Ch:5 File system and Memory management (24 M)

File Concept:

Computers can store information on several different storage media, such as magnetic disks, magnetic tapes, and optical disks. So that the computer system will be convenient to use, the operating system provides a uniform logical view of information storage. The operating system abstracts from the physical properties of its storage devices to define a logical storage unit (the file). Files are mapped, by the operating system, onto physical devices. These storage devices are usually nonvolatile, so the contents are persistent through power failures and system reboots.

A file is a named collection of related information that is recorded on secondary storage. From a user's perspective, 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.

File Attributes:

A file is named, for the convenience of its human users, and is referred to by its name. A name is usually a string of characters, such as *examp1e.c.* Some systems differentiate between upper- and lowercase characters in names, whereas other systems consider the two cases to be equivalent. When a file is named, it becomes independent of the process, the user, and even the system that created it. For instance, one user might create the file *example.^,* whereas another user might edit that file by specifying its name. The file's owner might write the file to a floppy disk, send it in an e-mail, or copy it across a network, and it could still be called *examp1e.c* on the destination system.

A file has certain other attributes, which 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.

Identifier: 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 those systems that support different types.

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.

Time, date, 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:

A file is an **abstract data type.** To define a file properly, we need to consider the operations that can be performed on files. The operating system can provide system calls to create, write, read, reposition, delete, and truncate files. Let us also consider what the operating system must do for each of the six basic file operations. It should then be easy to see how similar operations, such as renaming a file, would be implemented.

Creating a file: Two steps are necessary to create a file. First, space in the file system must be found for the file. We shall discuss how to allocate space for the file in Chapter 12. Second, an entry for the new file must be made in the directory. The directory entry records the name of the file and the location in the file system, and possibly other information.

Writing a file: To write a file, we make a system call specifying both the name of the file and the information to be written to the file. Given the name of the file, the system searches the directory to find the location of the 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: To read from a file, we use a system call that specifies the name of the file and where (in memory) the next block of the file should be put. Again, the directory is searched for the associated directory entry, and 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. A given process is usually only reading or writing a given file, and the current operation location is kept as a per-process **current-file-position pointer.** Both the read and write operations use this same pointer, saving space and reducing the system complexity.

Repositioning within a file: The directory is searched for the appropriate entry, and the current-file-position is set to a given value. Repositioning within a file does not need to involve any actual I/O. This file operation is also known as a file *seek.*

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, and erase the directory entry.

Truncating a file: The user may want to erase the contents of a file but keep its attributes. 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-but lets the file be reset to length zero and its file space released.

File Types:

When we design a file system, indeed the entire operating system, we always consider whether the operating system should recognize and support file types. If an operating system recognizes the type of a file, it can then operate on the file in reasonable ways. For example, a common mistake occurs when a user tries to print the binary-object form of a program. This attempt normally produces garbage, but can be prevented if the operating system has been told that the file is a binary-object program.

A common technique for implementing file types is to include the type as part of the file name. The name is split into two parts-a name and an extension, usually separated by a period character . In this way, the user and the operating system can tell from the name alone what the type of a file is. For example, in MS-DOS, a name can consist of up to eight characters followed by a period and terminated by an extension of up to three characters. The system uses the extension to indicate the type of the file and the type of operations that can be done on that file. For instance, only a file with a .corn, .exe, or .bat extension can be executed. The .corn and .exe files are two forms of binary executable files, whereas a *.bat* file is a **batch file** containing, in ASCII format, commands to the operating system. MS-DOS recognizes only a few extensions, but application programs also use extensions to indicate file types in which they are interested. For example, assemblers expect source files to have an *.asm* extension, and the Wordperfect word processor expects its file to end with a *.wp* extension. These extensions are not required, so a user may specify a file without the extension (to save typing), and the application will look for a file with the given name and the extension it expects. Because these extensions are not supported by the operating system, they can be considered as "hints" to applications that operate on them.

Access Methods:

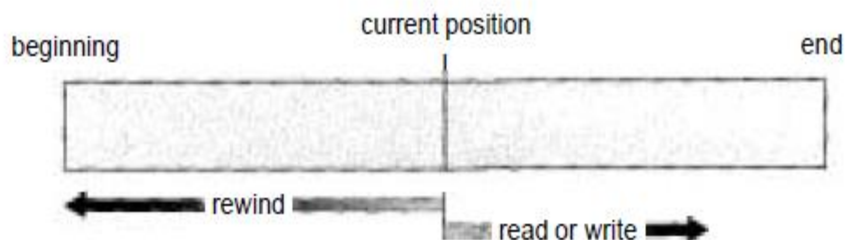
Files store information. When it is used, this information must be accessed and read into computer memory. The information in the file can be accessed in several ways. Some systems provide only one access method for files. Other systems, such as those of IBM, support many access methods, and choosing the right one for a particular application is a major design problem.

1) Sequential Access
2) Direct Access

Sequential Access

The simplest access method is **sequential access.** Information in the file is processed in order, one record after the other. This mode of access is by far the most common; for example, editors and compilers usually access files in this fashion.

The bulk of the operations on a file is reads and writes. A read operation reads the next portion of the file and automatically advances a file pointer, which tracks the I/O location. Similarly, a write 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 and, on some systems, a program may be able to skip forward or backward n records, for some integer n-perhaps only for $n = 1$. Sequential access is based on a tape model of a file, and works as well on sequential-access devices as it does on random-access ones.



Sequential-access file.

Direct Access:

Another method is **direct access** (or **relative access).** A file is made up of fixed length **logical records** that allow programs to read and write records rapidly in no particular order. The direct-access method is based on a disk model of a file, since disks allow random access to any file block. For direct access, the file is viewed as a numbered sequence of blocks or records. A direct-access file allows arbitrary blocks to be read or written. 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. When a query concerning a particular subject arrives, we compute which block contains the answer, and then read that block directly to provide the desired information.

As a simple example, on an airline-reservation system, we might store all the information about a particular flight (for example, flight 713) in the block identified by the flight number. Thus, the number of available seats for flight 713 is stored in block 713 of the reservation file. To store information about a larger set, such as people, we might compute a hash function on the people's
names, or search a small in-memory index to determine a block to read and search.

Directory Structure:

The file systems of computers can be extensive. Some systems store millions of files on terabytes of disk. To manage all these data, we need to organize them. This organization is usually done in two parts. **First**, disks are split into one or more *partitions,* also known as *minidisks* in the *IBM* world or *volumes* in the *PC* and Macintosh arenas. **Second**, each partition contains information about files within it. This information is kept in entries in a **device directory** or **volume table of contents.**

When considering a particular directory structure, we need to keep in mind the **operations** that are to be performed on a directory:

1)Search for a file: We need to be able to search a directory structure to find the entry for a particular file. Since files have symbolic names and similar names may indicate a relationship between files, we may want to be able to find all files whose names match a particular pattern.

2)Create a file: New files need to be created and added to the directory.
3)Delete a file: When a file is no longer needed, we want to remove it from the directory.
4)List a directory: We need to be able to list the files in a directory, and the contents of the directory entry for each file in the list.
5)Rename a file: Because the name of a file represents its contents to its users, the name must be changeable when the contents or use of the file changes. Renaming a file may also allow its position within the directory structure to be changed.
6)Traverse the file system: We may wish to access every directory, and every file within a directory structure. For reliability, it is a good idea to save the contents and structure of the entire file system at regular intervals. This saving often consists of copying all files to magnetic tape. This technique
provides a backup copy in case of system failure or if the file is simply no longer in use. In this case, the file can be copied to tape, and the disk space of that file released for reuse by another file.

In this sections we describe the most common schemes for defining the logical structure of a directory.

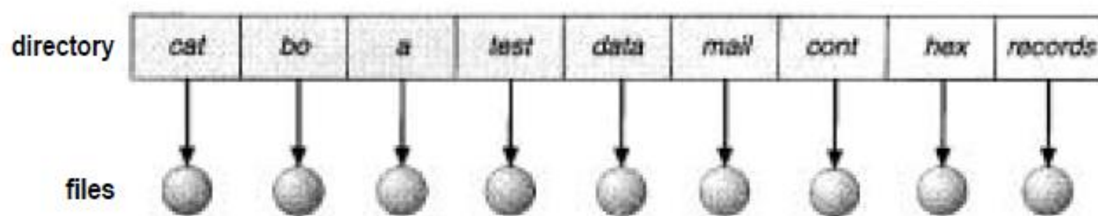
1) **Single-Level Directory**
2) **Two-Level Directory**
3) **Tree directory structure**

Single-Level Directory:

The simplest directory structure is the single-level directory. All files are contained in the same directory, which is easy to support and understand .

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,* then the unique-name rule is violated. For example, in one programming class, 23 students called the program for their second assignment *prog2;* another 11 called it *assign2*. Although file names are generally selected to reflect the content of the file, they are often limited in length. The MS-DOS operating system allows only 11-character file names; UNIX allows 255 characters.

Even a single user on a single-level directory may find it difficult ***to remember the names of all the files, as the number of files increases.*** It is not uncommon for a user to have hundreds of files on one computer system and an equal number of additional files on another system. In such an environment, keeping track of so many files is a daunting task.



Single-level directory.

Two-Level Directory:

A single-level directory often leads to confusion of file names between different users. The standard solution is to create a *separate* directory for each user.

In the two-level directory structure, each user has her own **user file directory (UFD).** Each UFD has a similar structure, but 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. The MFD is indexed by user name or account number, and each entry points to the UFD for that user

When a user refers to a particular file, only his own UFD is searched. Thus, different users may have files with the same name, as long as all the file names within each UFD are unique.

To create a file for a user, the operating system searches only that user's UFD to ascertain whether another file of that name exists. To delete a file, the operating system confines its search to the local UFD; thus, it cannot accidentally delete another user's file that has the same name.

Although the two-level directory structure solves the name-collision problem, it still has disadvantages. This structure effectively isolates one user from another. This isolation is an advantage when the users are completely independent.

Tree directory structure

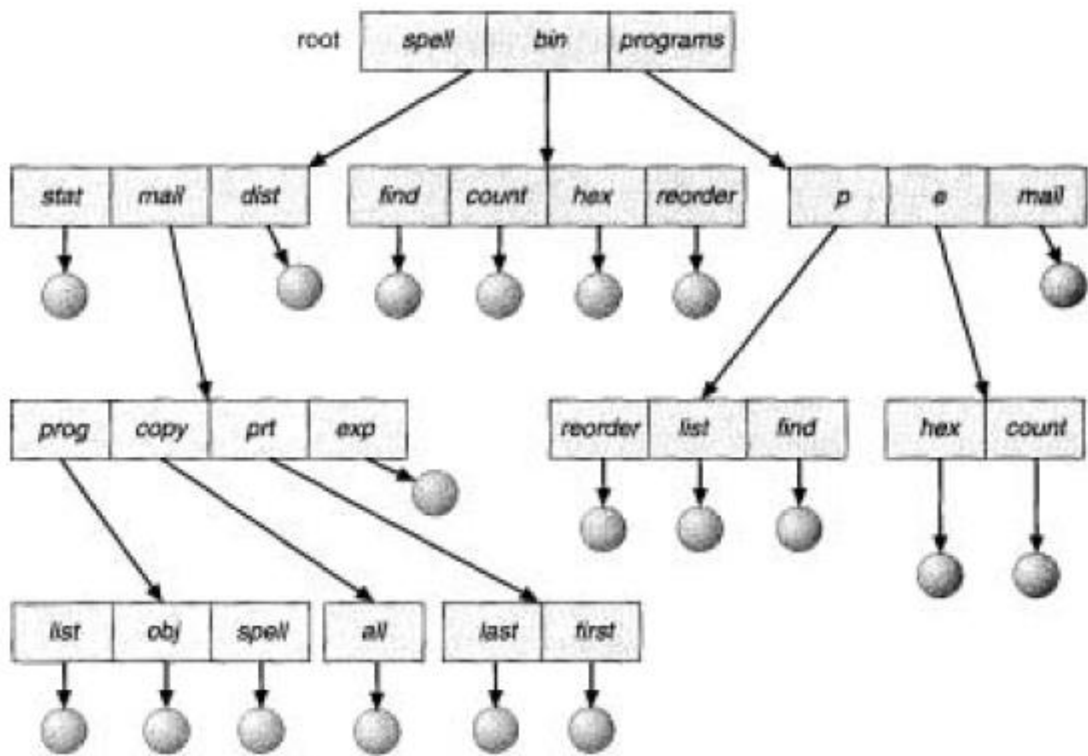
Path names can be of two types: absolicte path names or relative path names.

An absolute path name begins at the root and follows a path down to the specified file, giving the directory names on the path. A relative path name defines a path from the current directory. For example, in the tree-structured file system of Figure 11.8, if the current directory is root/spell/mail, then the relative path name prtlfirst refers to the same file as does the absolute path name root/spell/rnail/prtlfivst.

Allowing the user to define his own subdirectories permits him to impose a structure on his files. This structure might result in separate directories for files associated with different topics (for example, a subdirectory was created to hold the text of this book) or different forms of information (for example, the directory programs may contain source programs; the directory bin may store all the binaries).

An interesting policy decision in a tree-structured directory structure is how to handle the deletion of a directory. If a directory is empty, its entry in its containing directory can simply be deleted. However, suppose the directory to be deleted is not empty, but contains several files or subdirectories: One of two approaches can be taken. Some systems, such as MS-DOS, will not delete a directory unless it is empty. Thus, to delete a directory, the user must first delete all the files in that directory. If any subdirectories exist, this procedure must be applied recursively to them, so that they can be deleted also. This approach may result in a substantial amount of work.

An alternative approach, such as that taken by the UNIX r m command, is to provide the option that, when a request is made to delete a directory, all that directory's files and subdirectories are also to be deleted. Either approach is fairly easy to implement; the choice is one of policy. The latter policy is more convenient, but more dangerous, because an entire directory structure may be removed with one command. If that command were issued in error, a large number of files and directories would need to be restored from backup tapes.



root
spell
bin
programs
stat
mail
dist
find
count
hex
reorder
p
e
mail
prog
copy
prt
exp
reorder
list
find
hex
count
list
obj
spell
all
last
first

Allocation Methods:

The direct-access nature of disks allows us flexibility in the implementation of files. In almost every case, many files will be stored on the same disk. The main problem is how to allocate space to these files so that disk space is utilized effectively and files can be accessed quickly. Three major methods of allocating disk space are in wide use: contiguous, linked, and indexed. Each method has advantages and disadvantages.

1) **Contiguous Allocation**
2) **Linked Allocation**
3) **Indexed Allocation**

Contiguous Allocation:

The **contiguous-allocation** method requires each file to occupy a set of contiguous blocks on the disk. Disk addresses define a linear ordering on the disk. With this ordering, assuming that only one job is accessing the disk, accessing block b + 1 after block b normally requires no head movement.

Contiguous allocation of a file is defined by the disk address and length (in block units) of the first block. If the file is *n* blocks long and starts at location b, then it occupies blocks b, b + 1, b + 2, ..., b + *n* - 1. The directory entry for each file indicates the address of the starting block and the length of the area allocated for this file

Directory has structure like this.

directory

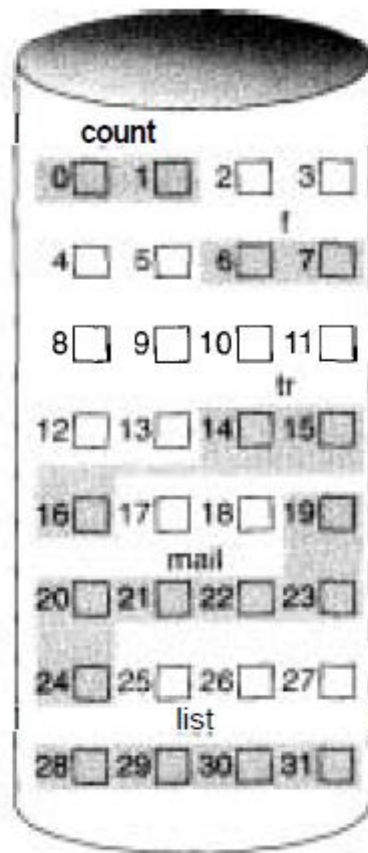
file	start	length
count	0	2
tr	14	3
mail	19	6
list	28	4
f	6	2

Advantages:

1) Thus, both sequential and direct access can be supported by contiguous allocation.
2) Accessing a file that has been allocated contiguously is easy

Disadvantages:

1) finding space for a new file
2) **dynamic storage-allocation**
3) **external fragmentation**
4) Size of file must be estimated in advance.



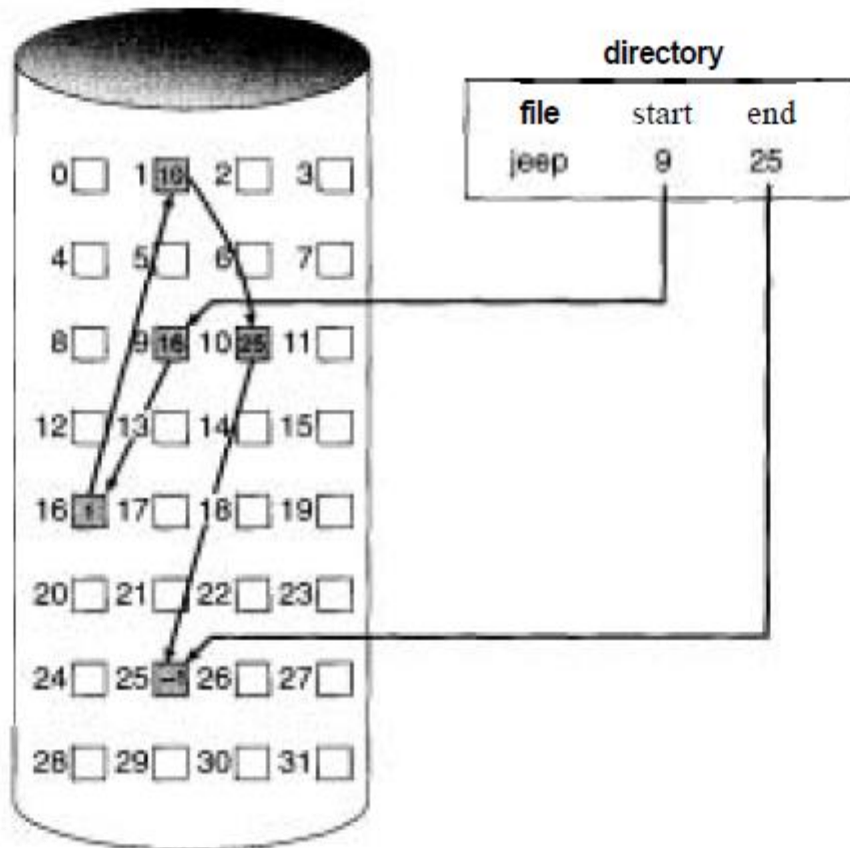
directory

file	start	length
count	0	2
tr	14	3
mail	19	6
list	28	4
f	6	2

Contiguous allocation of disk space.

Linked Allocation:

Linked allocation solves all problems of contiguous allocation. With linked 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. For example, a file of five blocks might start at block 9, continue at block 16, then block 1, block 10, and finally block 25. Each block contains a pointer to the next block. These pointers are not made available to the user. Thus, if each block is 512 bytes, and a disk address (the pointer) requires 4 bytes, then the user sees blocks of 508 bytes.



Linked allocation of disk space.

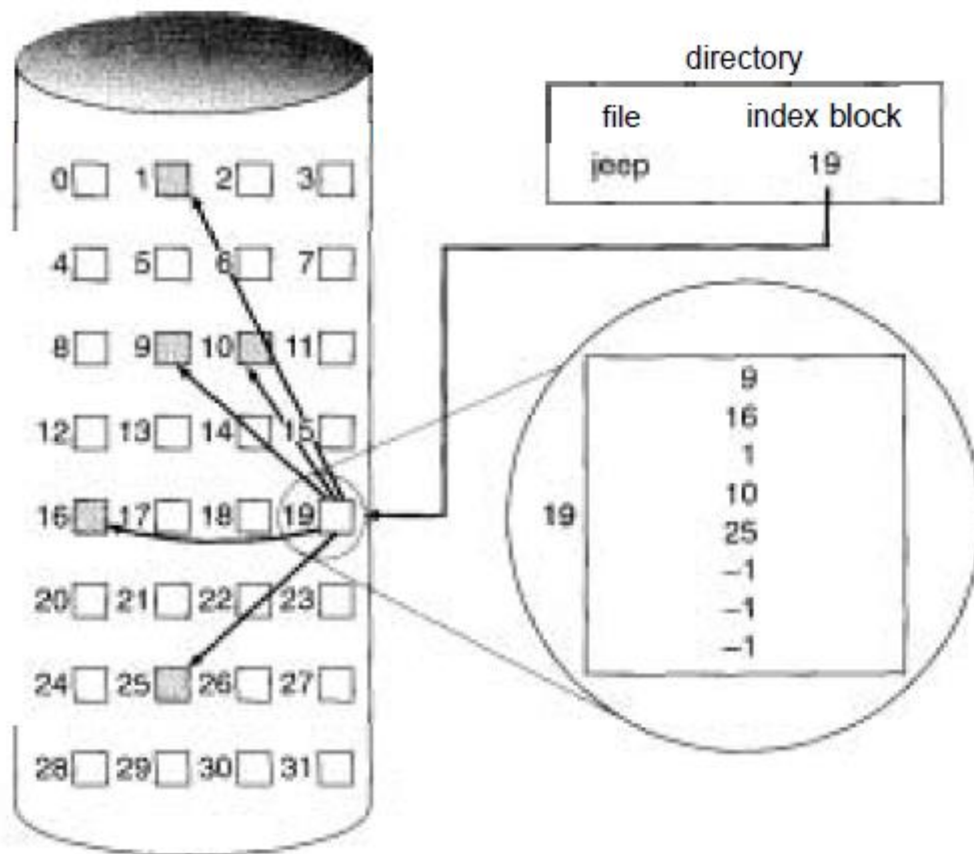
Advantages:
1)The size of a file does not need to be declared when that file is created. A file can continue to grow as long as free blocks are available.
2)Consequently, it is never necessary to compact disk space
3) No external fragmentation

Disadvantages:
1) it can be used effectively only for sequential-access files.
2) Space required for the pointers
3) Reliability

Indexed Allocation:

Linked allocation solves the external-fragmentation and size-declaration problems of contiguous allocation. However, in the absence of a FAT, linked allocation cannot support efficient direct access, since the pointers to the blocks are scattered with the blocks themselves all over the disk and need to be retrieved in order, **Indexed allocation** solves this problem by bringing all the pointers 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. The directory contains the address of the index block .To read the ith block, use the pointer in the ith index-block entry to find and read the desired block.



Indexed allocation of disk space.

Advantages:

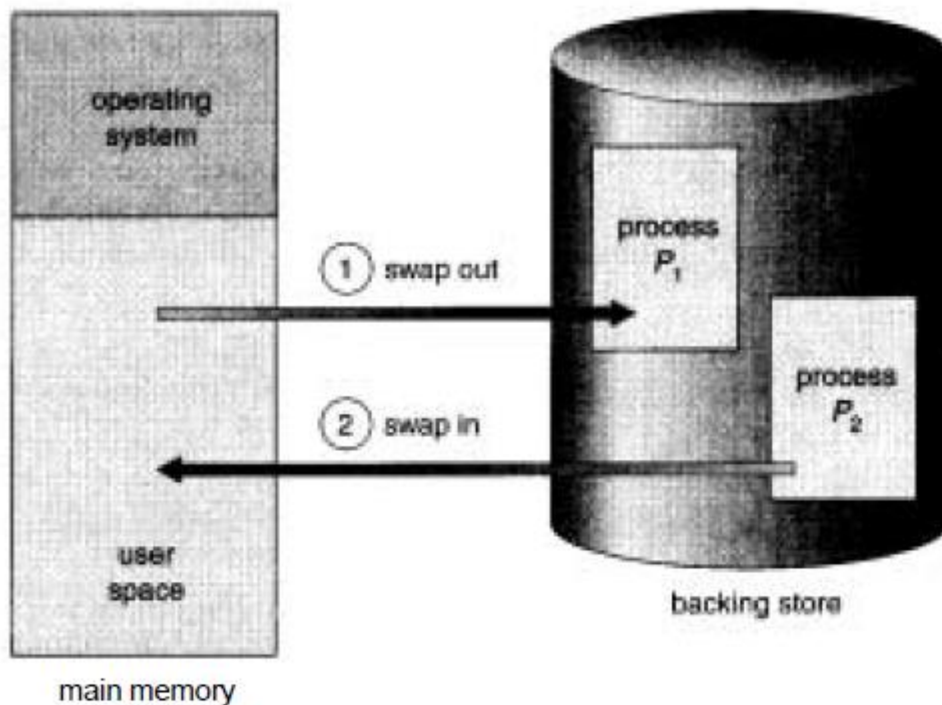
Indexed allocation supports direct access, without suffering from external fragmentation, because any free block on the disk may satisfy a request for more space.

Disadvantages:

Indexed allocation does suffer from wasted space

Swapping:

A process needs to 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. For example, assume a multiprogramming environment with a round-robin CPU-scheduling algorithm. When a quantum expires, the memory manager will start to swap out the process that just finished, and to swap in another process to the memory space that has been
freed (Figure 9.4). In the meantime, the CPU scheduler will allocate a time slice to some other process in memory. When each process finishes its quantum, it will be swapped with another process. Ideally, the memory manager can swap processes fast enough that some processes will be in memory, ready to execute, when the CPU scheduler wants to reschedule the CPU. The quantum must also be sufficiently large that reasonable amounts of computing are done between swaps.



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

A variant of this swapping policy is used for priority-based scheduling algorithms. If a higher-priority process arrives and wants service, the memory manager can swap out the lower-priority process so that it can load and execute the higher-priority process. When the higher-priority process finishes, the lower-priority process can be swapped back in and continued. This variant of swapping is sometimes called **roll out, roll in.**

Normally a process that is swapped out will be swapped back into the same memory space that it occupied previously. This restriction is dictated by the method of address binding. If binding is done at assembly or load time, then the process cannot be moved to different locations. If execution-time binding is being used, then a process can be swapped into a different memory space, because the physical addresses are computed during execution time.

Swapping requires 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 not, and 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 as normal and transfers control to the selected process.

Protection:

When information is kept in a computer system, we want to keep it safe from physical damage (reliability) and improper access (protection).

Reliability is generally provided by duplicate copies of files. Many computers have systems programs that automatically (or through computer-operator intervention) copy disk files to tape at regular intervals (once per day or week or month) to maintain a copy should a file system be accidentally destroyed. File systems can be damaged by hardware problems (such as errors in reading or writing), power surges or failures, head crashes, dirt, temperature extremes, and vandalism. Files may be deleted accidentally. Bugs in the file-system software can also cause file contents to be lost.

Protection can be provided in many ways. For a small single-user system, we might provide protection by physically removing the floppy disks and locking them in a desk drawer or file cabinet. In a multiuser system, however, other mechanisms are needed.

Types of Access:

The need to protect files is a direct result of the ability to access files. Systems that do not permit access to the files of other users do not need protection. Thus, we could provide complete protection by prohibiting access. Alternatively, we could provide free access with no protection. Both approaches are too extreme for general use. What is needed is **controlled access.**

Protection mechanisms provide controlled access by limiting the types of file access that can be made. Access is permitted or denied depending on several factors, one of which is the type of access requested. Several different types of operations may be controlled:

Read: Read from the file.
Write: Write or rewrite the file.
Execute: Load the file into memory and execute it.
Append: Write new information at the end of the file.
Delete: Delete the file and free its space for possible reuse.
List: List the name and attributes of the file.

Difficulty in this approach is that only static control of files (i.e. when you have set write permission to any file everyone can write the file i.e. all types of user. You cant remove write permission from any type of user)

Access Control:

The most common approach to the protection problem is to make access dependent on the identity of the user. Various users may need different types of access to a file or directory. The most general scheme to implement identity-dependent access is to associate with each file and directory an access-control list **(ACL)** specifying the user name and the types of access allowed for each user. When a user requests access to a particular file, the operating system checks the access list associated with that file. If that user is listed for the requested access, the access is allowed. Otherwise, a protection violation occurs, and the user job is denied access to the file.

This approach has the advantage of enabling complex access methodologies. The main problem with access lists is their length. If we want to allow everyone to read a file, we must list all users with read access. This technique has two undesirable consequences:

1) Constructing such a list may be a tedious and unrewarding task, especially if we do not know in advance the list of users in the system.
2) The directory entry, previously of fixed size, now needs to be of variable size, resulting in more complicated space management.

These problems can be resolved by use of a condensed version of the access list. To condense the length of the access control list, many systems recognize three classifications of users in connection with each file:
Owner: The user who created the file is the owner.
Group: A set of users who are sharing the file and need similar access is a group, or work group.
Universe: All other users in the system constitute the universe.

The most common recent approach is to combine access control lists with the more general (and easier to implement) owner, group, and universe access control scheme that was described above. For example, Solaris 2.6 and beyond uses the three categories of access by default, but allows access control lists to be added to specific files and directories when more fine-grained access control is desired.

Difficulty in this approach is that:

1) Types of users should be known in advance
2) Entry in directory list is dynamic (difficult to handle)

Basic Memory Management –Partitioning, Fixed & Variable.

Free Space management techniques
 -Bitmap
 -Linked List.

Since disk space is limited, we need to reuse the space from deleted files for new files, if possible. (Write-once optical disks only allow one write to any given sector, and thus such reuse is not physically possible.) To keep track of free disk space, the system maintains a **free-space list.** The free-space list records all free disk blocks-those not allocated to some file or directory. To create a file, we search the free-space list for the required amount of space, and allocate that space to the new file. This space is then removed from the free-space list. When a file is deleted, its disk space is added to the free-space list. The free-space list, despite its name, might not be implemented as a list, as we shall discuss.

Bit Vector

Frequently, the free-space list is implemented as a bit **map** 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 freespace bit map would be

001111001111110001100000011100000 ...

The main advantage of this approach is its relatively simplicity and efficiency in finding the first free block, or *n* consecutive free blocks on the disk. To find the first free block, the Macintosh operating system checks sequentially each word in the bit map to see whether that value is not 0, since a 0-valued word has all 0 bits and represents a set of allocated blocks. The first non-0 word is scanned for the first 1 bit, which is the location of the first free block

Again, we see hardware features driving software functionality. ***Unfortunately, bit vectors are inefficient unless the entire vector is kept in main memory*** (and is written to disk occasionally for recovery needs). Keeping it in main memory is possible for smaller disks, such as on microcomputers, but not for larger ones. A 1.3-GB disk with 512-byte blocks would need a bit map of over 332 KB to track its free blocks. Clustering the blocks in groups of four reduces this number to over 83 KB per disk.

Linked List:

Another approach to free-space management is to link together all the free disk blocks, keeping a pointer to the first free block in a special location on the disk and caching it in memory. This first block contains a pointer to the next free disk block, and so on. In our example , we would keep a pointer to block 2, as the first free block. Block 2 would contain a pointer to block **3,** which would point to block 4, which would point to block 5, which would point to block 8, and so on (Figure). However, ***this scheme is not efficient; to traverse the list, we must read each block, which requires substantial I/0 time***. Fortunately, traversing the free list is not a frequent action. Usually, the operating system simply needs a free block so that it can allocate that block to a file, so the first block in the free list is used. The FAT method incorporates free-block accounting into the allocation data structure. No separate method is needed.

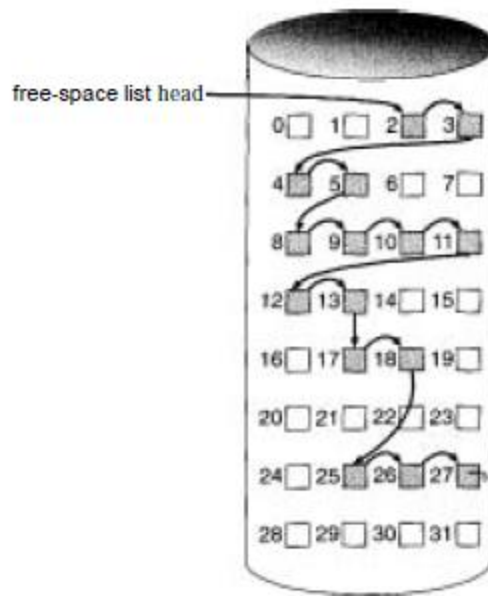


Figure 12.10 Linked free space list on disk.

Partitioning