demand-paging system, use of the term "swapper" is technically incorrect.
- A swapper manipulates entire processes, whereas a **pager is concerned with the individual pages of a process.**
- **We thus use "pager,"** rather than "swapper," in connection with demand paging.

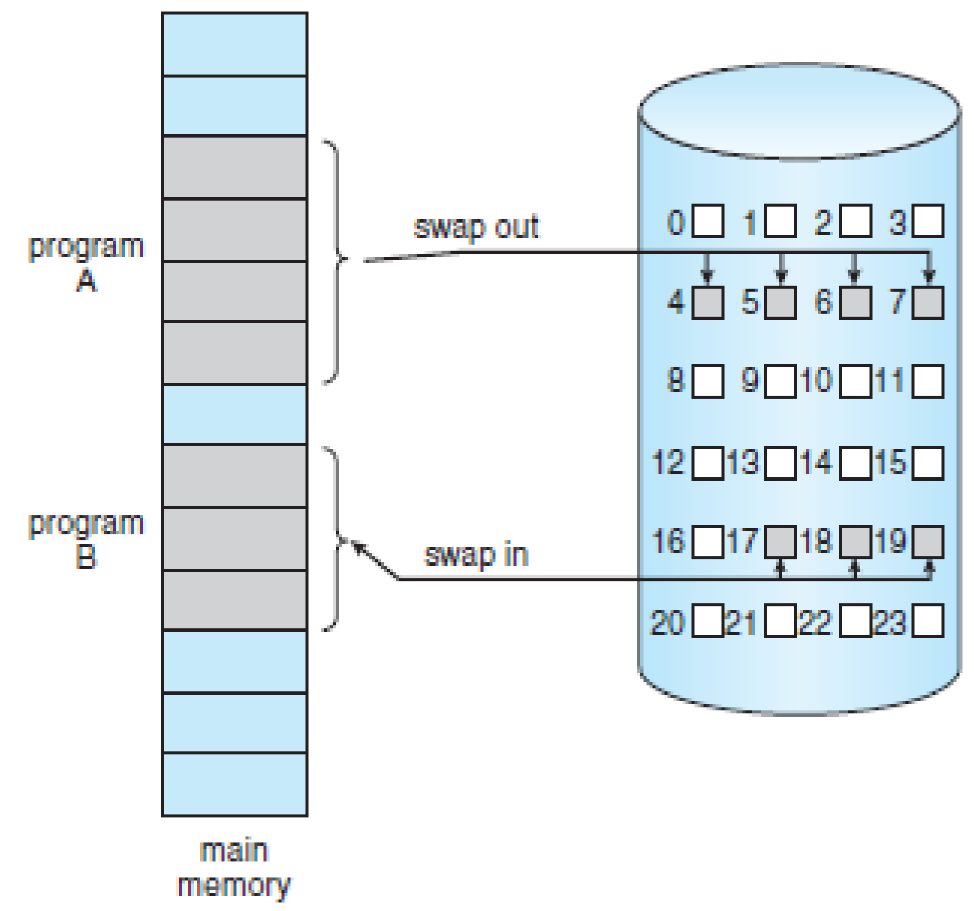


Figure 9.4 Transfer of a paged memory to contiguous disk space.

Mapping of logical mem to phy mem using demand paging

- we need some form of hardware support to distinguish between the pages that are in memory and the pages that are on the disk.
- The valid–invalid bit can be used for this purpose.
- When this bit is set to "valid," the associated page is both legal and in memory. If the bit is set to "invalid," the page either is not valid (that is, not in the logical address space of the process) or is valid but is currently on the disk.
- The page-table entry for a page that is brought into memory is set as usual, but the page-table entry for a page that is not currently in memory is either simply marked invalid or contains the address of the page on disk.

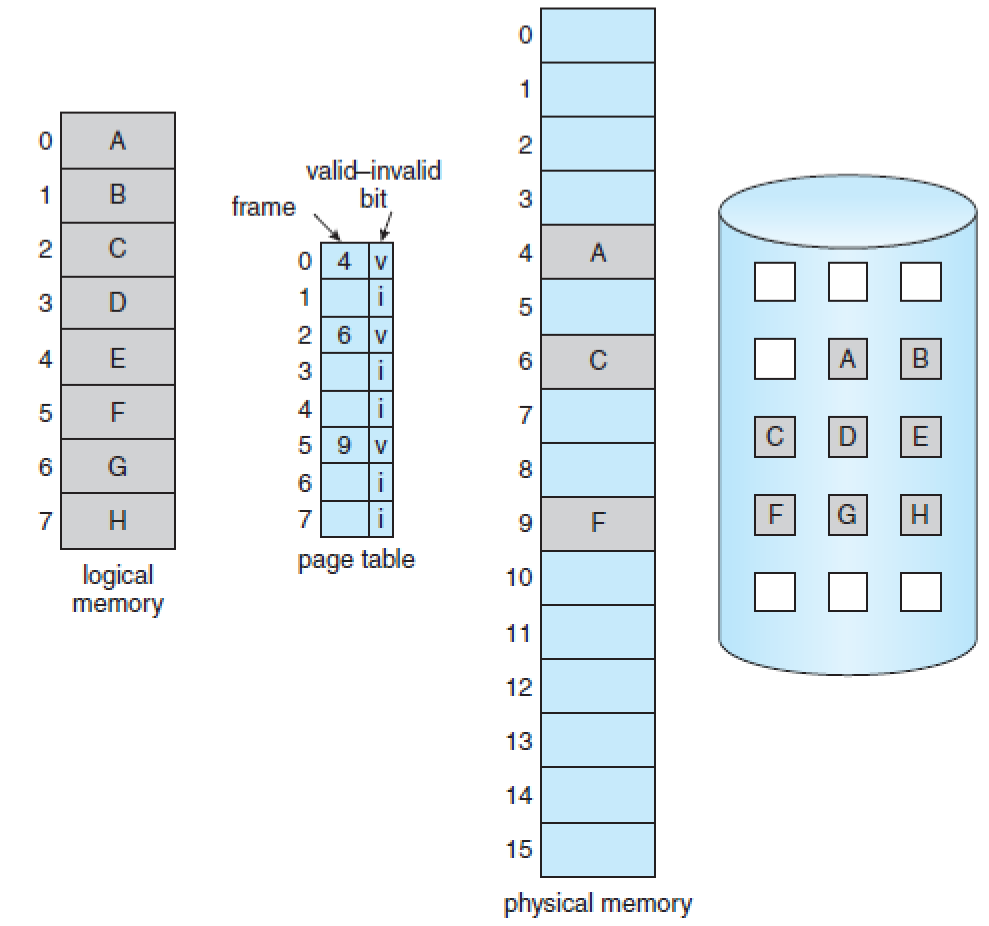


Figure 9.5 Page table when some pages are not in main memory.

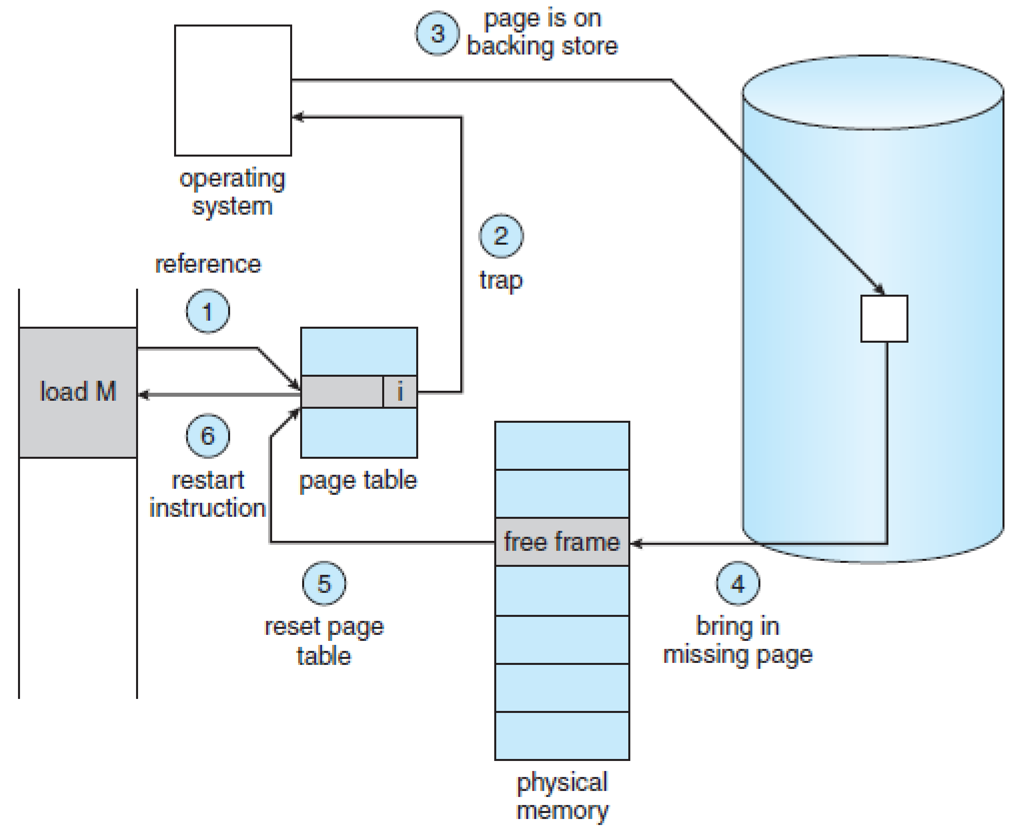


Figure 9.6 Steps in handling a page fault.

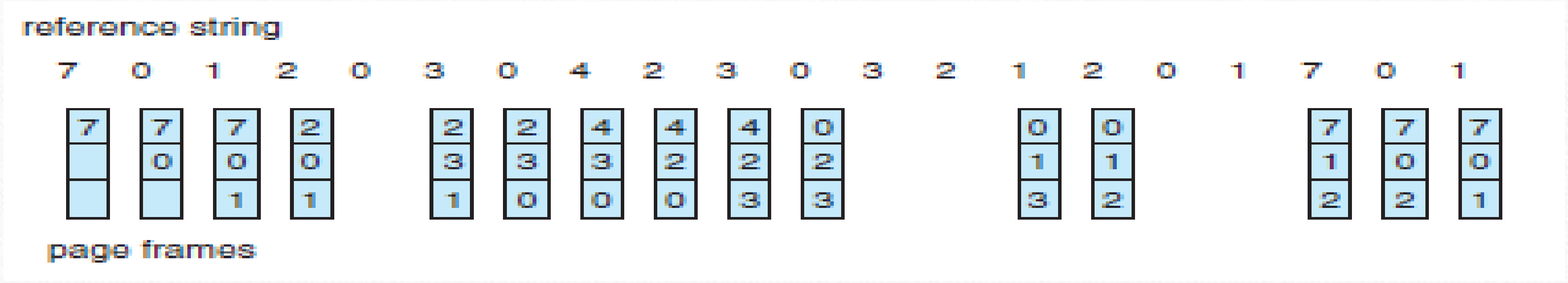
Page Replacement

- FIFO Page Replacement
- Optimal Page Replacement
- LRU Page Replacement
- We evaluate an algorithm by running it on a particular string of memory references and computing the number of page faults (lowest page-fault rate). The string of memory references is called a **reference string.**

FIFO Page Replacement

- A FIFO replacement algorithm associates with each page the time when that page was brought into memory.
- When a page must be replaced, the oldest page is chosen.
- It is not strictly necessary to record the time when a page is brought in.
- We can create a FIFO queue to hold all pages in memory.
- We replace the page at the head of the queue.
- When a page is brought into memory, we insert it at the tail of the queue.

- use the reference string 7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 1, 2, 0, 1, 7, 0, 1 for a memory with three frames



- There are fifteen faults altogether.

- To illustrate the problems that are possible with a FIFO page-replacement algorithm, consider the following reference string:

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

- **Belady's anomaly: for some page-replacement** algorithms, the page-fault rate may ***increase as the number of allocated frames*** increases.

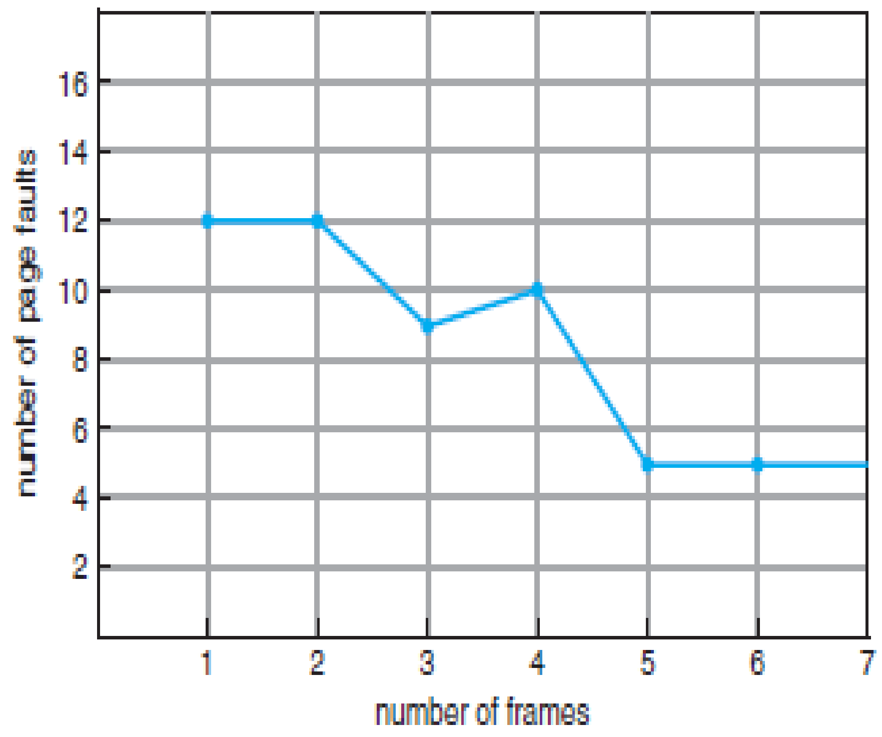


Figure 9.13 Page-fault curve for FIFO replacement on a reference string.

Optimal Page Replacement

- The algorithm that has the lowest page-fault rate of all algorithms and will never suffer from Belady's anomaly.
- Replace the page that will not be used for the longest period of time.
- Unfortunately, the optimal page-replacement algorithm is difficult to implement, because it requires future knowledge of the reference string.
- the optimal algorithm is used mainly for comparison studies.
- For instance, it may be useful to know that, although a new algorithm is not optimal, it is within 12.3 percent of optimal at worst and within 4.7 percent on average.

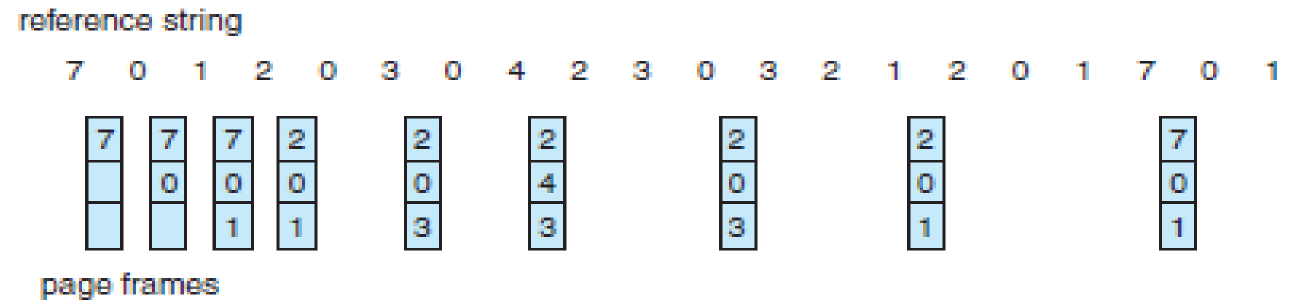
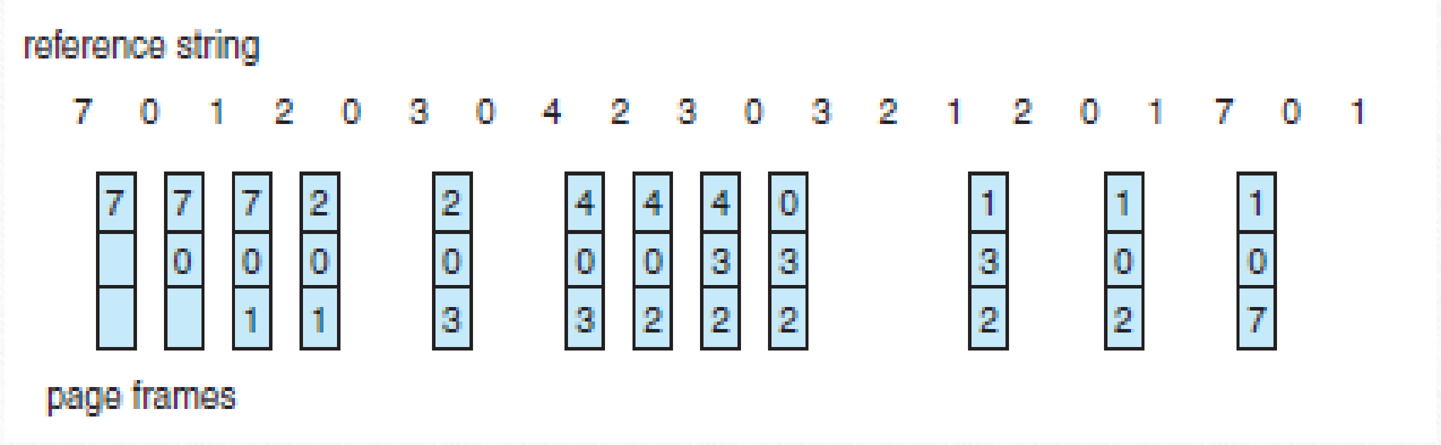


Figure 9.14 Optimal page-replacement algorithm.

LRU Page Replacement

- If the optimal algorithm is not feasible, perhaps an approximation of the optimal algorithm is possible.
- If we use the recent past as an approximation of the near future, then we can replace the page that ***has not been used for the longest period of time. This approach is the least*** **recently used (LRU) algorithm.**

- LRU replacement associates with each page the time of that page’s last use.
- When a page must be replaced, LRU chooses the page that has not been used for the longest period of time. We can think of this strategy as the optimal



- The LRU policy is often used as a page-replacement algorithm and is considered to be good. The major problem is ***how to implement LRU*** replacement (counters,stack).

Thrashing

- Look at any process that does not have "enough" frames.
- If the process does not have the number of frames it needs to support 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, replacing pages that it must bring back in immediately.
- This high paging activity is called **thrashing. A process is thrashing if it is** spending more time paging than executing.