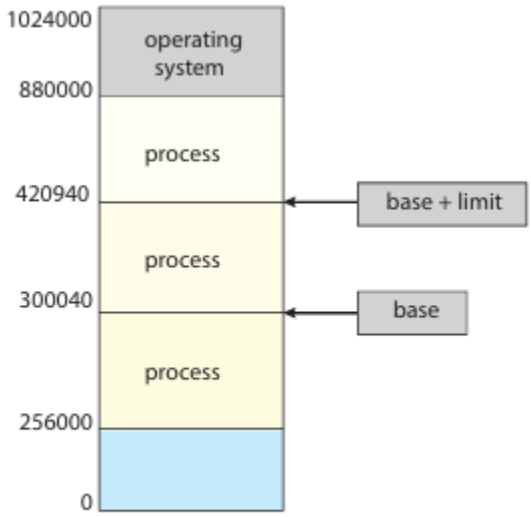
- [ ] We must determine whether this new system state is safe. To do so, we execute our safety algorithm and find that the sequence <P1, P3, P4, P0, P2> satisfies the safety requirement. Hence, we can immediately grant the request of process P1.
- [ ] when the system is in this state, a request for (3,3,0) by P4 cannot be granted, since the resources are not available.
- [ ]

Unit - 3 Memory Management

- [ ] Memory consists of a large array of bytes, each with its own address.



- [ ] **Figure 9.1** A base and a limit register define a logical address space.
- [ ] Each process has a separate memory space
- [ ] 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
- [ ] 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
- 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

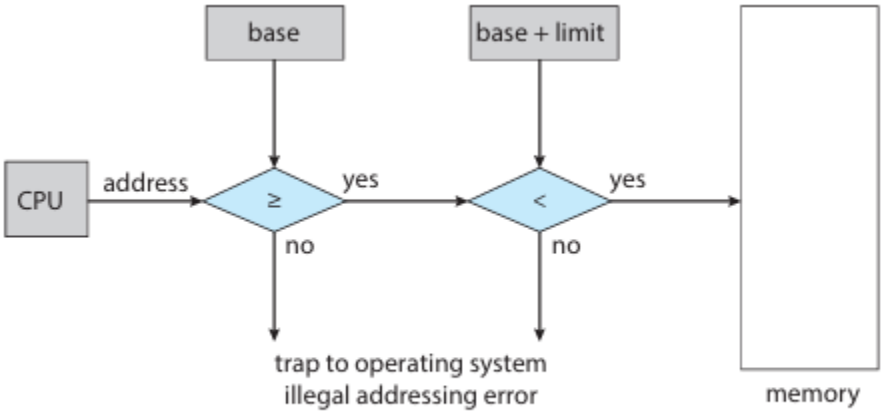


Figure 9.2 Hardware address protection with base and limit registers.

- **Address Binding**

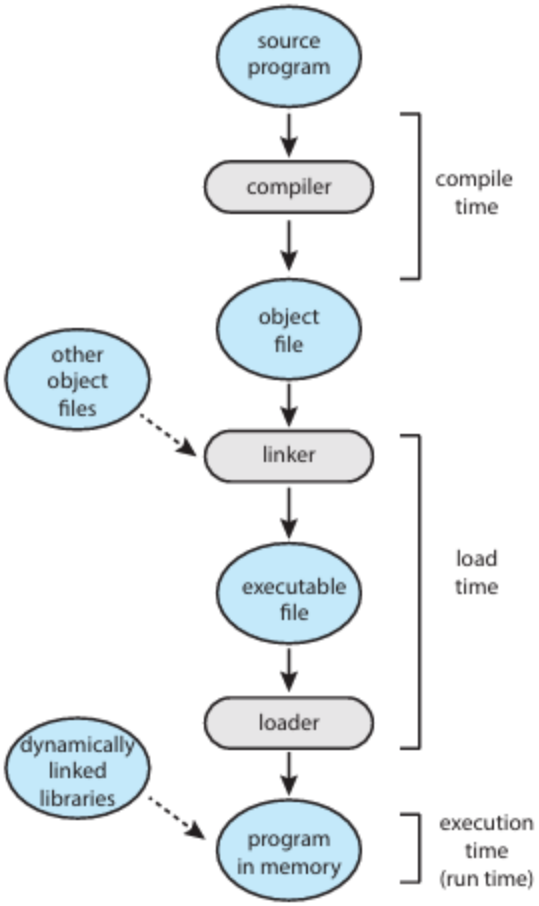


Figure 9.3 Multistep processing of a user program.

- The binding of instructions and data to memory addresses can be done at any step along the way:
- **Compiletime** - If you know at compile time where the process will reside in memory, then absolute code can be generated.
- If, at some later time, the starting location changes, then it will be necessary to recompile this code
- **Loadtime -** If it is not known at compile time where the process will reside in memory, then the compiler must generate relocatable code
- In this case, final binding is delayed until load time
- **Execution time** - If the process can be moved during its execution from one memory segment to another, then binding must be delayed until runtime

- **Logical Versus Physical Address Space**
- An address generated by the CPU is commonly referred to as a **logical address**
- 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**
- Binding addresses at either compile or load time generates identical logical and physical addresses
- However, the execution-time address-binding scheme results in differing logical and physical addresses. In this case, we usually refer to the logical address as a **virtual 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**
- The run-time mapping from virtual to physical addressesis done by a hardware device called the memory-management unit (MMU)

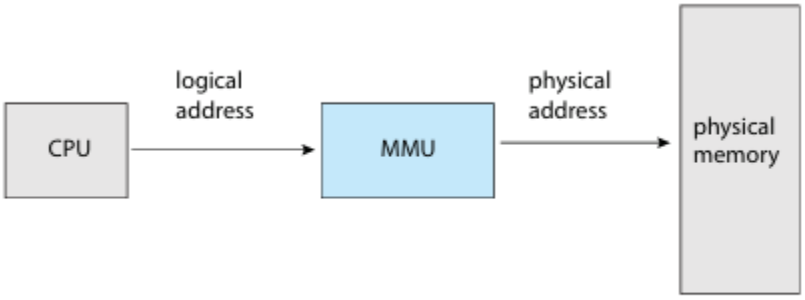


Figure 9.4 Memory management unit (MMU).

- **Dynamic Loading**
- With dynamic loading, All routines are kept on disk in a relocatable load format
- The main program is loaded into memory and is executed
- When a routine needs to call another routine, the calling routine first checks to see whether the other routine has been loaded. If it has not, the relocatable linking loader is called to load the desired routine into memory

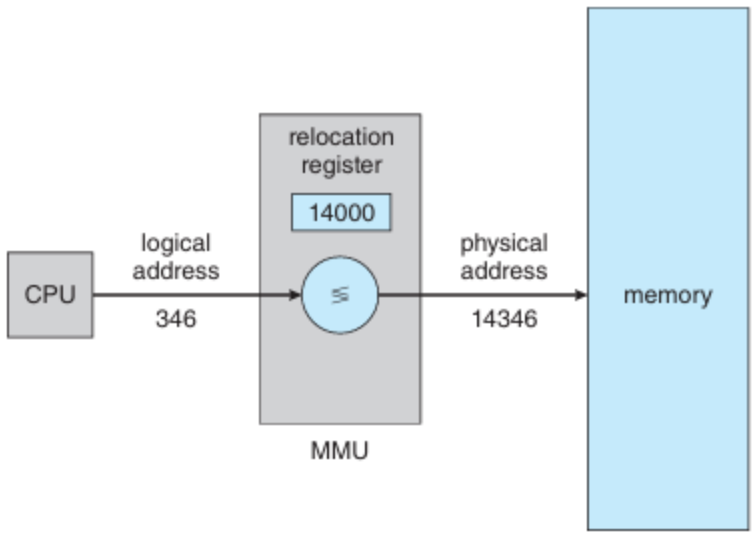


Figure 9.5 Dynamic relocation using a relocation register.

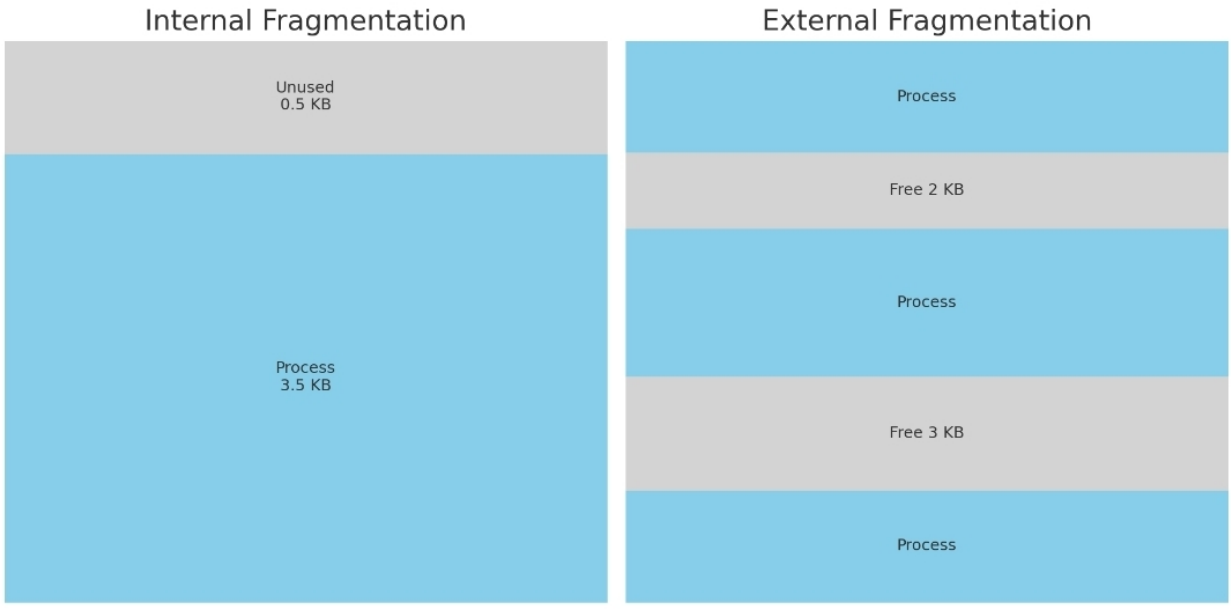
- **Contiguous Memory Allocation**
- The main memory must accommodate both the operating system and the various user processes.
- The memory is usually divided into two partitions:
  - one for the resident operating system and
  - one for the user processes.
- We can place the operating system in either low memory or high memory.
  - The major factor affecting this decision is the location of the interrupt vector. Since the interrupt vector is often in low memory, programmers usually place the operating system in low memory as well.
- We usually want several user processes to reside in memory at the same time.
- In contiguous memory allocation, each process is contained in a single section of memory that is contiguous to the section containing the next process.
- 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.
- ☐ 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.
- ☐ As processes enter the system, they are put into an input queue. When a process is allocated space, it is loaded into memory, and it can then compete for CPU time. When a process terminates, it releases its memory, which the operating system may then fill with another process from the input queue.
- ☐ 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.
- ☐ **dynamic storage-allocation problem** - concerns how to satisfy a request of size n from a list of free holes.
- ☐ Solutions to this problem
- ☐ **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.
- ☐ Problem discussed in class regarding first fit, best fit and worst fit strategies

Memory fragmentation

- ☐ can be internal as well as external.
- ☐ As processes are loaded and removed from memory,the free memory space is broken into little pieces
- ☐ 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 non-contiguous holes
- ☐ Internal fragmentation exists when the memory partitions are allocated to processes, but the process does not use the entire 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
- ☐ Two complementary techniques achieve this solution: segmentation and paging

Segmentation

- ☐ Segmentation permits the physical address space of a process to be noncontiguous
- ☐ 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.
- ☐ Segments vary in length, and the length of each is intrinsically defined by its purpose in the program.
- ☐ Segmentation is a memory-management scheme that supports this programmer view of memory.
- ☐ A logical address space is a collection of segments.
- ☐ Each segment has a name and a length.
- ☐ The addresses specify both the segment name and the offset within the segment.
- ☐ The programmer therefore specifies each address by two quantities: a segment name and an offset.
- ☐ A logical address: <segment-number, offset>.

- ☐ A C compiler might create separate segments for the following:
- ☐ 1. The code
- ☐ 2. Global variables
- ☐ 3. The heap, from which memory is allocated
- ☐ 4. The stacks used by each thread
- ☐ 5. 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.
- ☐ **Segmentation Hardware**
- ☐ 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.
- ☐ 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 offset d of the logical address must be between 0 and the segment limit.
- ☐

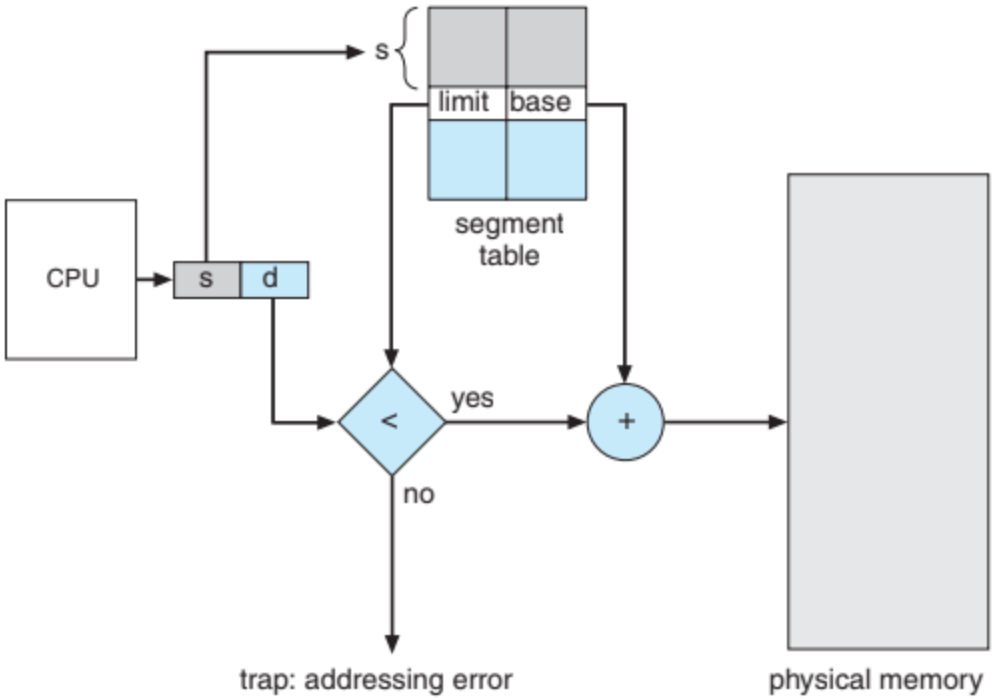


Figure 8.8 Segmentation hardware.

- Example of Segmentation
- We have five segments numbered from 0 through 4.
- The segment table has a separate entry for each segment, giving the beginning address of the segment in physical memory (or base) and the length of that segment (or limit).
- a reference to byte 53 of segment 2 is mapped onto location 4300 + 53 = 4353.
- A reference to segment 3, byte 852, is mapped to 3200 (the base of segment 3) + 852 = 4052.
- A reference to byte 1222 of segment 0 would result in a trap to the operating system, as this segment is only 1,000 bytes long.
-

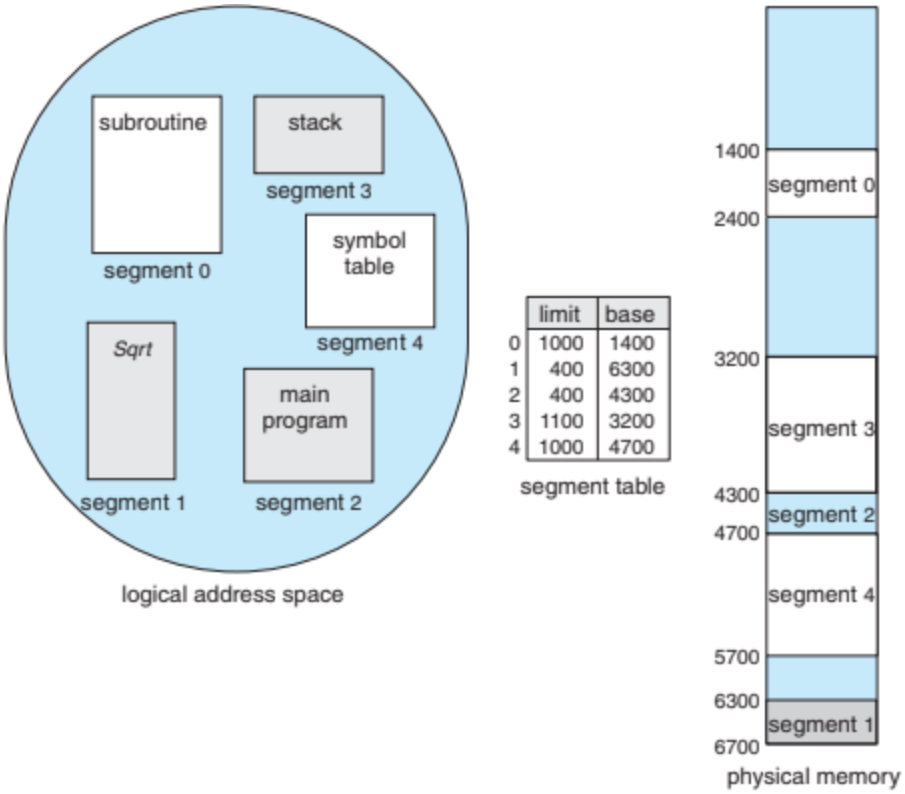


Figure 8.9 Example of segmentation.

- ☐
- ☐

Paging

- ☐ Paging permits the physical address space of a process to be noncontiguous
- ☐ paging involves breaking physical memory into fixed-sized blocks called **frames**
- ☐ and breaking logical memory into blocks of the same size called **pages**.
- ☐ Hardware support

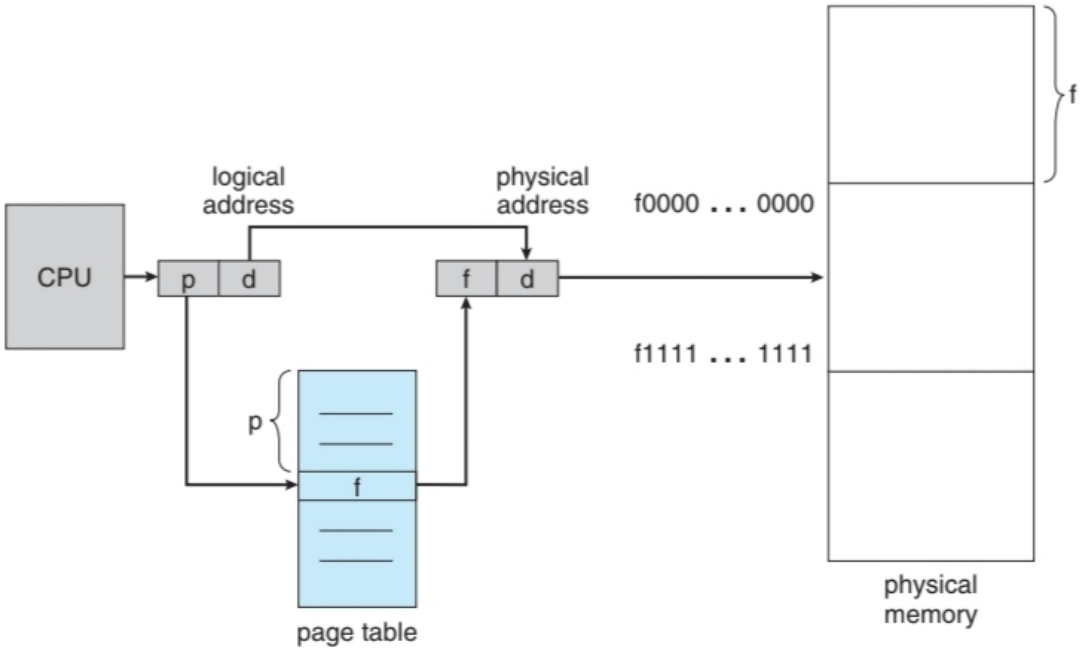


Figure 8.10 Paging hardware.

- 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

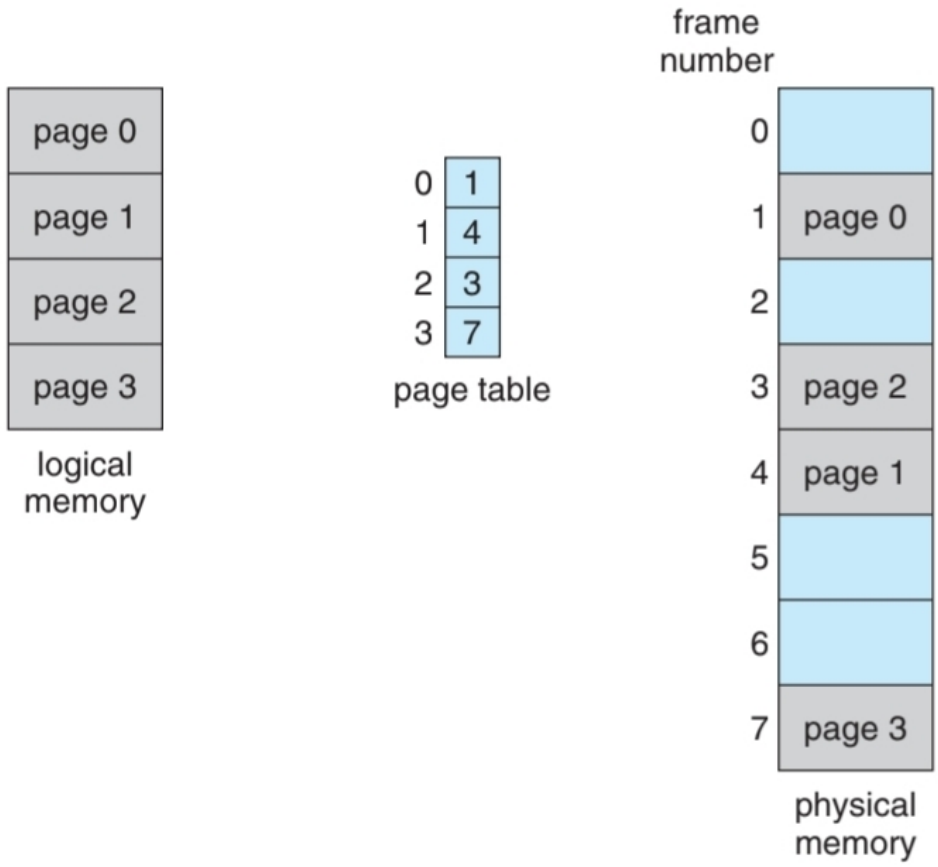


Figure 8.11 Paging model of logical and physical memory.

- ☐
- ☐ The size of a page is a power of 2, varying between 512 bytes and 1 GB per page, depending on the computer architecture
- ☐ The selection of a power of 2 as a page size makes the translation of a logical address into a page number and page offset particularly easy
- ☐ If the size of the logical address space is 2 power m, and a page size is 2 power n bytes, then the high-order m − n bits of a logical address designate the page number, and the n

low-order bits designate the page offset. Thus, the logical address is as follows:

- ☐
- ☐ Example
- ☐ Using a page size of 4 bytes and a physical memory of 32 bytes (8 pages),
- ☐ Logical address 0 is page 0, offset 0. Indexing into the page table, we find that page 0 is in frame 5. Thus, logical address 0 maps to physical address 20 [= (5 × 4) + 0]. Logical address 3 (page 0, offset 3) maps to physical address 23 [= (5 × 4) + 3]. Logical address 4 is page 1, offset 0; according to the page table, page 1 is mapped to frame 6. Thus, logical address 4 maps to physical address 24 [= (6 × 4) + 0]. Logical address 13 maps to physical address 9

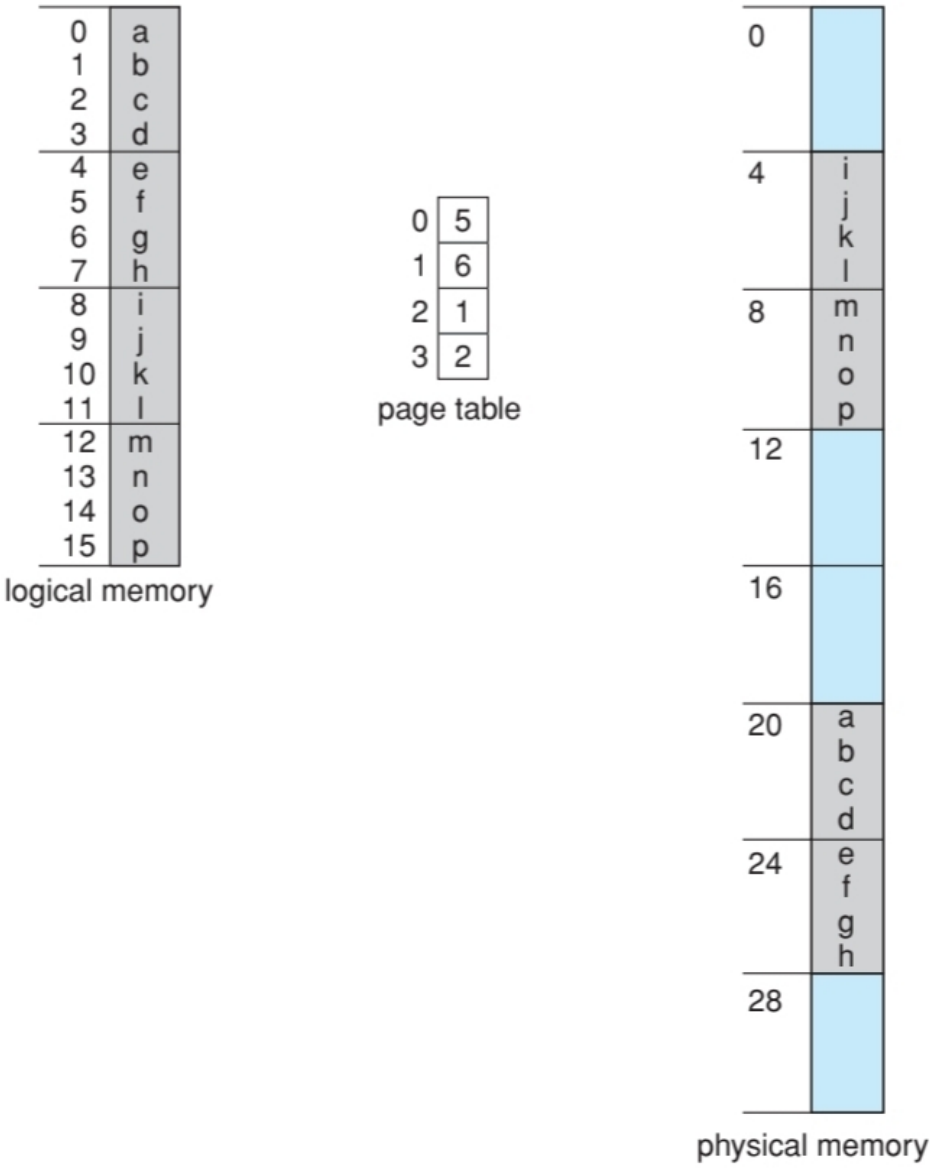


Figure 8.12 Paging example for a 32-byte memory with 4-byte pages.

- 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.

- ☐ For example, if page size is 2,048 bytes, a process of 72,766 bytes will need 35 pages plus 1,086 bytes.
- ☐ It will be allocated 36 frames, resulting in internal fragmentation of 2,048 −1,086 = 962 bytes.

Hardware Support

- ☐ Page tables are per-process data structures
- ☐ a pointer to the page table is stored with the other register values (like the instruction pointer) in the process control block of each process.
- ☐ The page table is implemented as a set of dedicated high-speed hardware registers. This approach increases context-switch time
- ☐ The use of registers for the page table is satisfactory if the page table is reasonably small (for example, 256 entries).
- ☐ When the page table is large, the page table is kept in main memory, and a page-table base register (PTBR) points to the page table. Changing page tables requires changing only this one register, substantially reducing context-switch time.
- ☐ **Translation Look-Aside Buffer**
- ☐ Although storing the page table in main memory can yield faster context switches, it may also result in slower memory access times.
- ☐ The standard solution to this problem is to use a special, small, fast-lookup hardware cache called a translation look-aside buffer (TLB).
- ☐ 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.
- ☐ **TLB Miss**
    - ☐ If the page number is not in the TLB, it is known as a TLB miss. When the frame number is obtained, 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.
- ☐ Some TLBs allow certain entries to be **wired down**, meaning that they cannot be removed from the TLB.
- ☐ Some TLBs store **address-space identifier (ASIDs)** in each TLB entry.
- ☐ An ASID uniquely identifies each process and is used to provide address-space protection for that process.

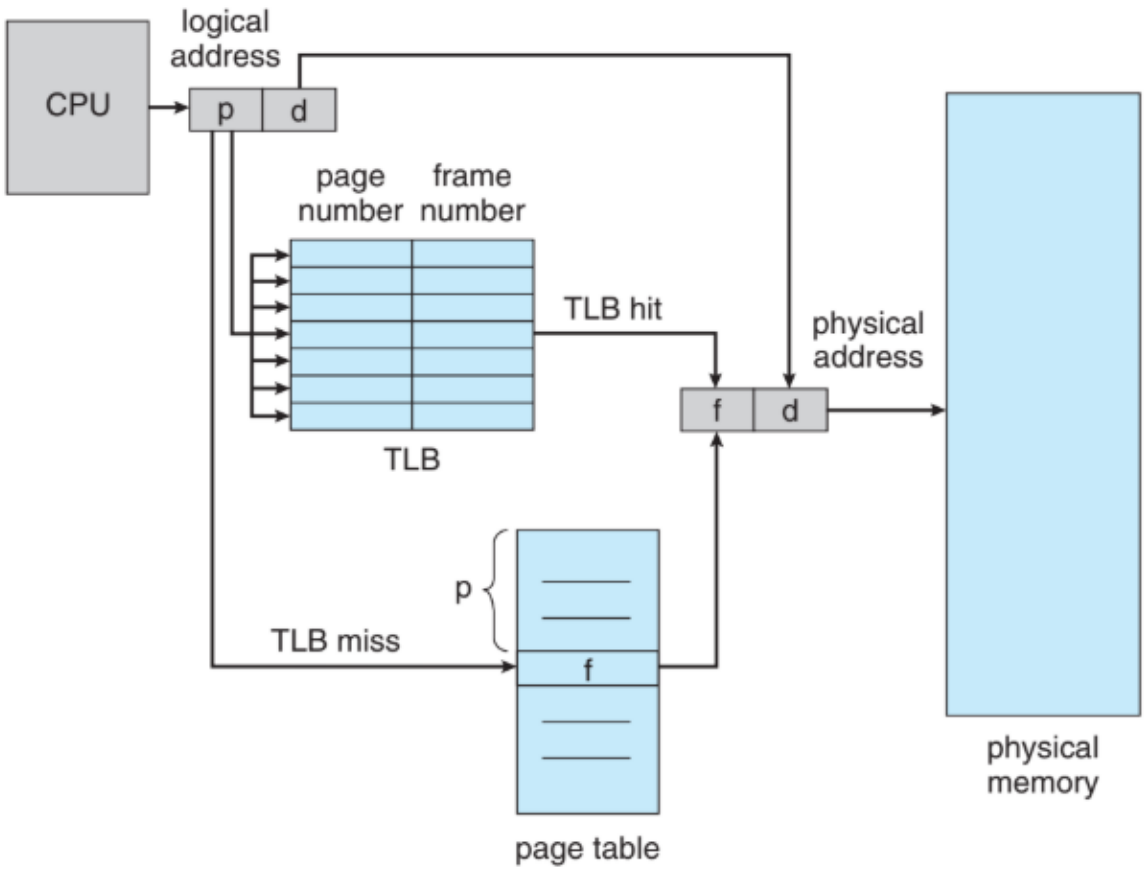


Figure 9.12 Paging hardware with TLB.

- Example: Find the Effective Memory Access Time in a paging scheme with 80-percent hit ratio and memory access time of 10ns
- An 80-percent hit ratio, for example, means that we find the desired page number in the TLB 80 percent of the time. If it takes 10 nanoseconds to access memory, then a mapped-memory access takes 10 nanoseconds when the page number is in the TLB.
- If we fail to find the page
- number in the TLB then we must first access memory for the page table and frame number (10 nanoseconds) and then

access the desired byte in memory (10 nanoseconds), for a total of 20 nanoseconds.

- ☐ Effective access time = 0.80 × 10 + 0.20 × 20
- ☐ = 12 nanoseconds
- ☐ we suffer a 20-percent slowdown in average memory-access time (from 10 to 12 nanoseconds).
- ☐ For a 99-percent hit ratio,

$$\begin{aligned}\text{effective access time} &= 0.99 \times 10 + 0.01 \times 20 \\ &= 10.1 \text{ nanoseconds}\end{aligned}$$

This increased hit rate produces only a 1 percent slowdown in access time.

- ☐
- ☐

Page Replacement

- ☐ If no frame is free, we find one that is not currently being used and free it.
- ☐ We can free a frame by writing its contents to swap space and changing the page table (and all other tables) to indicate that the page is no longer in memory.
- ☐ We can now use the freed frame to hold the page for which the process faulted

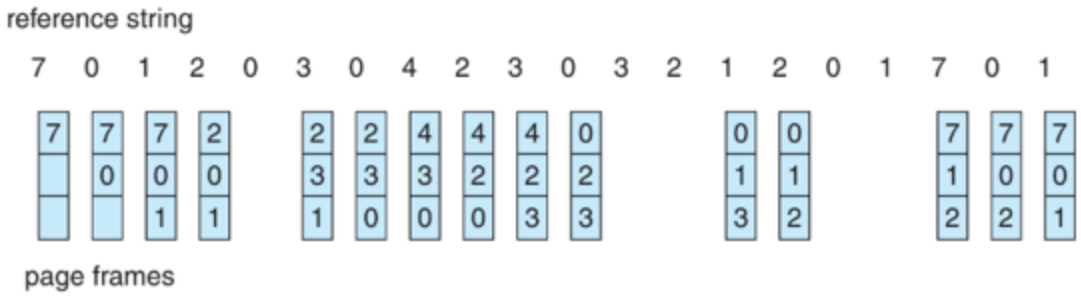


Figure 10.12 FIFO page-replacement algorithm.

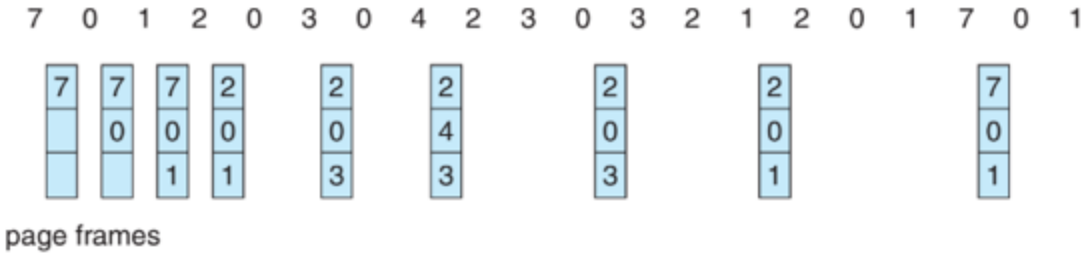


Figure 10.14 Optimal page-replacement algorithm.

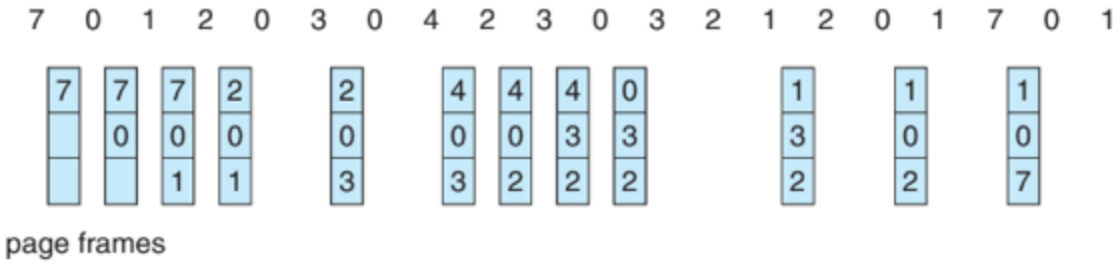


Figure 10.15 LRU page-replacement algorithm.

☐

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
- ☐ 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**
- A demand-paging system is similar to a paging system with swapping
- A lazy swapper is used.
- A lazy swapper never swaps a page into memory unless that page will be needed

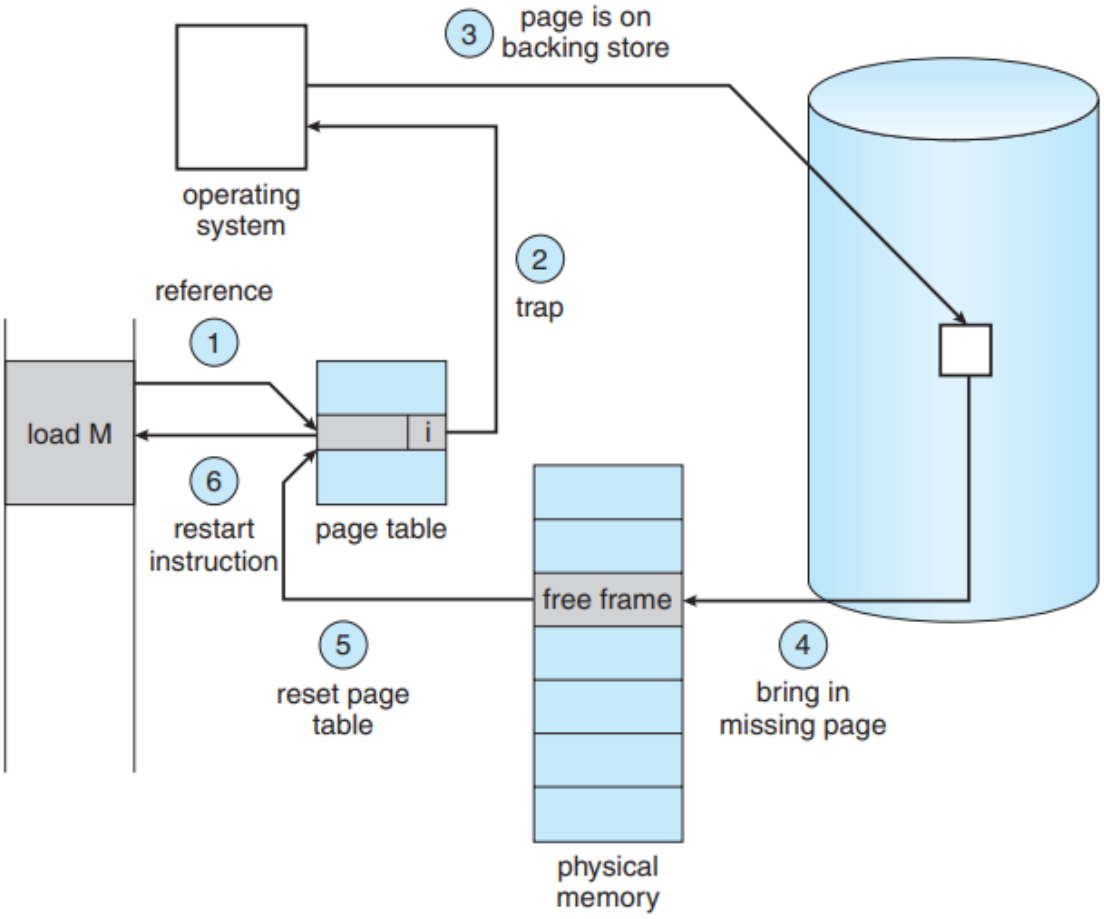


Figure 9.6 Steps in handling a page fault.

The procedure for handling this page fault is straightforward (Figure 9.6):

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.

☐

☐ A page fault causes the following sequence to occur:
☐ 1. Trap to the operating system.
☐ 2. Save the user registers and process state
☐ 3. Determine that the interrupt was a page fault.
☐ 4. Check that the page reference was legal and determine the location of the page on the disk.
☐ 5. Issue a read from the disk to a free frame:
- ☐ a.Wait in a queue for this device until the read request is serviced.
- ☐ b. Wait for the device seek and/or latency time.
- ☐ c. Begin the transfer of the page to a free frame.

☐ 6. While waiting, allocate the CPU to some other user (CPU scheduling, optional).
☐ 7. Receive an interrupt from the disk I/O subsystem (I/O completed).

☐ 8. Save the registers and process state for the other user (if step 6 is executed).

☐

☐ **Copy-on-Write**

☐ Recall that the fork() system call creates a child process that is a duplicate of its parent

☐ Traditionally, fork() worked by creating a copy of the parent's address space for the child, duplicating the pages belonging to the parent

☐ Instead, we can use a technique known as copy-on-write, which works by allowing the parent and child processes initially to share the same pages.

☐ These shared pages are marked as copy-on-write pages, meaning that if either process writes to a shared page, a copy of the shared page is created.

☐ Copy-on-write is illustrated in Figures 9.7 and 9.8, which show the contents of the physical memory before and after process 1 modifies page C

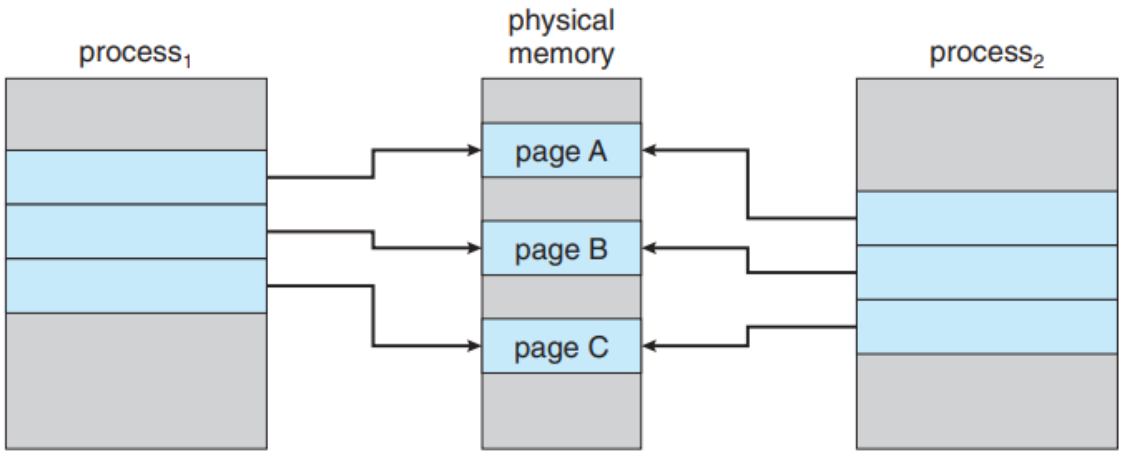


Figure 9.7 Before process 1 modifies page C.

☐

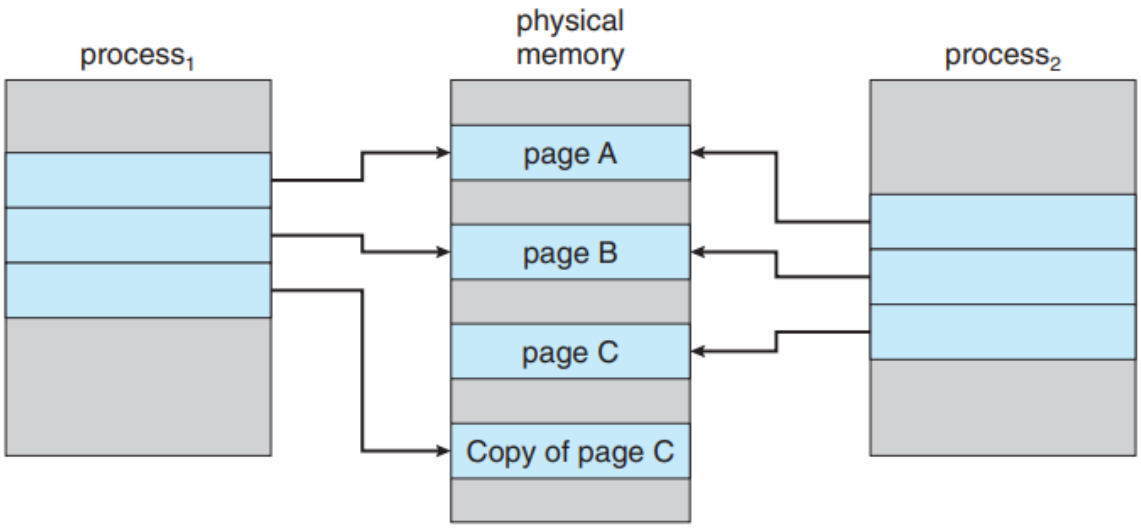


Figure 9.8 After process 1 modifies page C.

- **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

Magnetic Disks

- Magnetic disks provide the bulk of secondary storage for modern computer systems
- Each disk platter has a flat circular shape, like a CD
- The two surfaces of a platter are covered with a magnetic material.

- ☐ We store information by recording it magnetically on the platters

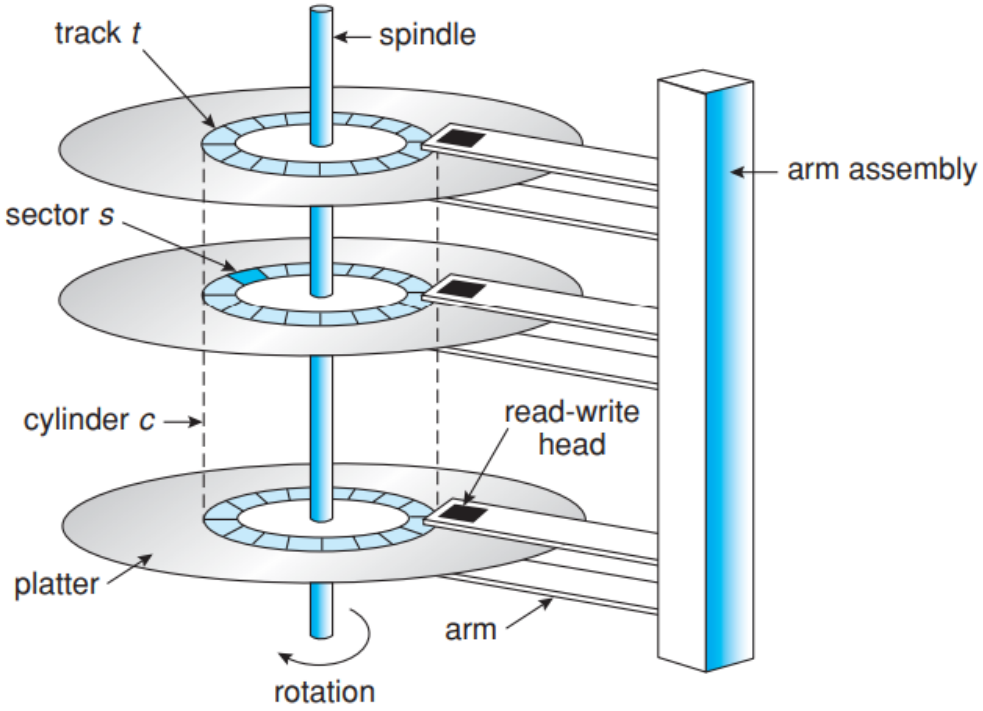


Figure 10.1 Moving-head disk mechanism.

- ☐
- ☐ A read–write head “flies” just above each surface of every platter.
- ☐ The heads are attached to a disk arm that moves all the heads as a unit.
- ☐ The surface of a platter is logically divided into circular tracks, which are subdivided into sectors.
- ☐ The set of tracks that are at one arm position makes up a cylinder

Disk scheduling

- ☐ Disk scheduling manages how data is read from/written into the hard disk

- ☐ **Seek Time** - The time taken to move the disk arm to the track where the data is located
- ☐ **Rotational Latency** - Time taken for the desired sector to rotate under the read/write head
- ☐ **Transfer Time** - Time taken to read or write the data
- ☐ **Disk Access Time** = Seek Time + Rotational Latency + Transfer Time
- ☐ **Disk scheduling algorithms**
- ☐ Total no. of cylinders = 200
- ☐ Requests for I/O to blocks on cylinders
- ☐ 98, 183, 37, 122, 14, 124, 65, 67
- ☐ disk head is initially at cylinder 53
- ☐ **1. FCFS Scheduling**
- ☐ first-come, first-served (FCFS) algorithm
- ☐

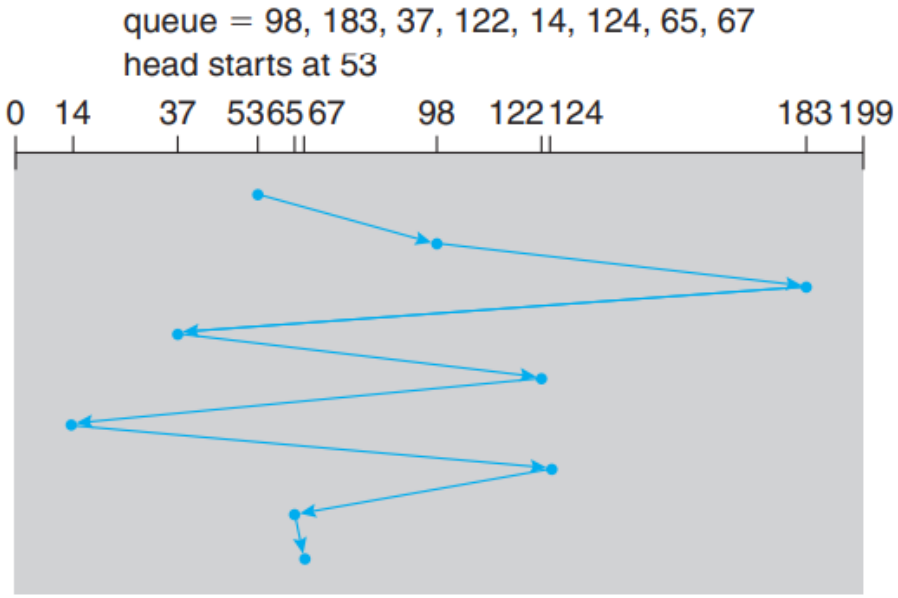


Figure 10.4 FCFS disk scheduling.

☐ Total head movement = 640 cylinders

☐ **2. SSTF Scheduling**

☐ shortest-seek-time-first (SSTF) algorithm

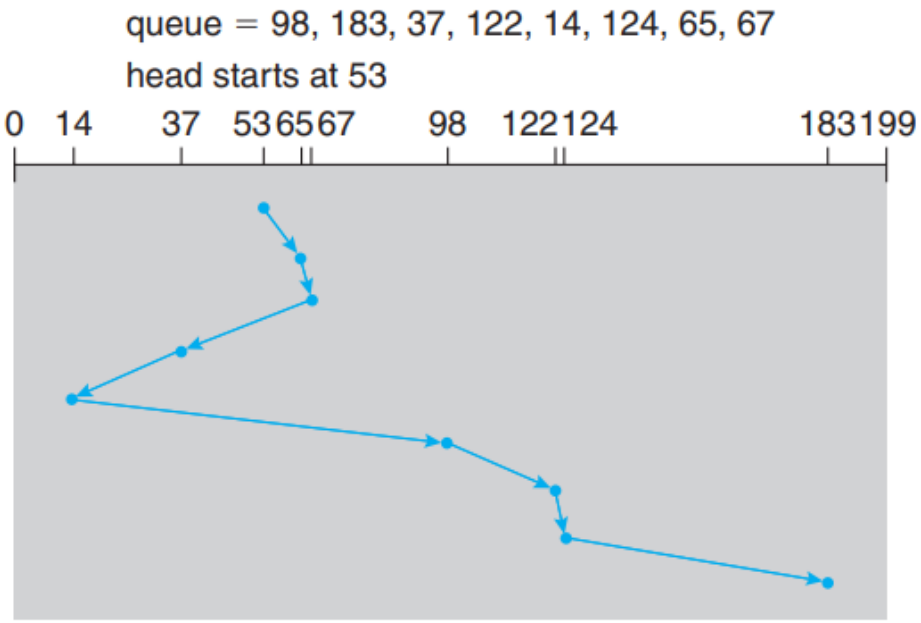


Figure 10.5 SSTF disk scheduling.

☐

☐ Total head movement = 208 cylinders

☐ **3. SCAN Scheduling**

☐ SCAN algorithm is sometimes called the elevator algorithm

☐ Assuming the disk arm is moving towards left

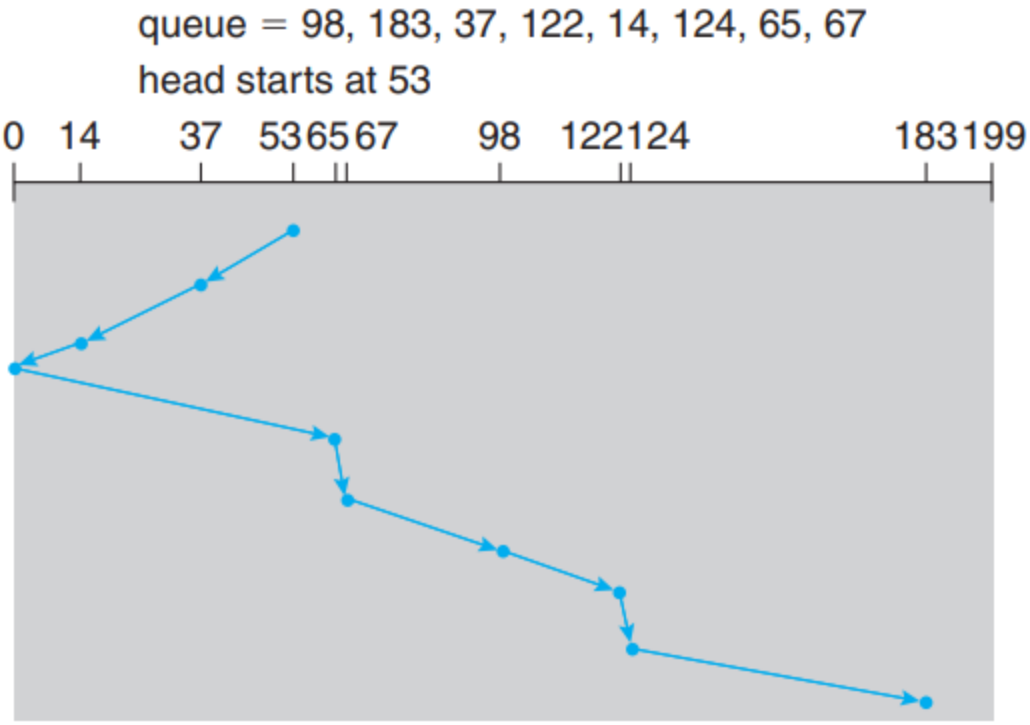


Figure 10.6 SCAN disk scheduling.

☐

☐ **4. C-SCAN Scheduling**

☐ Circular SCAN (C-SCAN) scheduling is a variant of SCAN designed to provide a more uniform wait time

☐ Assuming head moves towards the right

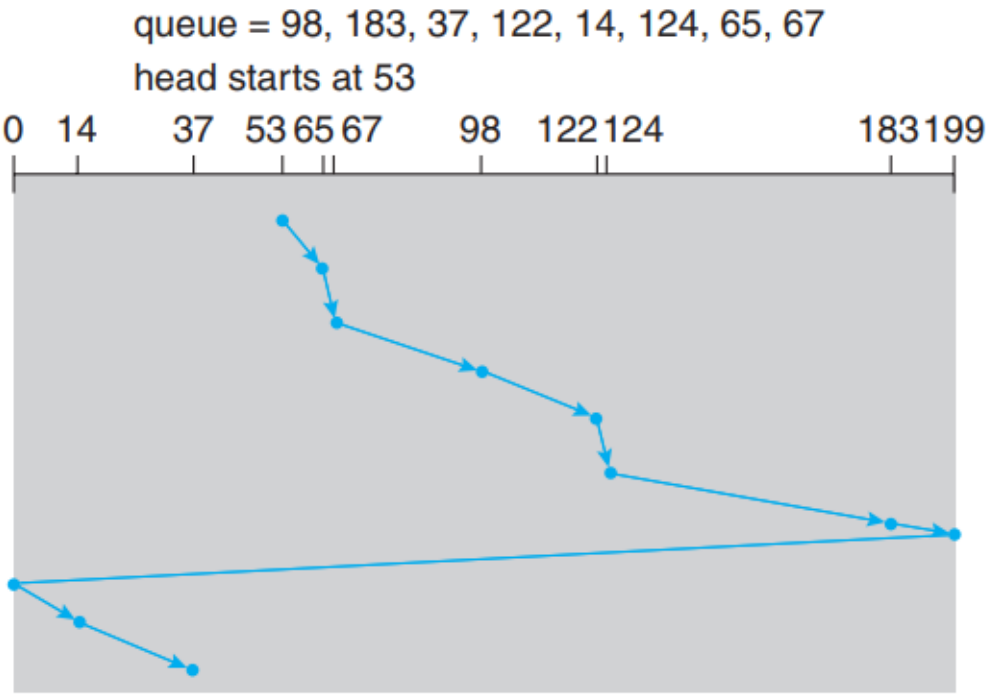


Figure 10.7 C-SCAN disk scheduling.

☐

☐ **C-LOOK Scheduling**

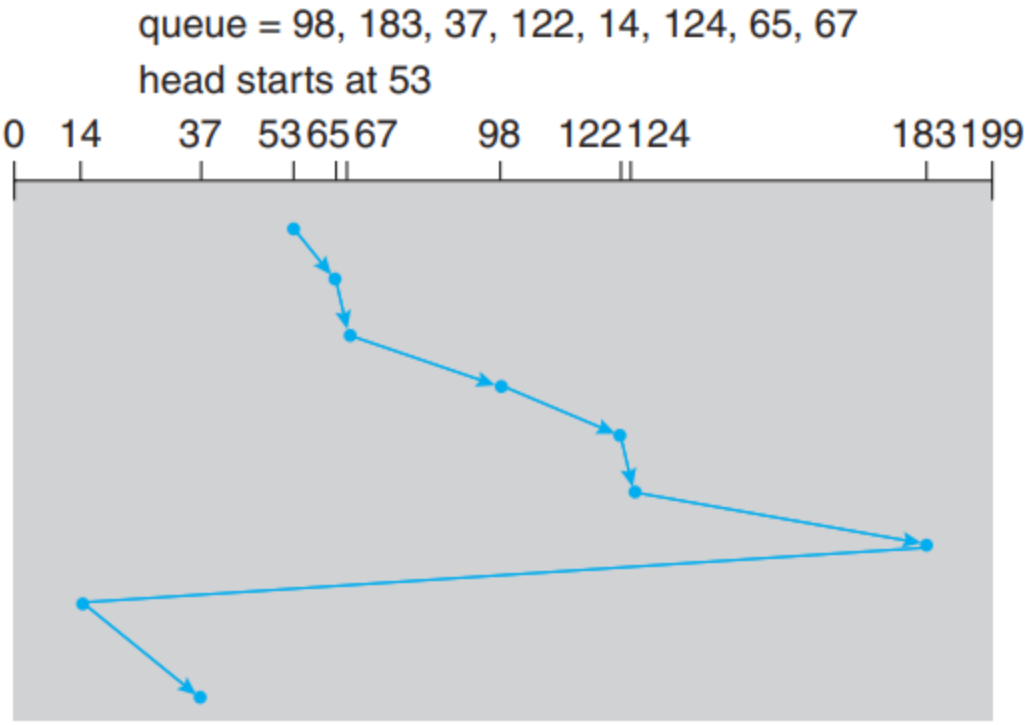


Figure 10.8 C-LOOK disk scheduling.

- [ ]

File System

- [ ] A file is a collection of related information defined by its creator.
- [ ] Files are mapped by the operating system onto physical mass-storage devices.
- [ ] Accessing physical storage can often be slow, so file systems must be designed for efficient access.
- [ ] **File Concept**
- [ ] A file is a named collection of related information that is recorded on secondary storage.

- ☐ a file is the smallest allotment of logical secondary storage; that is, data cannot be written to secondary storage unless they are within a file.
- ☐ Files represent programs (both source and object forms) and data.
- ☐ Data files may be numeric, alphabetic, alphanumeric, or binary.
- ☐ **File Attributes**
- ☐ A file's attributes vary from one operating system to another but typically consist of these:
- ☐ **Name**. The symbolic file name is the only information kept in human-readable form.
- ☐ **Identifie**r. This unique tag, usually a number, identifies the file within the file system; it is the non-human-readable name for the file.
- ☐ **Type.** This information is needed for systems that support different types of files.
- ☐ **Location**. This information is a pointer to a device and to the location of the file on that device.
- ☐ **Size**. The current size of the file (in bytes, words, or blocks) and possibly the maximum allowed size are included in this attribute.
- ☐ **Protection**. Access-control information determines who can do reading, writing, executing, and so on.
- ☐ **Timestamps and user identification** - This information may be kept for creation, last modification, and last use. These data can be useful for protection, security, and usage monitoring.

File Operations

- ☐ **Creating a file**
  - ☐ First, space in the file system must be found for the file
  - ☐ Second, an entry for the new file must be made in the directory
- ☐ **Writing a file**
  - ☐ The system must keep a write pointer to the location in the file where the next write is to take place.
  - ☐ The write pointer must be updated whenever a write occurs
- ☐ **Reading a file**
  - ☐ the directory is searched for the associated entry
  - ☐ the system needs to keep a read pointer to the location in the file where the next read is to take place.
  - ☐ Once the read has taken place, the read pointer is updated
- ☐ **Repositioning within a file**
  - ☐ The directory is searched for the appropriate entry, and the current-file-position pointer is repositioned to a given value
- ☐ **Deleting a file**
  - ☐ To delete a file, we search the directory for the named file
  - ☐ Having found the associated directory entry, we release all file space, so that it can be reused by other files
  - ☐ erase the directory entry
- ☐

- **Truncating a file.** Rather than forcing the user to delete the file and then recreate it, this function allows all attributes to remain unchanged—except for file length

File Types

file type	usual extension	function
executable	exe, com, bin or none	ready-to-run machine-language program
object	obj, o	compiled, machine language, not linked
source code	c, cc, java, perl, asm	source code in various languages
batch	bat, sh	commands to the command interpreter
markup	xml, html, tex	textual data, documents
word processor	xml, rtf, docx	various word-processor formats
library	lib, a, so, dll	libraries of routines for programmers
print or view	gif, pdf, jpg	ASCII or binary file in a format for printing or viewing
archive	rar, zip, tar	related files grouped into one file, sometimes compressed, for archiving or storage
multimedia	mpeg, mov, mp3, mp4, avi	binary file containing audio or A/V information

Figure 11.3 Common file types.

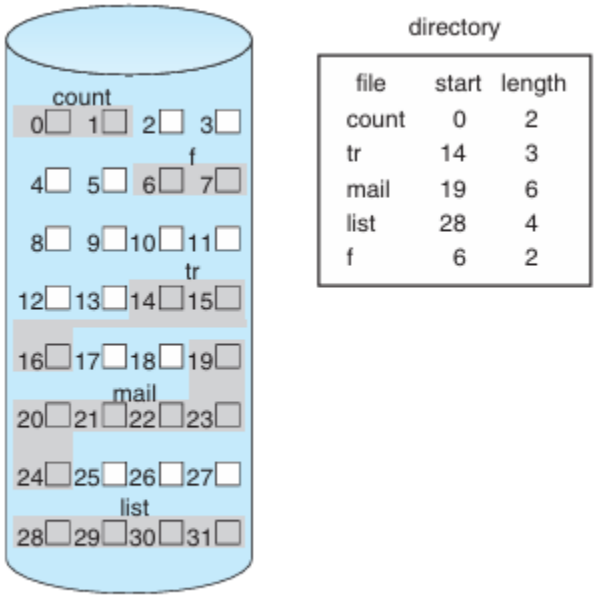
File Access Methods

- ☐ **Sequential Access**
- ☐ Information in the file is processed in order, one record after the other
- ☐ A read operation—read next()—reads the next portion of the file and automatically advances a file pointer
- ☐ the write operation—write next()—appends to the end of the file and advances to the end of the newly written material (the new end of file)
- ☐ Such a file can be reset to the beginning
- ☐ may be able to skip forward or backward 1 record
- ☐ **Direct Access**
- ☐ Another method is direct access (or relative access)
- ☐ le is made up of fixed-length logical records that allow programs to read and write records rapidly in no particular order
- ☐ the file is viewed as a numbered sequence of blocks or records. Thus, we may read block 14, then read block 53, and then write block 7. There are no restrictions on the order of reading or writing for a direct-access file
- ☐ Direct-access files are of great use for immediate access to large amounts of information. Databases are often of this type

File Allocation Methods

- ☐ many files are stored on the same device. The main problem is how to allocate space to these files so that storage space is utilized effectively and files can be accessed quickly
- ☐ Three major methods of allocating secondary storage space are in wide use: contiguous, linked, and indexed

☐ Contiguous Allocation



☐ **Figure 14.4** Contiguous allocation of disk space.

☐ Contiguous allocation requires that each file occupy a set of contiguous blocks on the disk

☐ If the file is n blocks long and starts at location b, then it occupies blocks b, b + 1, b + 2, ..., b + n − 1

☐ **Advantages**

☐ Storing files sequentially is very easy

☐ Accessing a file frequently sequentially is made easy

☐ both **sequential and direct access can be supported** by contiguous allocation

☐ **Disadvantages**:

☐ finding space for a new file

☐ finding space for files that grow over time

☐ First fit and best fit are the most common strategies used to select a free hole from the set of available holes - All these

algorithms suffer from the problem of **external fragmentation -**

- Solution is **Compaction -** to copy an entire file system onto another disk. The original disk is then freed completely, creating one large contiguous free space, but **time-consuming**

- Another problem with contiguous allocation is determining how much space is needed for a file
-
- **Linked Allocation**

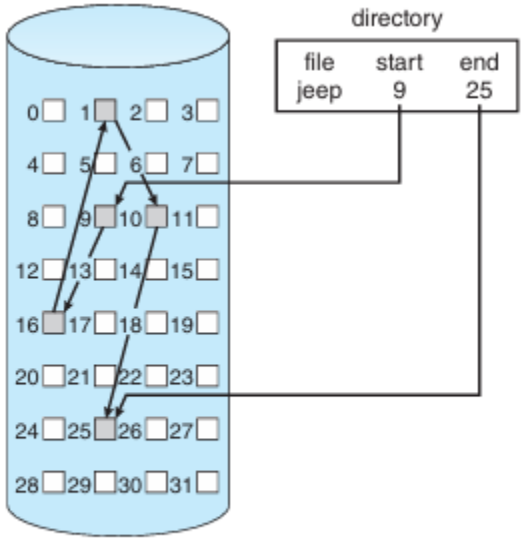


Figure 14.5 Linked allocation of disk space.

-
- Linked allocation solves all problems of contiguous allocation
- each file is a linked list of disk blocks; the disk blocks may be scattered anywhere on the disk
- The directory contains a pointer to the first and last blocks of the file
- Each block contains a pointer to the next block

- **Advantages**
- To **create a new file**, we simply create a new entry in the directory
- A **write to the file** causes the free-space management system to find a free block, and this new block is written to and is linked to the end of the file
- To **read a file**, we simply read blocks by following the pointers from block to block
- There is no external fragmentation
- A file can continue to grow as long as free blocks are available
- Sequential access of files is easy
-
- **Disadvantages**
- Inefficient to direct access of files
- Space required for pointers
    - If a pointer requires 4 bytes out of a 512-byte block, then 0.78 percent of the disk is being used for pointers, rather than for information
    - The usual solution to this problem is to collect blocks into multiples, called clusters, and to allocate clusters rather than blocks, but can result in internal fragmentation
- what would happen if a pointer were lost or damaged

An important variation on linked allocation is the use of a **file-allocation table (FAT)**

- The directory entry contains the block number of the first block of the file

- ☐ The table entry indexed by that block number contains the block number of the next block in the file
- ☐ This chain continues until it reaches the last block, which has a special end-of-file value
- ☐ the FAT is cached

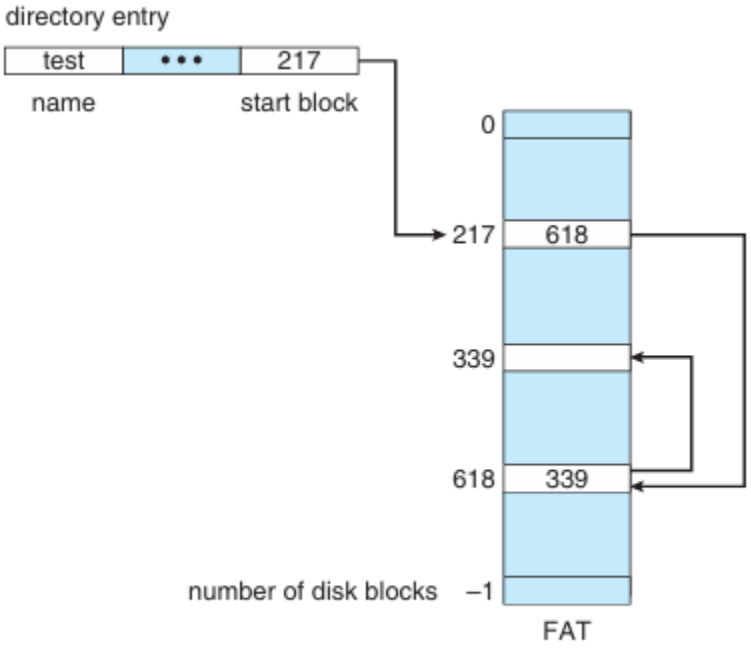


Figure 14.6 File-allocation table.

- ☐
- ☐
- ☐ **Indexed Allocation**
- ☐ all the pointers are brought together into one location: the **index block**
- ☐ Each file has its own index block, which is an array of disk-block addresses
- ☐ The ith entry in the index block points to the ith block of the file
- ☐ **Advantages**

- Indexed allocation **supports direct access**, without suffering from external fragmentation
- any free block on the disk can satisfy a request for more space
- **Disadvantages**
- Pointer overhead of the index block is generally greater than the pointer overhead of linked allocation

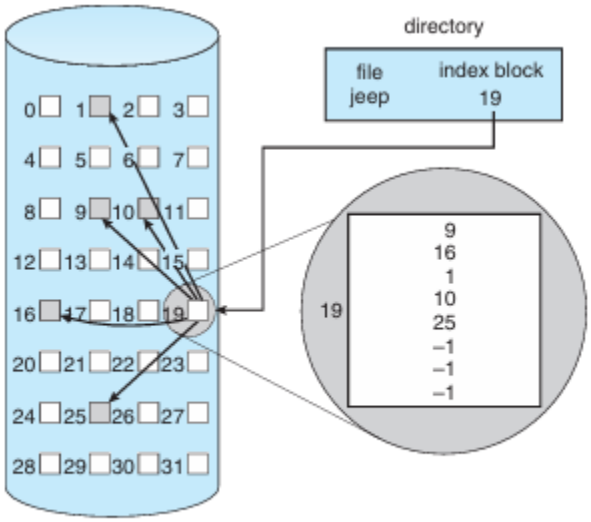


Figure 14.7 Indexed allocation of disk space.

- Mechanisms for large files include the following
- 1. Linked scheme
  - we can link together several index blocks
- 2. Multilevel index
  - uses a first-level index block to point to a set of second-level index blocks, which in turn point to the file blocks
  - This approach could be continued to a third or fourth level
- 3. Combined scheme
  - The first set of pointers point to direct blocks(data)

- The next set of pointers point to indirect blocks
  - Single indirect blocks
  - Double indirect blocks
  - Triple indirect blocks

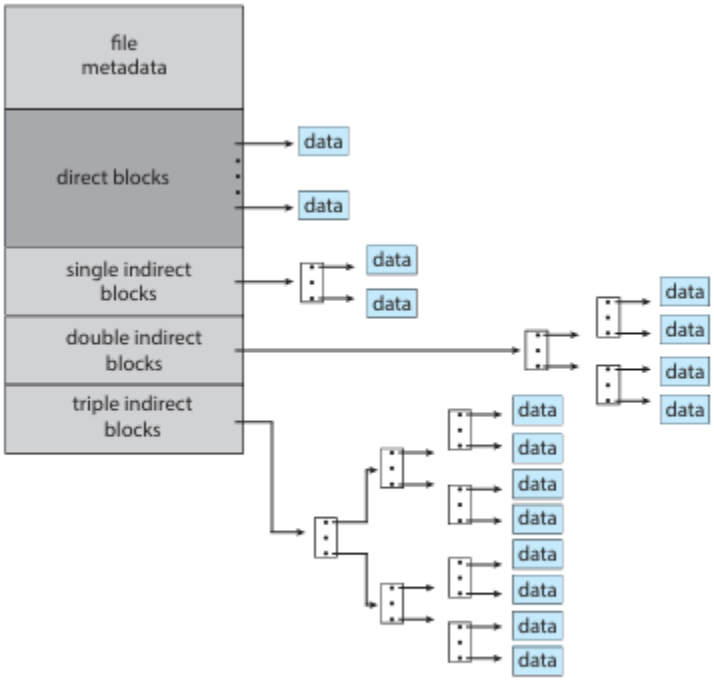


Figure 14.8 The UNIX inode.

Free-Space Management

- Since storage space is limited, we need to reuse the space from deleted files for new files, if possible.
- To keep track of free disk space, the system maintains a free-space list.
- **1.Bit Vector**
- the free-space list is implemented as a bitmap or bit vector
- Each block is represented by 1 bit. If the block is free, the bit is 1; if the block is allocated, the bit is 0

- For example, consider a disk where blocks 2, 3, 4, 5, 8, 9, 10, 11, 12, 13, 17, 18, 25, 26, and 27 are free and the rest of the blocks are allocated. The free-space bitmap would be 00111100111111000110000011100000 ..
- The main advantage of this approach is its relative simplicity and its efficiency in finding the first free block or n consecutive free blocks on the disk
- Disadv - Keeping it in main memory is possible for smaller devices but not necessarily for larger ones
- **2. Linked List**

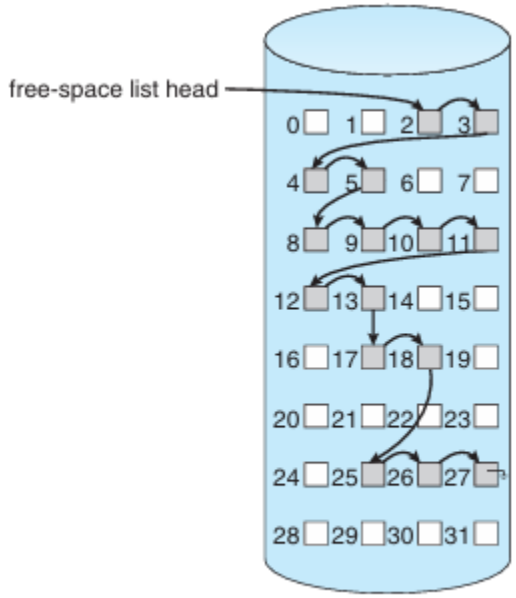


Figure 14.9 Linked free-space list on disk.

- **3. Grouping**
- stores the addresses of n free blocks in the first free block.
- The first n−1 of these blocks are actually free.
- The last block contains the addresses of another n free blocks, and so on
- **4. Counting**

- rather than keeping a list of n free block addresses, we can keep the address of the first free block and the number (n)offreecontiguousblocksthat follow the first block

Directory Structure

- The directory can be viewed as a symbol table that translates file names into their file control blocks
- If we take such a view, we see that the directory itself can be organized in many ways.
- The organization must allow us to insert entries, to delete entries, to search for a named entry, and to list all the entries in the directory.
- **1.Single-Level Directory**
- All files are contained in the same directory, which is easy to support and understand

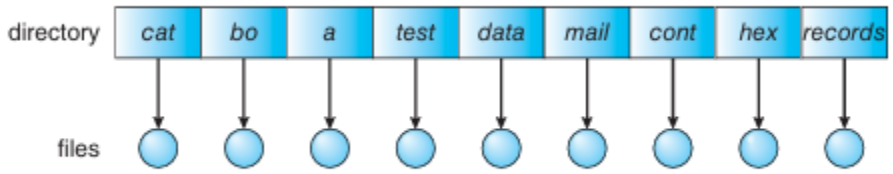


Figure 13.7 Single-level directory.

- A single-level directory has significant limitations, however, when the number of files increases or when the system has more than one user.
- Since all files are in the same directory, they must have unique names. If two users call their data file test.txt, then the unique-name rule is violated
- **2. Two-Level Directory**

- In the two-level directory structure, each user has his own user file directory (UFD).
- The UFDs have similar structures, but each lists only the files of a single user.
- When a user job starts or a user logs in, the system's master file directory (MFD)is searched.
- TheMFD is indexed by user name or account number, and each entry points to the UFD for that user

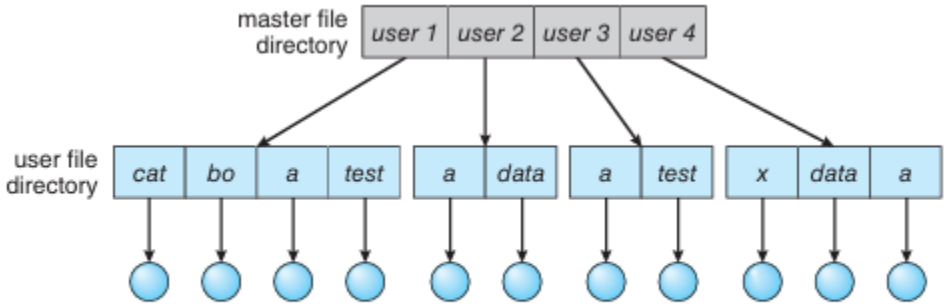


Figure 13.8 Two-level directory structure.

-

- **3.Tree-Structured Directories**
- extend the directory structure to a tree of arbitrary height
- This generalization allows users to create their own subdirectories and to organize their files accordingly
- The tree has a root directory, and every file in the system has a unique path name

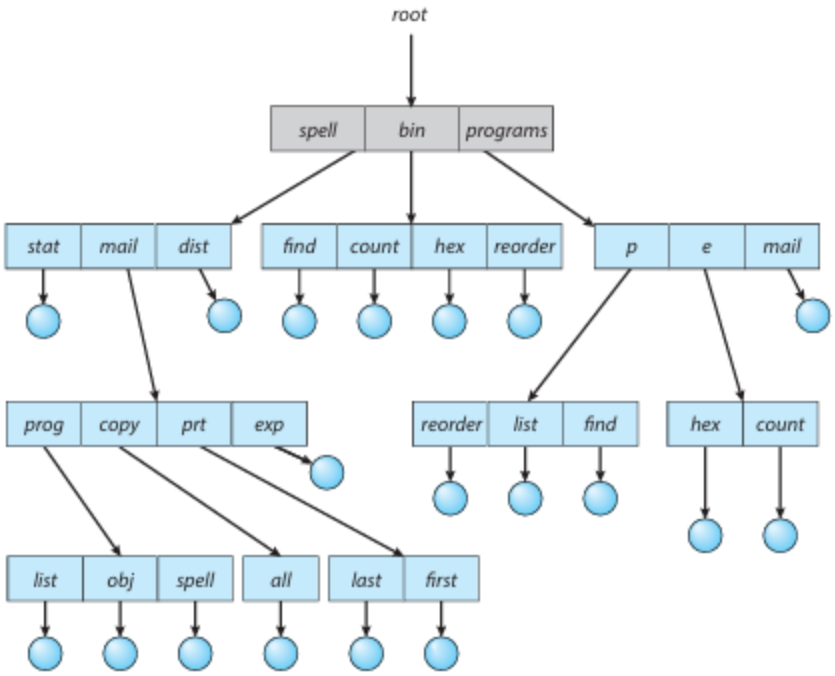


Figure 13.9 Tree-structured directory structure.

☐

☐ **4. Acyclic-Graph Directories**

☐ Consider two programmers who are working on a joint project.

☐ The files associated with that project can be stored in a subdirectory, separating them from other projects and files of the two programmers

☐ The common subdirectory should be shared. A shared directory or file exists in the file system in two (or more) places at once

☐ A tree structure prohibits the sharing of files or directories. An acyclic graph—that is, a graph with no cycles—allows directories to share sub directories and files

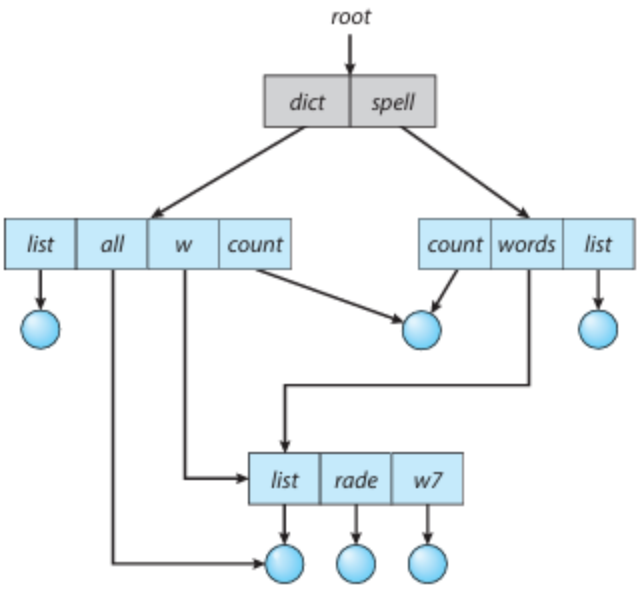


Figure 13.10 Acyclic-graph directory structure.

5. General Graph Directory

- A serious problem with using an acyclic-graph structure is ensuring that there are no cycles
- If we start with a two-level directory and allow users to create subdirectories, a tree-structured directory results
- However, when we add links, the tree structure is destroyed, resulting in a simple graph structure

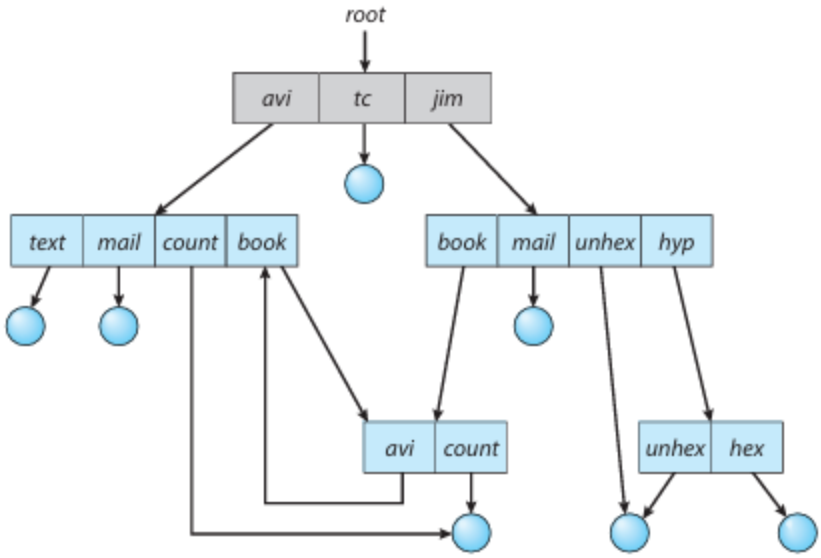


Figure 13.11 General graph directory.

I/O Subsystems

- **I/O hardware**
- Computers operate a great many kinds of devices.
  - storage devices (disks, tapes),
  - transmission devices (network connections, Bluetooth)
  - human-interface devices (screen, keyboard, mouse, audio in and out).
  - Specialized devices
    - In the steering of a jet, a human gives input to the flight computer via a joystick and foot pedals, and the computer sends output commands that cause motors to move rudders and flaps and fuels to the engines.
- We need only a few concepts to understand how the devices are attached and how the software can control the hardware.
- A device communicates with a computer system by sending signals over a cable or even through the air.

- The device communicates with the machine via a connection point, or port
- If devices share a common set of wires, the connection is called a bus.

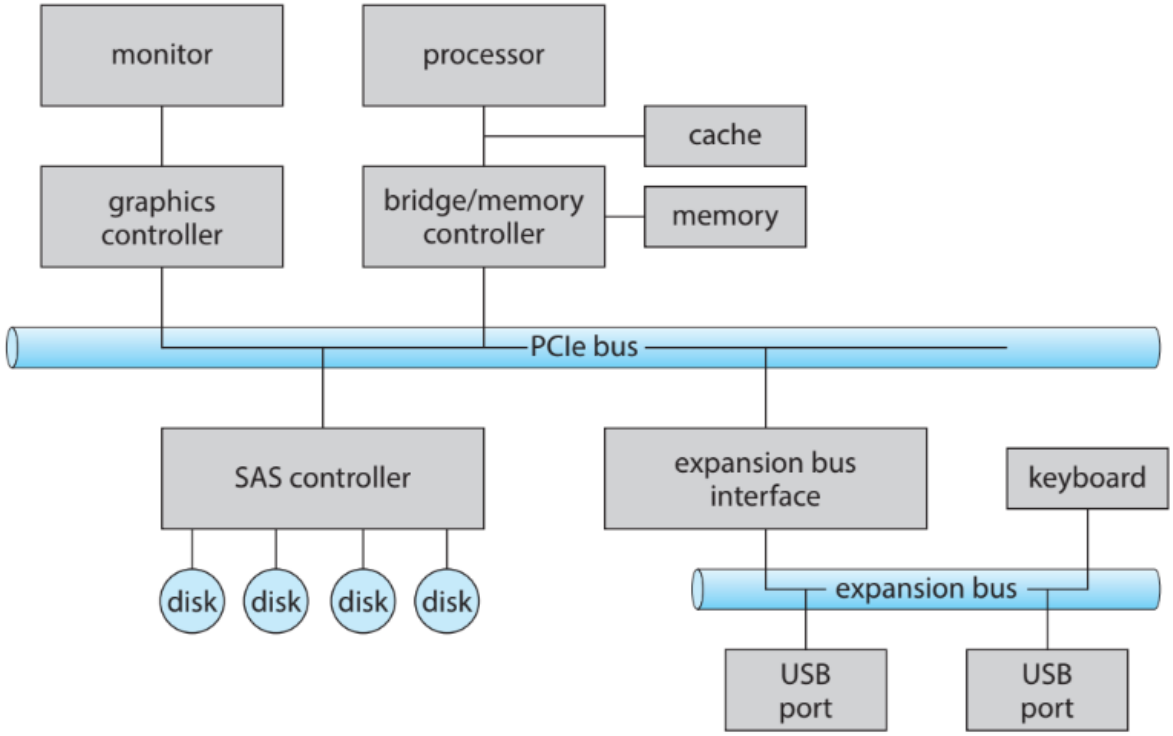


Figure 12.1 A typical PC bus structure.

- In the figure, a **PCIe bus** (the common PC system bus) connects the processor–memory subsystem to fast devices
- an **expansion bus** connects relatively slow devices, such as the keyboard and serial and USB ports.
- In the lower-left portion of the figure, four disks are connected together on a serial-attached **SCSI (SAS) bus** plugged into an **SAS controller**.
- PCIe is a flexible bus that sends data over one or more “lanes.”

- ☐ A lane is composed of two signaling pairs, one pair for receiving data and the other for transmitting.
- ☐ Each lane is therefore composed of four wires, and each lane is used as a full-duplex byte stream, transporting data packets in an eight-bit byte format simultaneously in both directions.
- ☐ A **controller** is a collection of electronics that can operate a port, a bus, or a device.
- ☐ A **serial-port controller** is a simple device controller. It is a single chip (or portion of a chip) in the computer that controls the signals on the wires of a serial port.
- ☐ By contrast, a **fiber channel (FC) bus controller** is not simple. Because the FC protocol is complex and used in data centers rather than on PCs, the FC bus controller is often implemented as a separate circuit board —or a host bus adapter (HBA)— that connects to a bus in the computer.
- ☐ It typically contains a processor, microcode, and some private memory to enable it to process the FC protocol messages.
- ☐ **Memory-Mapped I/O**
- ☐ How does the processor give commands and data to a controller to accomplish an I/O transfer?
- ☐ the device can support memory-mapped I/O
- ☐ the device-control registers are mapped into the address space of the processor
- ☐ The CPU executes I/O requests using the standard data transfer instructions to read and write the device-control registers at their mapped locations in physical memory
- ☐ Figure 12.2 shows the usual I/O port addresses for PCs

I/O address range (hexadecimal)	device
000–00F	DMA controller
020–021	interrupt controller
040–043	timer
200–20F	game controller
2F8–2FF	serial port (secondary)
320–32F	hard-disk controller
378–37F	parallel port
3D0–3DF	graphics controller
3F0–3F7	diskette-drive controller
3F8–3FF	serial port (primary)

Figure 12.2 Device I/O port locations on PCs (partial).

Polling

- The complete protocol for interaction between the host and a controller can be intricate, but the basic handshaking notion is simple
- The controller indicates its state through the busy bit in the status register

1. The host repeatedly reads the `busy` bit until that bit becomes clear.
2. The host sets the `write` bit in the `command` register and writes a byte into the `data-out` register.
3. The host sets the `command-ready` bit.
4. When the controller notices that the `command-ready` bit is set, it sets the `busy` bit.
5. The controller reads the command register and sees the `write` command. It reads the `data-out` register to get the byte and does the I/O to the device.
6. The controller clears the `command-ready` bit, clears the `error` bit in the status register to indicate that the device I/O succeeded, and clears the `busy` bit to indicate that it is finished.

This loop is repeated for each byte.

- ☐ In step 1, the host is busy-waiting or polling
- ☐ **Interrupts**
- ☐ The CPU hardware has a wire called the interrupt-request line that the CPU senses after executing every instruction
- ☐ When the CPU detects that a controller has asserted a signal on the interrupt-request line, the CPU performs a state save and jumps to the interrupt-handler routine at a fixed address in memory
- ☐ The interrupt handler determines the cause of the interrupt, performs the necessary processing, performs a state restore, and executes a return from interrupt instruction to return the CPU to the execution state prior to the interrupt

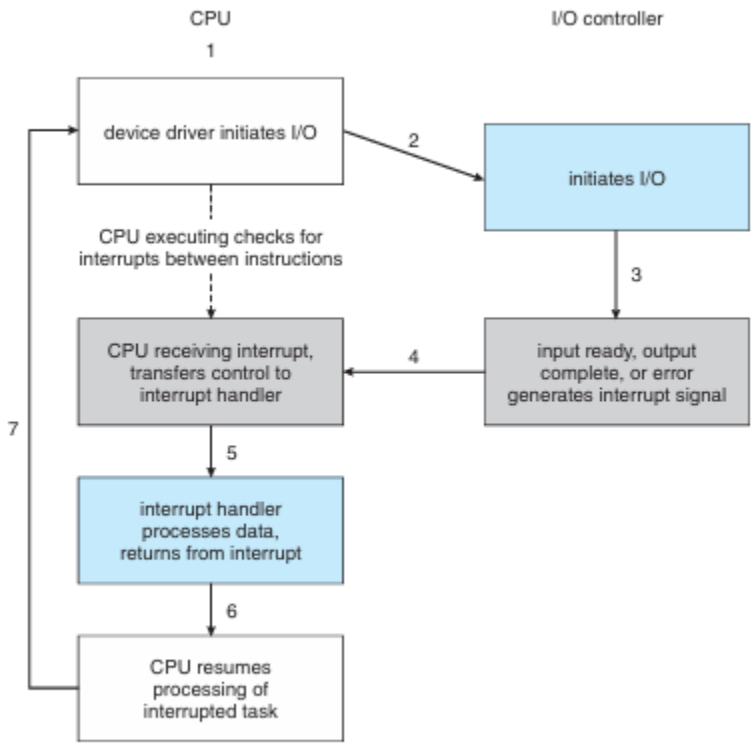


Figure 12.3 Interrupt-driven I/O cycle.

- ☐
- ☐ **Direct Memory Access**

- ☐ For a device that does large transfers, such as a disk drive, it seems wasteful to use an expensive general-purpose processor to watch status bits and to feed data into a controller register one byte at a time—a process termed **programmedI/O(PIO)**
- ☐ Computers avoid burdening the main CPU with PIO by offloading some of this work to a special-purpose processor called a **direct memory-access (DMA) controller**
- ☐ To initiate a DMA transfer, the host writes a DMA command block into memory.
- ☐ This block contains a pointer to the source of a transfer, a pointer to the destination of the transfer, and a count of the number of bytes to be transferred
- ☐ This scatter–gather method allows multiple transfers to be executed via a single DMAcommand
- ☐ The CPU writes the address of this command block to the DMA controller, then goes on with other work

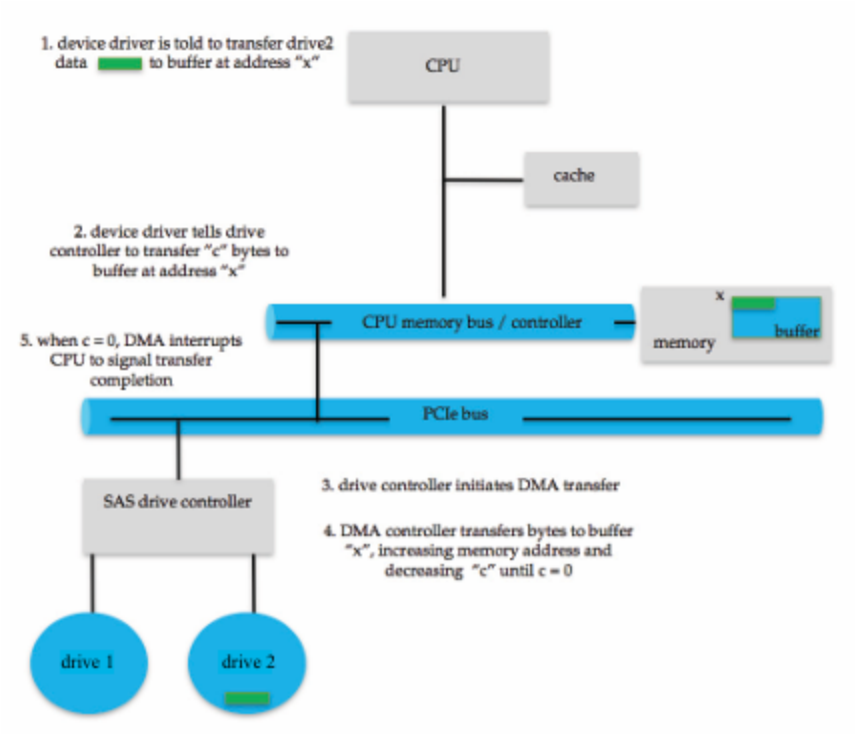


Figure 12.6 Steps in a DMA transfer.

☐

☐