

Module IV Virtual-Memory Management

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.

- In practice, most real processes do not need all their pages, or at least not all at once, for several reasons:
 1. Error handling code is not needed unless that specific error occurs, some of which are quite rare.
 2. Certain features of certain programs are rarely used.
- The ability to load only the portions of processes that are actually needed has several benefits:
 - Programs could be written for a much larger address space (virtual memory space) than physically exists on the computer.
 - Because each process is only using a fraction of their total address space, there is more memory left for other programs, improving CPU utilization and system throughput.
 - Less I/O is needed for swapping processes in and out of RAM, speeding things up.

The figure below shows the general layout of *virtual memory*, which can be much larger than physical memory:

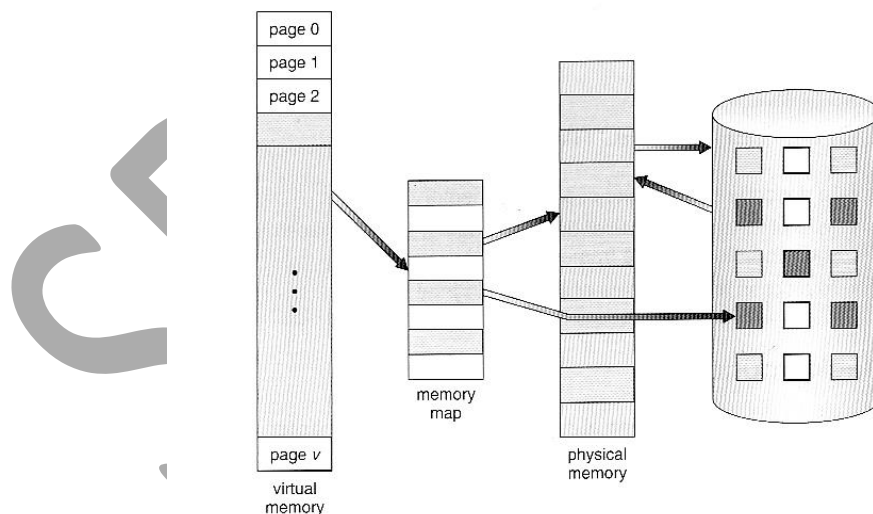


Figure 9.1 Diagram showing virtual memory that is larger than physical memory.

- The figure below shows *virtual address space*, which is the programmer's logical view of process memory storage. The actual physical layout is controlled by the process's page table.

Module IV – Virtual Memory Management & File System

- Note that the address space shown in Figure 9.2 is *sparse* - A great hole in the middle of the address space is never used, unless the stack and/or the heap grow to fill the hole.

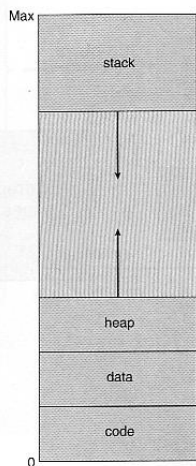


Figure 9.2 Virtual address space.

- Virtual memory also allows the sharing of files and memory by multiple processes, with several benefits:
- System libraries can be shared by mapping them into the virtual address space of more than one process.
- Processes can also share virtual memory by mapping the same block of memory to more than one process.
- Process pages can be shared during a fork() system call, eliminating the need to copy all of the pages of the original (parent) process.

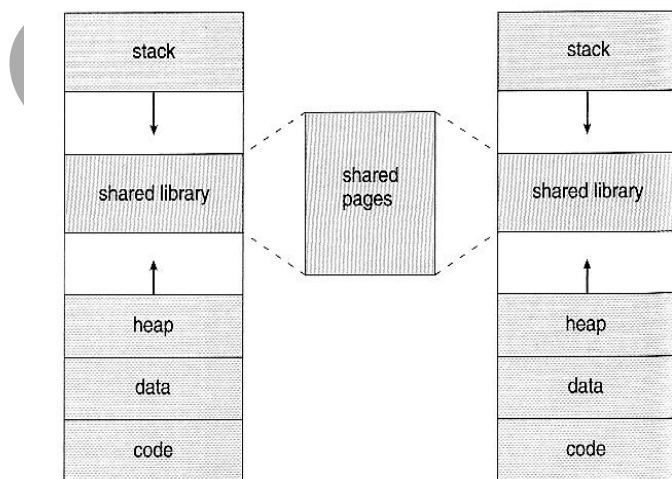


Figure 9.3 Shared library using virtual memory.

9.2 Demand Paging

- The basic idea behind **demand paging** is that when a process is swapped in, its pages are not swapped in all at once. Rather they are swapped in only when the process needs them. (on demand.) This is termed as **lazy swapper**, although a **pager** is a more accurate term.

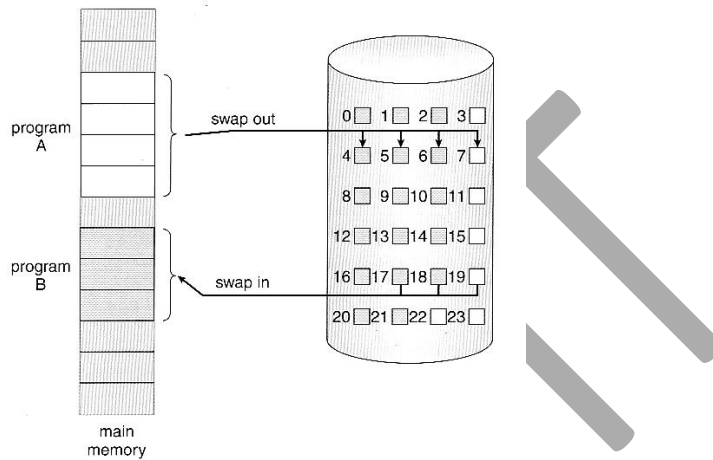


Figure 9.4 Transfer of a paged memory to contiguous disk space.

- The basic idea behind demand paging is that when a process is swapped in, the pager only loads into memory those pages that is needed presently.
- Pages that are not loaded into memory are marked as invalid in the page table, using the invalid bit. Pages loaded in memory are marked as valid.
- If the process only ever accesses pages that are loaded in memory (**memory resident** pages), then the process runs exactly as if all the pages were loaded in to memory.

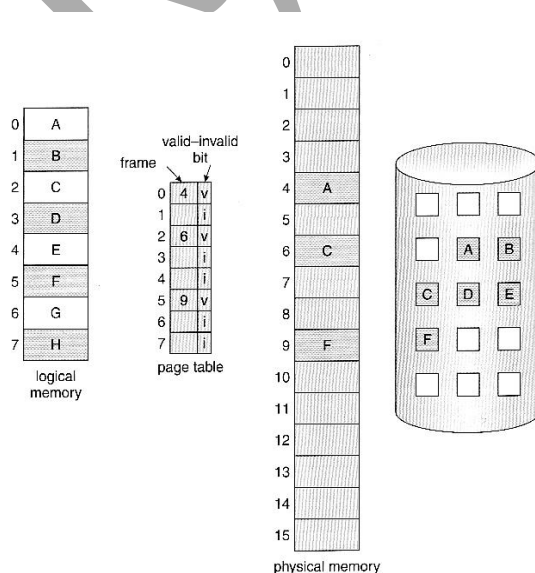


Figure 9.5 Page table when some pages are not in main memory.

Module IV – Virtual Memory Management & File System

- On the other hand, if a page is needed that was not originally loaded up, then a **page fault trap** is generated, which must be handled in a series of steps:
 1. The memory address requested is first checked, to make sure it was a valid memory request.
 2. If the reference is to an invalid page, the process is terminated. Otherwise, if the page is not present in memory, it must be paged in.
 3. A free frame is located, possibly from a free-frame list.
 4. A disk operation is scheduled to bring in the necessary page from disk.
 5. After the page is loaded to memory, the process's page table is updated with the new frame number, and the invalid bit is changed to indicate that this is now a valid page reference.
 6. The instruction that caused the page fault must now be restarted from the beginning.

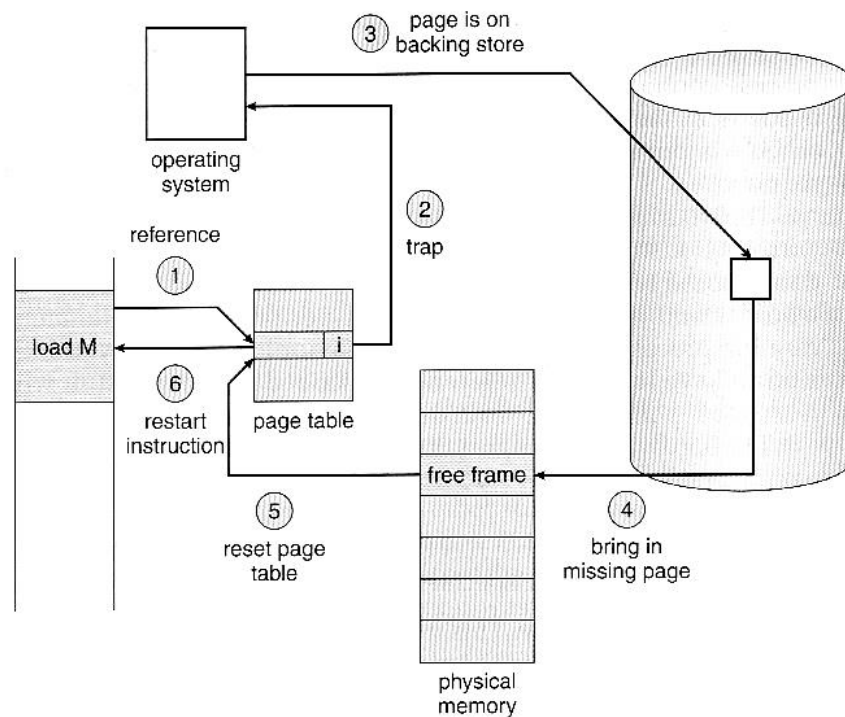


Figure 9.6 Steps in handling a page fault.

- In an extreme case, the program starts execution with zero pages in memory. Here NO pages are swapped in for a process until they are requested by page faults. This is known as **pure demand paging**.
- The **hardware necessary** to support demand paging is the same as for paging and swapping: A page table and secondary memory.

Performance of Demand Paging

- There is some slowdown and performance hit whenever a page fault occurs (as the required page is not available in memory) and the system has to go get it from memory.
- There are many steps that occur when servicing a page fault and some of the steps are optional or variable. But just for the sake of discussion, suppose that a normal memory access requires 200 nanoseconds, and that servicing a page fault takes 8 milliseconds. (8,000,000 nanoseconds, or 40,000 times a normal memory access.) With a **page fault rate** of p , (on a scale from 0 to 1), the effective access time is now:

$$\begin{aligned} \text{Effective access time} &= p * \text{time taken to access memory in page fault} + (1-p) * \text{time taken to} \\ &\quad \text{access memory} \\ &= p * 8000000 + (1 - p) * (200) \\ &= 200 + 7,999,800 * p \end{aligned}$$

Even if only one access in 1000 causes a page fault, the effective access time drops from 200 nanoseconds to 8.2 microseconds, a slowdown of a factor of 40 times. In order to keep the slowdown less than 10%, the page fault rate must be less than 0.0000025, or one in 399,990 accesses.

9.3 Copy-on-Write

- The idea behind a copy-on-write is that the pages of a parent process is shared by the child process, until one or the other of the processes changes the page. Only when a process changes any page content, that page is copied for the child.
- Only pages that can be modified need to be labeled as copy-on-write. Code segments can simply be shared.
- Some systems provide an alternative to the `fork()` system call called a **virtual memory fork, `vfork()`**. In this case the parent is suspended, and the child uses the parent's memory pages. This is very fast for process creation, but requires that the child not modify any of the shared memory pages

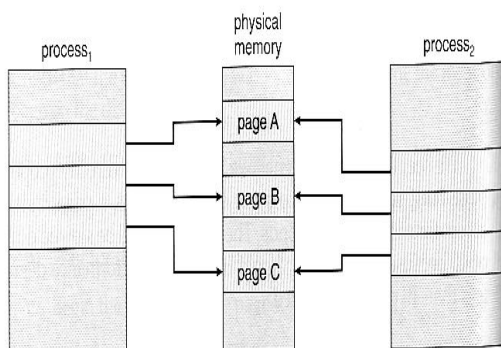


Figure 9.7 Before process 1 modifies page C.

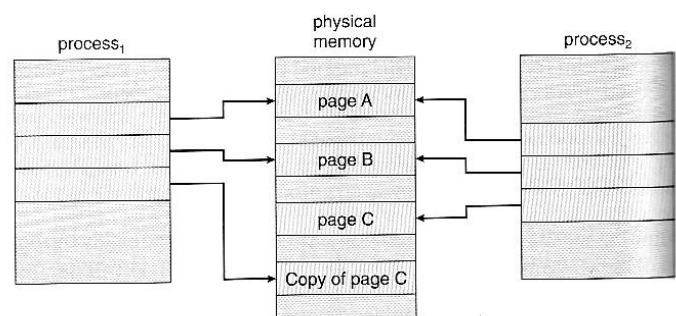


Figure 9.8 After process 1 modifies page C.

before performing the `exec()` system call.

9.4 Page Replacement

- In order to make the most use of virtual memory, we load several processes into memory at the same time. Since we only load the pages that are actually needed by each process at any given time, there are frames to load many more processes in memory.
- If some process suddenly decides to use more pages and there aren't any free frames available. Then there are several possible solutions to consider:
 1. Adjust the memory used by I/O buffering, etc., to free up some frames for user processes.
 2. Put the process requesting more pages into a wait queue until some free frames become available.
 3. Swap some process out of memory completely, freeing up its page frames.
 4. Find some page in memory that isn't being used right now, and swap that page only out to disk, freeing up a frame that can be allocated to the process requesting it. This is known as **page replacement**, and is the most common solution. There are many different algorithms for page replacement.

The previously discussed page-fault processing assumed that there would be free frames available on the free-frame list. Now the page-fault handling must be modified to free up a frame if necessary, as follows:

1. Find the location of the desired page on the disk.
2. Find a free frame:
 - a. If there is a free frame, use it.
 - b. If there is no free frame, use a page-replacement algorithm to select an existing frame to be replaced, known as the **victim frame**.
 - c. Write the victim frame to disk. Change all related page tables to indicate that this page is no longer in memory.
3. Read in the desired page and store it in the frame. Change the entries in page table.
4. Restart the process that was waiting for this page.

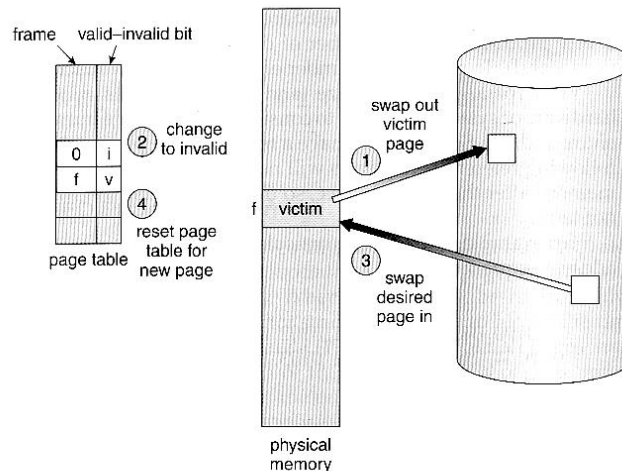


Figure 9.10 Page replacement.

- Note that step 2c adds an extra disk write to the page-fault handling, thus doubling the time required to process a page fault. This can be reduced by assigning a *modify bit*, or *dirty bit* to each page in memory, indicating whether or not it has been changed since it was last loaded in from disk. If the page is not modified the bit is not set. If the dirty bit has not been set, then the page is unchanged, and does not need to be written out to disk. Many page replacement strategies specifically look for pages that do not have their dirty bit set.
- There are two major requirements to implement a successful demand paging system.

A *frame-allocation algorithm* and a *page-replacement algorithm*. The former centers around how many frames are allocated to each process, and the latter deals with how to select a page for replacement when there are no free frames available.

- The overall goal in selecting and tuning these algorithms is to generate the fewest number of overall page faults. Because disk access is so slow relative to memory access, even slight improvements to these algorithms can yield large improvements in overall system performance.
- Algorithms are evaluated using a given string of page accesses known as a *reference string*.

Few Page Replacement algorithms –

a) FIFO Page Replacement

- A simple and obvious page replacement strategy is *FIFO*, i.e. first-in-first-out.
- This 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.

Module IV – Virtual Memory Management & File System

- Or a FIFO queue can be created to hold all pages in memory. As new pages are brought in, they are added to the tail of a queue, and the page at the head of the queue is the next victim.
- In the following example, a reference string is given and there are 3 free frames. There are 20 page requests, which results in 15 page faults.

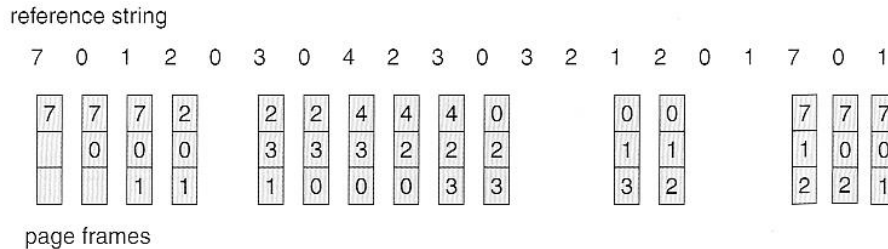


Figure 9.12 FIFO page-replacement algorithm.

- Although FIFO is simple and easy to understand, it is not always optimal, or even efficient.
- **Belady's anomaly** tells that for some page-replacement algorithms, the page-fault rate may *increase* as the number of allocated frames increases.

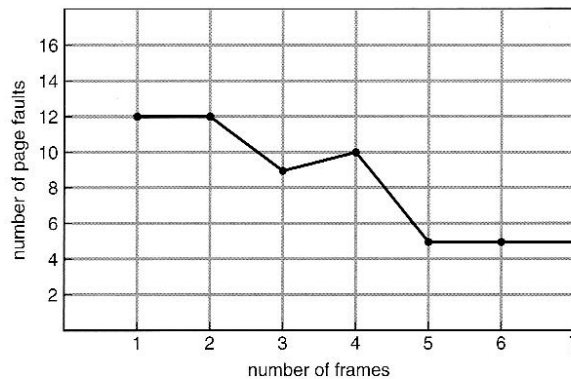


Figure 9.13 Page-fault curve for FIFO replacement on a reference string.

b) Optimal Page Replacement

- The discovery of Belady's anomaly led to the search for an **optimal page-replacement algorithm**, which is simply that which yields the lowest of all possible page-faults, and which does not suffer from Belady's anomaly.
- Such an algorithm does exist, and is called **OPT or MIN**. This algorithm is "Replace the page that will not be used for the longest time in the future."
- The same reference string used for the FIFO example is used in the example below, here the minimum number of possible page faults is 9.

Module IV – Virtual Memory Management & File System

- Unfortunately OPT cannot be implemented in practice, because it requires the knowledge of future string, but it makes a nice benchmark for the comparison and evaluation of real proposed new algorithms.

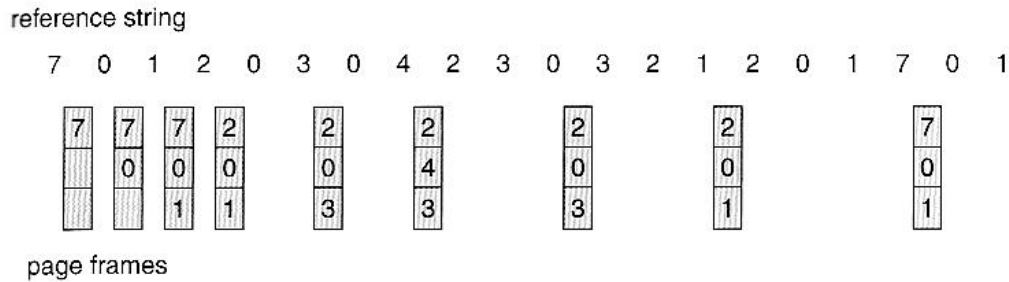


Figure 9.14 Optimal page-replacement algorithm.

c) LRU Page Replacement

- The **LRU (Least Recently Used)** algorithm, predicts that the page that has not been used in the longest time is the one that will not be used again in the near future.
- Some view LRU as analogous to OPT, but here we look backwards in time instead of forwards.
- Figure 9.15 illustrates LRU for our sample string, yielding 12 page faults, (as compared to 15 for FIFO and 9 for OPT.)

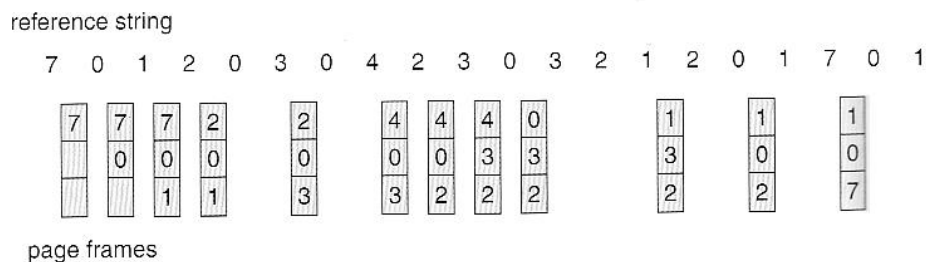


Figure 9.15 LRU page-replacement algorithm.

- LRU is considered a good replacement policy, and is often used. There are two simple approaches commonly used to implement this:
 - Counters.** With each page-table entry a time-of-use field is associated. Whenever a reference to a page is made, the contents of the clock register are copied to the time-of-use field in the page-table entry for that page. In this way, we always have the "time" of the last reference to each page. This scheme requires a search of the page table to find the LRU page and a write to memory for each memory access.

2. **Stack.** Another approach is to use a stack, and whenever a page is accessed, pull that page from the middle of the stack and place it on the top. The LRU page will always be at the bottom of the stack. Because this requires removing objects from the middle of the stack, a doubly linked list is the recommended data structure.
- Neither LRU or OPT exhibit Belady's anomaly. Both belong to a class of page-replacement algorithms called *stack algorithms*, which can never exhibit Belady's anomaly.

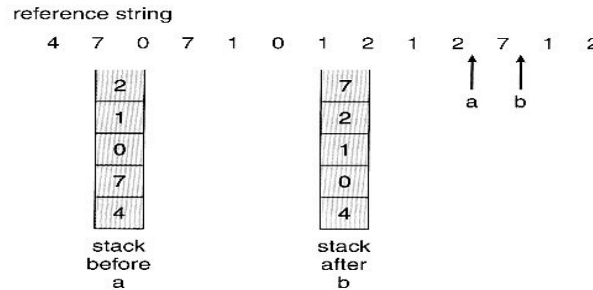


Figure 9.16 Use of a stack to record the most recent page references.

d) LRU-Approximation Page Replacement

- Many systems offer some degree of hardware support, enough to approximate LRU.
- In particular, many systems provide a *reference bit* for every entry in a page table, which is set anytime that page is accessed. Initially all bits are set to zero, and they can also all be cleared at any time. One bit distinguishes pages that have been accessed since the last clear from those that have not been accessed.

d.1 Additional-Reference-Bits Algorithm

- An 8-bit byte (reference bit) is stored for each page in a table in memory.
- At regular intervals (say, every 100 milliseconds), a timer interrupt transfers control to the operating system. The operating system shifts the reference bit for each page into the high-order bit of its 8-bit byte, shifting the other bits right by 1 bit and discarding the low-order bit.
- These 8-bit shift registers contain the history of page use for the last eight time periods.
- If the shift register contains 00000000, then the page has not been used for eight time periods.
- A page with a history register value of 11000100 has been used more recently than one with a value of 01110111.

d.2 Second-Chance Algorithm

- The *second chance algorithm* is a FIFO replacement algorithm, except the reference bit is used to give pages a second chance at staying in the page table.
- When a page must be replaced, the page table is scanned in a FIFO (circular queue) manner.

- If a page is found with its reference bit as '0', then that page is selected as the next victim.
- If the reference bit value is '1', then the page is given a second chance and its reference bit value is cleared(assigned as '0').

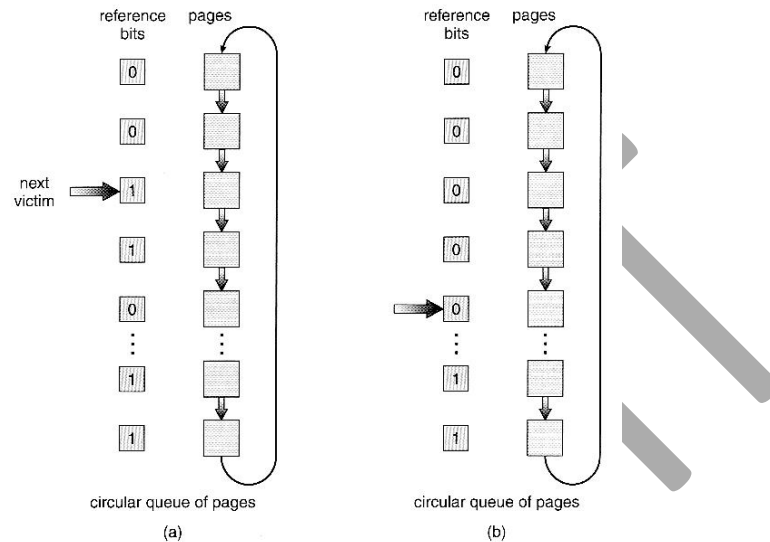


Figure 9.17 Second-chance (clock) page-replacement algorithm.

- Thus, a page that is given a second chance will not be replaced until all other pages have been replaced (or given second chances). In addition, if a page is used often, then it sets its reference bit again.
- This algorithm is also known as the *clock* algorithm.

One way to implement the second-chance algorithm is as a circular queue. A pointer indicates which page is *to* be replaced next. When a frame is needed, the pointer advances until it finds a page with a 0 reference bit. As it advances, it clears the reference bits. Once a victim page is found, the page is replaced, and the new page is inserted in the circular queue in that position.

d.3 Enhanced Second-Chance Algorithm

- The **enhanced second chance algorithm** looks at the reference bit and the modify bit (dirty bit) as an ordered page, and classifies pages into one of four classes:
 1. (0, 0) - Neither recently used nor modified.
 2. (0, 1) - Not recently used, but modified.
 3. (1, 0) - Recently used, but clean.
 4. (1, 1) - Recently used and modified.
- This algorithm searches the page table in a circular fashion, looking for the first page it can find in the lowest numbered category. i.e. it first makes a pass looking for a (0, 0), and then if it can't find one, it makes another pass looking for a (0, 1), etc.
- The main difference between this algorithm and the previous one is the preference for replacing clean pages if possible.

e) Counting-Based Page Replacement

- There are several algorithms based on counting the number of references that have been made to a given page, such as:
 - **Least Frequently Used, LFU:** Replace the page with the lowest reference count. A problem can occur if a page is used frequently initially and then not used any more, as the reference count remains high. A solution to this problem is to right-shift the counters periodically, yielding a time-decaying average reference count.
 - **Most Frequently Used, MFU:** Replace the page with the highest reference count. The logic behind this idea is that pages that have already been referenced a lot have been in the system a long time, and we are probably done with them, whereas pages referenced only a few times have only recently been loaded, and we still need them.

f) Page-Buffering Algorithms

- Maintain a certain minimum number of free frames at all times. When a page-fault occurs, go ahead and allocate one of the free frames from the free list first, so that the requesting process is in memory as early as possible, and then select a victim page to write to disk and free up a frame.
- Keep a list of modified pages, and when the I/O system is idle, these pages are written to disk, and then clear the modify bits, thereby increasing the chance of finding a "clean" page for the next potential victim and page replacement can be done much faster.

9.5 Allocation of Frames

- The absolute minimum number of frames that a process must be allocated is dependent on system architecture.
- The maximum number is defined by the amount of available physical memory.

Allocation Algorithms

After loading of OS, there are two ways in which the allocation of frames can be done to the processes.

- **Equal Allocation** - If there are m frames available and n processes to share them, each process gets m / n frames, and the leftovers are kept in a free-frame buffer pool.
- **Proportional Allocation** - Allocate the frames proportionally depending on the size of the process. If the size of process i is S_i , and S is the sum of size of all processes in the system, then the allocation for process P_i is $a_i = m * S_i / S$. where m is the free frames available in the system.

Consider a system with a 1KB frame size. If a small student process of 10 KB and an interactive database of 127 KB are the only two processes running in a system with 62 free frames.

Module IV – Virtual Memory Management & File System

With proportional allocation, we would split 62 frames between two processes, as follows-

$$m=62, S = (10+127)=137$$

$$\text{Allocation for process 1} = 62 \times 10/137 \sim 4$$

$$\text{Allocation for process 2} = 62 \times 127/137 \sim 57$$

Thus allocates 4 frames and 57 frames to student process and database respectively.

- Variations on proportional allocation could consider priority of process rather than just their size.

Global versus Local Allocation

- Page replacement can occur both at local or global level.
- With local replacement, the number of pages allocated to a process is fixed, and page replacement occurs only amongst the pages allocated to this process.
- With global replacement, any page may be a potential victim, whether it currently belongs to the process seeking a free frame or not.
- Local page replacement allows processes to better control their own page fault rates, and leads to more consistent performance of a given process over different system load levels.
- Global page replacement is overall more efficient, and is the more commonly used approach.

Non-Uniform Memory Access (New)

- Usually the time required to access all memory in a system is equivalent.
- This may not be the case in multiple-processor systems, especially where each CPU is physically located on a separate circuit board which also holds some portion of the overall system memory.
- In such systems, CPUs can access memory that is physically located on the same board much faster than the memory on the other boards.
- The basic solution is akin to processor affinity - At the same time that we try to schedule processes on the same CPU to minimize cache misses, we also try to allocate memory for those processes on the same boards, to minimize access times.

9.6 Thrashing

Thrashing is the state of a process where there is **high paging activity**. A process that is spending more time paging than executing is said to be *thrashing*.

9.6.1 Cause of Thrashing

Module IV – Virtual Memory Management & File System

- When memory is filled up and processes start spending lots of time waiting for their pages to page in, then CPU utilization decreases (Processes are not executed as they are waiting for some pages), causing the scheduler to add in even more processes and increase the degree of multiprogramming even more. Thrashing has occurred, and system throughput plunges. No work is getting done, because the processes are spending all their time paging.
- In the graph given below, CPU utilization is plotted against the degree of multiprogramming. As the degree of multiprogramming increases, CPU utilization also increases, although more slowly, until a maximum is reached. If the degree of multiprogramming is increased even further, thrashing sets in, and CPU utilization drops sharply. At this point, to increase CPU utilization and stop thrashing, we must *decrease* the degree of multiprogramming.

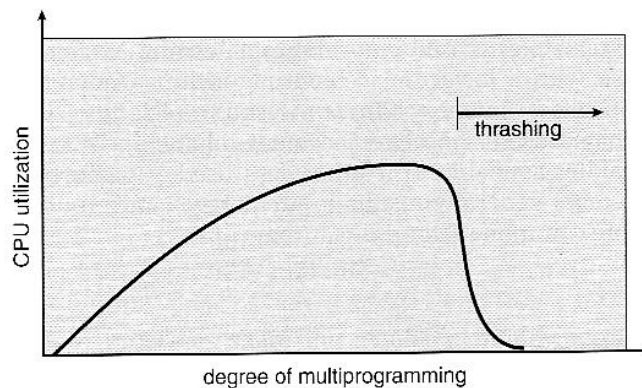


Figure 9.18 Thrashing.

- Local page replacement policies can prevent thrashing process from taking pages away from other processes, but it still tends to clog up the I/O queue.

9.6.2 Working-Set Model

- The *working set model* is based on the concept of locality, and defines a *working set window*, of length Δ . Whatever pages are included in the most recent Δ page references are said to be in the processes working set window, and comprise its current working set, as illustrated in Figure 9.20:

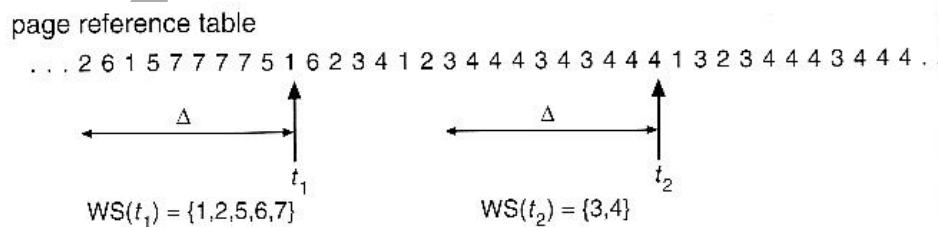


Figure 9.20 Working-set model.

- The selection of delta is critical to the success of the working set model - If it is too small then it does not encompass all of the pages of the current locality, and if it is too large, then it encompasses pages that are no longer being frequently accessed.
- The total demand of frames, D , is the sum of the sizes of the working sets for all processes ($D = \sum WSS_i$). If D exceeds the total number of available frames, then at least one process is thrashing, because there are not enough frames available to satisfy its minimum working set. If D is significantly less than the currently available frames, then additional processes can be launched.
- The hard part of the working-set model is keeping track of what pages are in the current working set, since every reference adds one to the set and removes one older page.

9.6.3

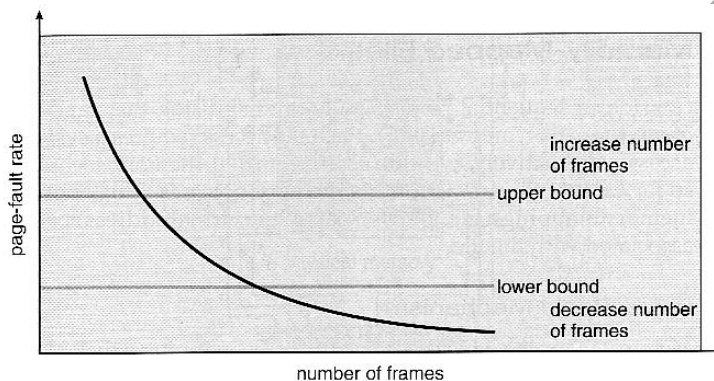


Figure 9.21 Page-fault frequency.

Page-Fault Frequency

- When page-fault rate is too high, the process needs more frames and when it is too low, the process may have too many frames.
- The upper and lower bounds can be established on the page-fault rate. If the actual page-fault rate exceeds the upper limit, allocate the process another frame or suspend the process. If the page-fault rate falls below the lower limit, remove a frame from the process. Thus, we can directly measure and control the page-fault rate to prevent thrashing.

Module IV

Implementation of File System

- **File Concept**
 - **File Attributes**
 - **File Operations**
 - **File Types**
 - **File Structure**
 - **Internal File Structure**
- **Access Methods**
 - **Sequential Access**
 - **Direct Access**
 - **Other Access Methods – Indexed Access**
- **Directory and Disk Structure**
 - **Storage Structure**
 - **Directory Overview**
 - **Single-Level Directory**
 - **Two-Level Directory**
 - **Tree –Structured Directories**
 - **Acyclic –Graph Directories**
 - **General Graph Directory**
- **File System Mounting**
- **File Sharing**
 - **Multiple Users**
 - **Remote File Systems**
 - **The Client-Server Model**
 - **Distributed Information Systems**
 - **Failure Modes**
- **Consistency Semantics**
 - **Unix Semantics**
 - **Session Semantics**
 - **Immutable – Shared Files Semantics**
- **Protection – Types, Access Control, other approaches**
- **File-System Structure**
- **File-System Implementation**
- **Directory Implementation**

- **Allocation Methods**
- **Free-Space Management**

File Concepts

The file system consists of two distinct parts: a collection of files, each storing related data, and a directory structure, which organizes and provides information about all the files in the system.

A file is a collection of related information that is recorded on secondary storage. A file is a sequence of bits, bytes, lines, or records, the meaning of which is defined by the file's creator and user.

a) File Attributes

A file is named, for the convenience of its human users, and is referred to by its name. A file's attributes vary from one operating system to another. The attributes of a file are p-

- **Name** - The symbolic file name is the only information kept in human-readable form. Some special significance is given to names, and particularly extensions (.exe, .txt, etc.).
- **Identifier** - It is a unique number, that identifies the file within the file system.
- **Type** - Type of the file like text, executable, other binary, etc.
- **Location** - . location of the file on that device.
- **Size** - The current size of the file (in bytes, words, or blocks)
- **Protection** - Access-control information (reading, writing, executing).
- **Time, date, and user identification** - These data can be useful for protection, security, and usage monitoring.

b) File Operations

The operating system provides system calls to create, write, read, reposition, delete, and truncate files.

- **Creating a file** - Two steps are necessary to create a file
 - Find space in the file system for the file.
 - Make an entry for the new file in the directory.
- **Writing a file** - To write a file, the system call consists of 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 file's location. 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, the system call that specifies the name of the file and where the next block of the file should be put. The directory is searched for the file, 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.
- **Repositioning within a file** - The directory is searched for the file, and the file pointer is repositioned to a given value. This file operation is also known as a file seek.
- **Deleting a file** - To delete a file, search the directory for the file. 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. The file size is reset to zero.
- Information about currently open files is stored in an **open file table**. It contains informations like:
 - **File pointer** - records the current position in the file, for the next read or write access.
 - **File-open count** - How many times has the current file been opened by different processes, at the same time and not yet closed? When this counter reaches zero the file can be removed from the table.
 - **Disk location of the file** – The information needed to locate the file on disk is kept in memory so that the system does not have to read it from disk for each operation.
 - **Access rights** – The file access permissions are stored on the per-process table so that the operating system can allow or deny subsequent I/O requests.
- Some systems provide support for **file locking**.
 - A **shared lock** is for reading only.
 - A **exclusive lock** is for writing as well as reading.
 - An **advisory lock**, it is up to the software developers to ensure that locks are acquired or released.
 - A **mandatory lock**, prevents any other process from accessing the locked file.. (A truly locked door.)
 - UNIX uses advisory locks, and Windows uses mandatory locks.

c) 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, pas, asm, a	source code in various languages
batch	bat, sh	commands to the command interpreter
text	txt, doc	textual data, documents
word processor	wp, tex, rtf, doc	various word-processor formats
library	lib, a, so, dll	libraries of routines for programmers
print or view	ps, pdf, jpg	ASCII or binary file in a format for printing or viewing
archive	arc, zip, tar	related files grouped into one file, sometimes compressed, for archiving or storage
multimedia	mpeg, mov, rm, mp3, avi	binary file containing audio or A/V information

Figure 10.2 Common file types.

- File name consists of two parts: name and extension
- The user and the operating system can identify the type of a file using the name.
- Most operating systems allow users to specify a file name as a sequence of characters followed by a period and terminated by an extension. Example : resume.doc, threads.c etc.
- 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", can be executed.

d) File Structure

- The study of different ways of storing files in secondary memory such that they can be easily accessed.
- File types can be used to indicate the internal structure of the file. Certain files must be in a particular structure that is understood by the operating system.
- For example, the operating system requires that an executable file have a specific structure so that it can determine where in memory to load the file and the location of the first instruction.
- UNIX treats all files as sequences of bytes, with no further consideration of the internal structure.
- Macintosh files have two **forks** - a **resource fork**, and a **data fork**. The resource fork contains information relating to the UI, such as icons and button images. The data fork contains the traditional file contents-program code or data.

e) Internal File Structure

- Disk systems typically have a well-defined block size determined by the size of a sector. A group of sectors form a group
- All disk I/O is performed in units of one block, and all blocks are the same size.
- Logical records may even vary in length. Padding a number of logical records into physical blocks is a common solution to this problem.
- The packing can be done either by the user's application program or by the operating system. In either case, the file may be considered a sequence of blocks.
- All the basic I/O functions operate in terms of blocks.
- Disk space is always allocated in terms of blocks. Some portion of last block(while storing a file) is always wasted. This is called internal fragmentation.

10.2 Access Methods

The file information is accessed and read into computer memory. The information in the file can be accessed in several ways.

a) Sequential Access

- Here information in the file is processed in order, one record after the other.
- This mode of access is a common method; for example, editors and compilers usually access files in this fashion.
- A sequential access file emulates magnetic tape operation, and generally supports a few operations:
 - read next - read a record and advance the file pointer to the next position.
 - write next - write a record to the end of file and advance the file pointer to the next position.
 - skip n records - May or may not be supported. 'n' may be limited to positive numbers, or may be limited to +/- 1.

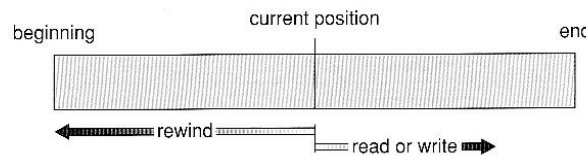


Figure 10.3 Sequential-access file.

b) Direct Access

A file is made up of fixed-length logical records that allow programs to read and write records randomly. The records can be rapidly accessed in any order.

Direct access are of great use for immediate access to large amount of information.

Eg : Database file. When a query occurs, the query is computed and only the selected rows are access directly to provide the desired information.

Operations supported include:

- read n - read record number n. (position the cursor to n and then read the record)
- write n - write record number n. (position the cursor to n and then write the record)
- jump to record n – move to nth record (n- could be 0 or the end of file)
- If the record length is L, there is a request for record 'N'. Then the direct access to the starting byte of record 'N' is at $L*(N-1)$

Eg: if 3rd record is required and length of each record(L) is 50, then the starting position of 3rd record is $L*(N-1)$

$$\text{Address} = 50*(3-1) = 100.$$

c) Other Access Methods(Indexed method)

- These methods generally involve the construction of an *index* for the file called **index file**.

- The index file is like an index page of a book, which contains key and address. To find a record in the file, we first search the index and then use the pointer to access the record directly and find the desired record.
- An indexed access scheme can be easily built on top of a direct access system.
- For very large files, the index file itself is very large. The solution to this is to create an index for index file. i.e. multi-level indexing.

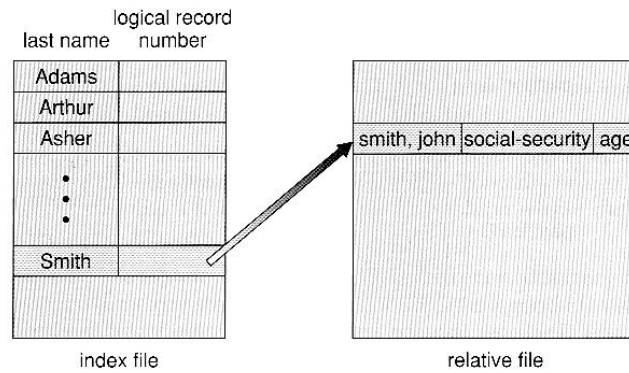


Figure 10.5 Example of index and relative files.

10.3 Directory Structure

Directory is a structure which contains filenames and information about the files like location, size, type etc. The files are put in different directories. Partitioning is useful for limiting the sizes of individual file systems, putting multiple file-system types on the same device, or leaving part of the device available for other uses.

Partitions are also known as *slices* or *minidisks*. A file system can be created on each of these parts of the disk. Any entity containing a file system is generally known as a *volume*.

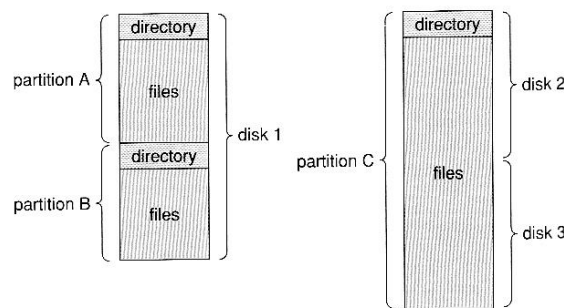


Figure 10.6 A typical file-system organization.

Directory Overview

The directory can be viewed as a symbol table that translates file names into their directory entries.

Directory operations to be supported include:

- Search for a file - search a directory structure to find the entry for a particular file.
- Create a file – create new files and add to the directory
- Delete a file - When a file is no longer needed, erase it from the directory
- List a directory - list the files in a directory and the contents of the directory entry.
- Rename a file – Change the name of the file. Renaming a file may also allow its position within the directory structure to be changed.
- Traverse the file system - Access every directory and every file within a directory structure.

Directory Structures -

a) Single-Level Directory

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

The limitations of this structure is that -

- All files in the same directory must have unique names.
- 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.

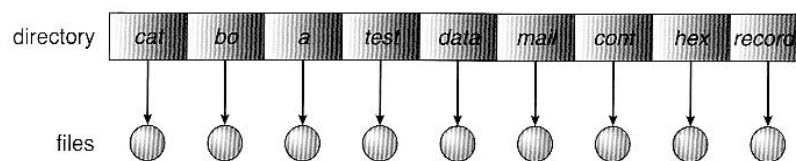


Figure 10.7 Single-level directory.

b) Two-Level Directory

- Each user gets their own directory space - *user file directory(UFD)*
- File names only need to be unique within a given user's directory.
- A **master file directory(MFD)** is used to keep track of each users directory, and must be maintained when users are added to or removed from the system.
- When a user refers to a particular file, only his own UFD is searched.
- All the files 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. The user directories themselves must be created and deleted as necessary.
- This structure **isolates** one user from another. Isolation is an advantage when the users are completely independent but is a disadvantage when the users *want* to cooperate on some task and to access one another's files.

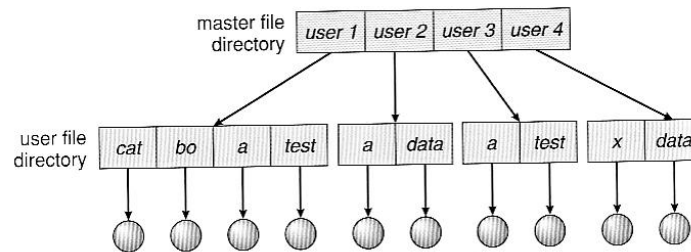


Figure 10.8 Two-level directory structure.

c) Tree-Structured Directories

- A tree structure is the most common directory structure.
- The tree has a root directory, and every file in the system has a unique path name.
- A directory (or subdirectory) contains a set of files or subdirectories.
- One bit in each directory entry defines the entry as a file (0) or as a subdirectory (1). Special system calls are used to create and delete directories.
- Path names can be of two types: *absolute* and *relative*. An absolute path begins at the root and follows a down to the specified file, giving the directory names on the path. A relative path defines a path from the current directory.
- For example, in the tree-structured file system of figure below if the current directory is *root/spell/mail*, then the relative path name is *prt/first* and the files absolute path name *root/spell/mail/prt/first*.

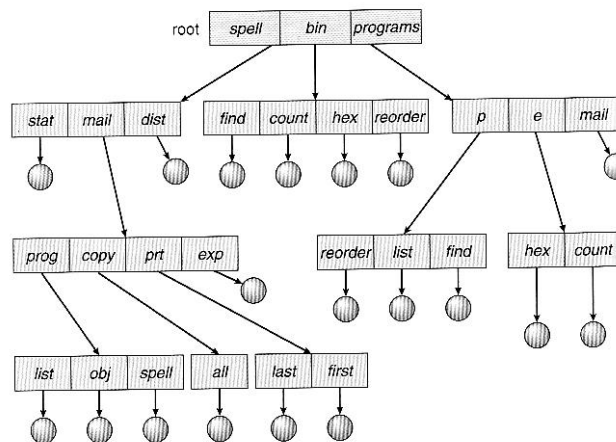


Figure 10.9 Tree-structured directory structure.

- Directories are stored the same as any other file in the system, except there is a bit that identifies them as directories, and they have some special structure that the OS understands.
- One question for consideration is whether or not to allow the removal of directories that are not empty - Windows requires that directories be emptied first, and UNIX provides an option for deleting entire sub-trees.

d) Acyclic-Graph Directories

- When the same files need to be accessed in more than one place in the directory structure (e.g. because they are being shared by more than one user), it can be useful to provide an acyclic-graph structure. (Note the **directed** arcs from parent to child.)
 - UNIX provides two types of **links** (pointer to another file)for implementing the acyclic-graph structure.
 - A **hard link** (usually just called a link) involves multiple directory entries that both refer to the same file. Hard links are only valid for ordinary files in the same filesystem.
 - A **symbolic link**, that involves a special file, containing information about where to find the linked file. Symbolic links may be used to link directories and/or files in other filesystems, as well as ordinary files in the current filesystem.
- Windows only supports symbolic links, termed **shortcuts**.
- Hard links require a **reference count**, or **link count** for each file, keeping track of how many directory entries are currently referring to this file. Whenever one of the references is removed the link count is reduced, and when it reaches zero, the disk space can be reclaimed.

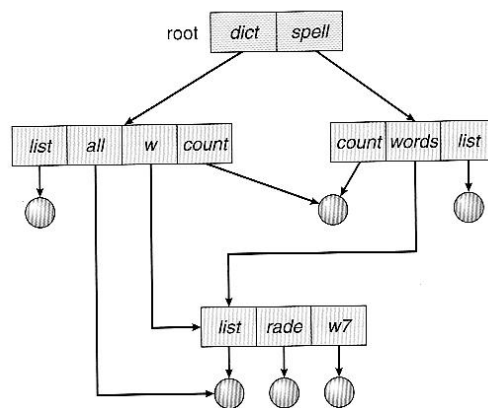


Figure 10.10 Acyclic-graph directory structure.

- For symbolic links there is some question as to what to do with the symbolic links when the original file is moved or deleted:
 - One option is to find all the symbolic links and adjust them also.

- Another is to leave the symbolic links dangling, and discover that they are no longer valid the next time they are used.
- What if the original file is removed, and replaced with another file having the same name before the symbolic link is next used?

Another approach to deletion is to preserve the file until all references to it are deleted. To implement this approach, we must have some mechanism for determining that the last reference to the file has been deleted.

When a link or a copy of the directory entry is established, a new entry is added to the file-reference list. When a link or directory entry is deleted, we remove its entry on the list. The file is deleted when its file-reference list is empty.

e) General Graph Directory

- If cycles are allowed in the graphs, then several problems can arise:
 - Search algorithms can go into infinite loops. One solution is to not follow links in search algorithms. (Or not to follow symbolic links, and to only allow symbolic links to refer to directories)
 - Sub-trees can become disconnected from the rest of the tree and still not have their reference counts reduced to zero. Periodic garbage collection is required to detect and resolve this problem. (chkdsk in DOS and fsck in UNIX search for these problems, among others, even though cycles are not supposed to be allowed in either system. Disconnected disk blocks that are not marked as free are added back to the file systems with made-up file names, and can usually be safely deleted.)

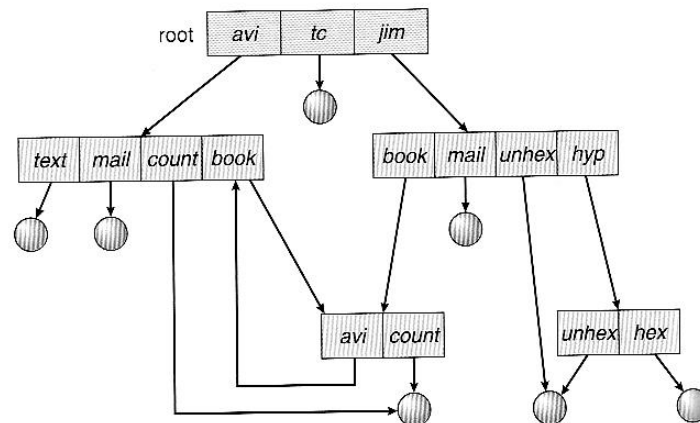


Figure 10.11 General graph directory.

10.4 File-System Mounting

- The basic idea behind mounting file systems is to combine multiple file systems into one large tree structure.
- The mount command is given a filesystem to mount and a **mount point** (directory) on which to attach it.
- Once a file system is mounted onto a mount point, any further references to that directory actually refer to the root of the mounted file system.
- Any files (or sub-directories) that had been stored in the mount point directory prior to mounting the new filesystem are now hidden by the mounted filesystem, and are no longer available. For this reason some systems only allow mounting onto empty directories.
- Filesystems can only be mounted by root, unless root has previously configured certain filesystems to be mountable onto certain pre-determined mount points. (E.g. root may allow users to mount floppy filesystems to /mnt or something like it) Anyone can run the mount command to see what filesystems are currently mounted.
- Filesystems may be mounted read-only, or have other restrictions imposed.

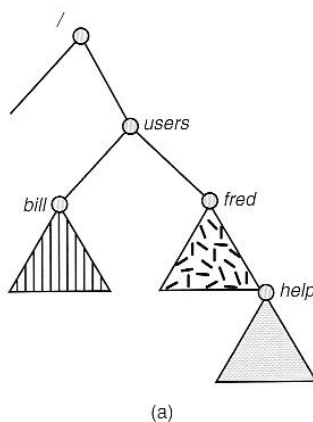


Figure 10.12 File system. (a) Existing system. (b) Unmounted volume.

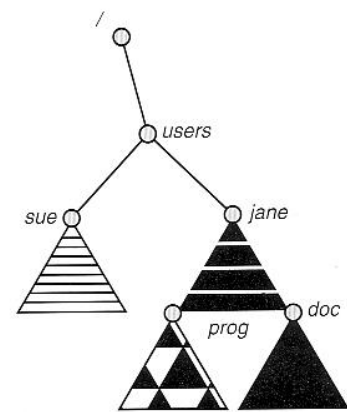
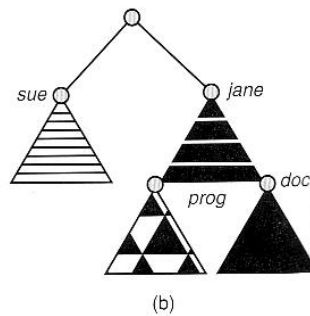


Figure 10.13 Mount point.

The traditional Windows OS runs an extended two-tier directory structure, where the first tier of the structure separates volumes by drive letters, and a tree structure is implemented below that level.

- Macintosh runs a similar system, where each new volume that is found is automatically mounted and added to the desktop when it is found.
- More recent Windows systems allow filesystems to be mounted to any directory in the filesystem, much like UNIX.

10.5 File Sharing

10.5.1 Multiple Users

- On a multi-user system, more information needs to be stored for each file:
 - The owner (user) who owns the file, and who can control its access.
 - The group of other user IDs that may have some special access to the file.
 - What access rights are afforded to the owner (**U**ser), the **G**roup, and to the rest of the world (the universe, a.k.a. **O**thers.)
 - Some systems have more complicated access control, allowing or denying specific accesses to specifically named users or groups.

10.5.2 Remote File Systems

- The advent of the Internet introduces issues for accessing files stored on remote computers
 - The original method was ftp, allowing individual files to be transported across systems as needed. Ftp can be either account and password controlled, or **anonymous**, not requiring any user name or password.
 - Various forms of **distributed file systems** allow remote file systems to be mounted onto a local directory structure, and accessed using normal file access commands. (The actual files are still transported across the network as needed, possibly using ftp as the underlying transport mechanism.)
 - The WWW has made it easy once again to access files on remote systems without mounting their filesystems, generally using (anonymous) ftp as the underlying file transport mechanism.

a) The Client-Server Model

- When one computer system remotely mounts a filesystem that is physically located on another system, the system which physically owns the files acts as a **server**, and the system which mounts them is the **client**.
- User IDs and group IDs must be consistent across both systems for the system to work properly. (I.e. this is most applicable across multiple computers managed by the same organization, shared by a common group of users.)
- The same computer can be both a client and a server. (E.g. cross-linked file systems.)
- There are a number of security concerns involved in this model:
 - Servers commonly restrict mount permission to certain trusted systems only. Spoofing (a computer pretending to be a different computer) is a potential security risk.
 - Servers may restrict remote access to read-only.
 - Servers restrict which filesystems may be remotely mounted. Generally the information within those subsystems is limited, relatively public, and protected by frequent backups.

- The NFS (Network File System) is a classic example of such a system.

b) Distributed Information Systems

- The **Domain Name System, DNS**, provides for a unique naming system across all of the Internet.
- Domain names are maintained by the **Network Information System, NIS**, which unfortunately has several security issues. NIS+ is a more secure version, but has not yet gained the same widespread acceptance as NIS.
- Microsoft's **Common Internet File System, CIFS**, establishes a **network login** for each user on a networked system with shared file access. Older Windows systems used **domains**, and newer systems (XP, 2000), use **active directories**. User names must match across the network for this system to be valid.
- A newer approach is the **Lightweight Directory-Access Protocol, LDAP**, which provides a **secure single sign-on** for all users to access all resources on a network. This is a secure system which is gaining in popularity, and which has the maintenance advantage of combining authorization information in one central location.

c) Failure Modes

- When a local disk file is unavailable, the result is generally known immediately, and is generally non-recoverable. The only reasonable response is for the response to fail.
- However when a remote file is unavailable, there are many possible reasons, and whether or not it is unrecoverable is not readily apparent. Hence most remote access systems allow for blocking or delayed response, in the hopes that the remote system (or the network) will come back up eventually.

10.5.3 Consistency Semantics

- **Consistency Semantics** deals with the consistency between the views of shared files on a networked system. When one user changes the file, when do other users see the changes?
- The series of accesses between the open() and close() operations of a file is called the **file session**.

Examples of consistency semantics -

a) UNIX Semantics

- The UNIX file system uses the following semantics:
 - Writes to an open file are immediately visible to any other user who has the file open.

- One implementation uses a shared location pointer, which is adjusted for all sharing users.
- There is a single copy of the file, which may delay some accesses.

b) Session Semantics

- The Andrew File System, AFS uses the following semantics:
 - Writes to an open file are not immediately visible to other users.
 - When a file is closed, any changes made become available only to users who open the file at a later time.
- According to these semantics, a file can be associated with multiple (possibly different) views. Almost no constraints are imposed on scheduling accesses. No user is delayed in reading or writing their personal copy of the file.
- AFS file systems may be accessible by systems around the world. Access control is maintained through (somewhat) complicated access control lists, which may grant access to the entire world (literally) or to specifically named users accessing the files from specifically named remote environments.

c) Immutable-Shared-Files Semantics

- Under this system, when a file is declared as **shared** by its creator, then the name cannot be re-used by any other process and it cannot be modified.

10.6 Protection

The information in a computer system must be stored safely without any physical damage (the issue of *reliability*) and improper access (the issue of *protection*).

Reliability is generally provided by duplicate copies of files. Many computers have systems programs that automatically copy disk files to tape at regular intervals (once per day or week or month). The damage could be due to hardware problems (such as errors in reading or writing), power surges or failures, head crashes, dirt, temperature extremes, and vandalism.

10.6.1 Types of Access

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. 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.

10.6.2 Access Control

To make access to files depending on the identity of the user. Different users may need different types of access to a file or directory. The most general scheme to implement dependent access is to associate with each file and directory an access-control list (ACL) specifying user names 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.

If we want to allow everyone to read a file, we must list all users with read access. This technique has two undesirable consequences:

- Constructing such a list may be a tedious, if we do not know in advance the list of users in the system.
- The directory entry, must be of variable size, as the list grows, 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 owner, group, and universe access control scheme.

10.6.3 Other Protection Approaches

Associate a password with each file.

Using of password is effective, but has a few disadvantages:

- The number of passwords that a user needs to remember maybe large
- If one password is used, then once the password is discovered, all the files can be accessed.

Some system allow users to associate a password to a subdirectory, rather than only to file.

10.7 File-System Structure

Disks provide the bulk of secondary storage on which a file system is maintained. The two characteristics that make them a convenient medium for storing multiple files:

1. A disk can be rewritten in place; it is possible to read a block from the disk, modify the block, and write it back into the same place.
 2. A disk can access directly any given block of information it contains. Thus, it is simple to access any file either sequentially or randomly, and switching from one file to another requires only moving the read-write heads and waiting for the disk to rotate.
- To improve I/O efficiency, I/O transfers between memory and disk are performed in units of *blocks*. Block sizes may range from 512 bytes to 4K or larger.(Rather than transferring a byte at a time,)
 - A file system poses two quite different design problems.
 - The first problem is defining how the file system should look to the user. This task involves defining a file and its attributes, the operations allowed on a file, and the directory structure for organizing files.
 - The second problem is creating algorithms and data structures to map the logical file system onto the physical secondary-storage devices.
 - File systems organize storage on disk drives, and can be viewed as a layered design:
 - At the lowest layer are the physical devices, consisting of the magnetic media, motors & controls, and the electronics connected to them and controlling them. Modern disk put more and more of the electronic controls directly on the disk drive itself, leaving relatively little work for the disk controller card to perform.
 - Lowest level, **I/O Control** consists of **device drivers**, which communicate with the devices by reading and writing special codes directly to and from memory addresses corresponding to the controller card's registers. Each controller card (device) on a system has a different set of addresses (registers, **ports**) that it listens to, and a unique set of command codes and results codes that it understands.
 - The **basic file system** level works directly with the device drivers in terms of retrieving and storing raw blocks of data, without any consideration for what is in each block.
 - The **file organization module** knows about files and their logical blocks, and how they map to physical blocks on the disk. In addition to translating from logical to physical blocks, the file organization module also maintains the list of free blocks, and allocates free blocks to files as needed.
 - The **logical file system** deals with all of the meta data associated with a file (UID, GID, mode, dates, etc), i.e. everything about the file except the data itself. This level manages the directory structure and the mapping of file names to **file control blocks, FCBs**, which contain all of the meta data as well as block number information for finding the data on the disk.
 - The layered approach to file systems means that much of the code can be used uniformly for a wide variety of different file systems, and only certain layers need to be filesystem specific.

- When a layered structure is used for file-system implementation, duplication of code is minimized. The I/O control and sometimes the basic file-system code can be used by multiple file systems.
- Common file systems in use include the UNIX file system, UFS, the Berkeley Fast File System, FFS, Windows systems FAT, FAT32, NTFS, CD-ROM systems ISO 9660, and for Linux the extended file systems ext2 and ext3 .

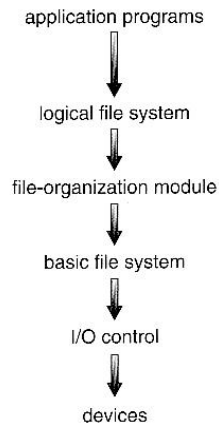


Figure 11.1 Layered file system.

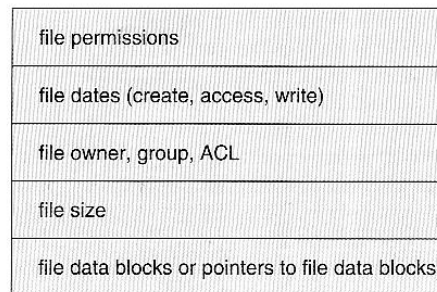
10.8 File-System Implementation

On disk, the file system may contain information about how to boot an operating system stored there, the total number of blocks, the number and location of free blocks, the directory structure, and individual files.

File systems store several important data structures on the disk:

- A **boot-control block**, (per volume) can contain information needed by the system to boot an operating system from that volume. If the disk does not contain an operating system, this block can be empty. It is typically the first block of a volume In UFS, it is called the **boot block**; in NTFS, it is the partition **boot sector**.
- A **volume control block**, (per volume) contains volume (or partition) details, such as the number of blocks in the partition, size of the blocks, freeblock count and free-block pointers, and free FCB count and FCB pointers. In UFS, this is called a **superblock**; in NTFS, it is stored in. the **master file table**.
 - A directory structure (per file system), containing file names and pointers to corresponding FCBs. UNIX uses inode numbers, and NTFS uses a **master file table**.
 - The **File Control Block, FCB**, (per file) containing details about ownership, size, permissions, dates, etc. UNIX stores this information in inodes, and NTFS in the master file table as a relational database structure.

- There are also several key data structures stored in memory:
 - An in-memory mount table contains information about each mounted volume..
 - An in-memory directory cache of recently accessed directory information.
 - A **system-wide open file table**, containing a copy of the FCB for every currently open file in the system, as well as some other related information.
 - A **per-process open file table**, containing a pointer to the system open file table as well as some other information. (For example the current file position pointer may be either here or in the system file table, depending on the implementation and whether the file is being shared or not.)



file permissions
file dates (create, access, write)
file owner, group, ACL
file size
file data blocks or pointers to file data blocks

Figure 11.2 A typical file-control block.

- Figure 11.3 illustrates some of the interactions of file system components when files are created and/or used:
 - When a new file is created, a new FCB is allocated and filled out with important information regarding the new file.
 - When a file is accessed during a program, the `open()` system call reads in the FCB information from disk, and stores it in the system-wide open file table. An entry is added to the per-process open file table referencing the system-wide table, and an index into the per-process table is returned by the `open()` system call. UNIX refers to this index as a **file descriptor**, and Windows refers to it as a **file handle**.
 - If another process already has a file open when a new request comes in for the same file, and it is sharable, then a counter in the system-wide table is incremented and the per-process table is adjusted to point to the existing entry in the system-wide table.
 - When a file is closed, the per-process table entry is freed, and the counter in the system-wide table is decremented. If that counter reaches zero, then the system wide table is also freed. Any data currently stored in memory cache for this file is written out to disk if necessary.

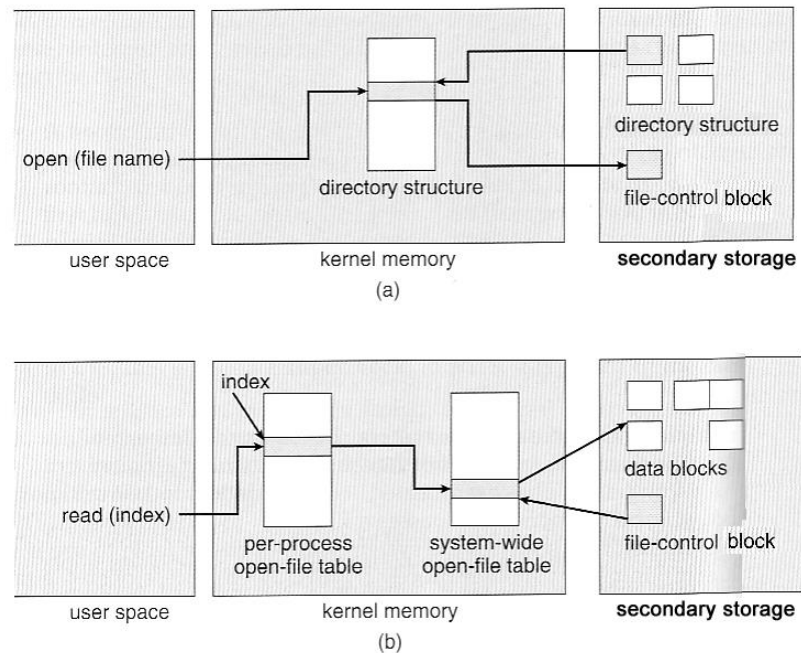


Figure 11.3 In-memory file-system structures. (a) File open. (b) File read.

10.8.1 Partitions and Mounting

- Physical disks are commonly divided into smaller units called partitions. They can also be combined into larger units, but that is most commonly done for RAID installations and is left for later chapters.
- Partitions can either be used as raw devices (with no structure imposed upon them), or "cooked," containing a file system. They can be formatted to hold a filesystem (i.e. populated with FCBs and initial directory structures as appropriate.) Raw partitions are generally used for swap space, and may also be used for certain programs such as databases that choose to manage their own disk storage system. Partitions containing filesystems can generally only be accessed using the file system structure by ordinary users, but can often be accessed as a raw device also by root.
- Boot information can be stored in a separate partition. Again, it has its own format, because at boot time the system does not have file-system device drivers loaded and therefore cannot interpret the file-system format.
- The boot block is accessed as part of a raw partition, by the boot program prior to any operating system being loaded. Modern boot programs understand multiple OSes and filesystem formats, and can give the user a choice of which of several available systems to boot.
- The **root partition** contains the OS kernel and at least the key portions of the OS needed to complete the boot process. At boot time the root partition is mounted, and control is transferred from the boot program to the kernel found there. (Older systems required that the root partition lie completely within the first 1024 cylinders of the disk, because that was as far as the boot program could reach.)

Once the kernel had control, then it could access partitions beyond the 1024 cylinder boundary.)

- Continuing with the boot process, additional filesystems get mounted, adding their information into the appropriate mount table structure. As a part of the mounting process the file systems may be checked for errors or inconsistencies, either because they are flagged as not having been closed properly the last time they were used, or just for general principals. Filesystems may be mounted either automatically or manually. In UNIX a mount point is indicated by setting a flag in the in-memory copy of the inode, so all future references to that inode get re-directed to the root directory of the mounted filesystem.

10.8.2 Virtual File Systems

- Virtual File Systems, VFS**, provide a common interface to multiple different filesystem types. In addition, it provides for a unique identifier (*vnode*) for files across the entire space, including across all filesystems of different types. (UNIX inodes are unique only across a single filesystem, and certainly do not carry across networked file systems.)
- The VFS in Linux is based upon four key object types:
 - The **inode** object, representing an individual file
 - The **file** object, representing an open file.
 - The **superblock** object, representing a filesystem.
 - The **dentry** object, representing an individual directory entry.
- Linux VFS provides a set of common functionalities for each filesystem, using function pointers accessed through a table. The same functionality is accessed through the same table position for all filesystem types, though the actual functions pointed to by the pointers may be filesystem-specific. Common operations provided include `open()`, `read()`, `write()`, and `mmap()`.

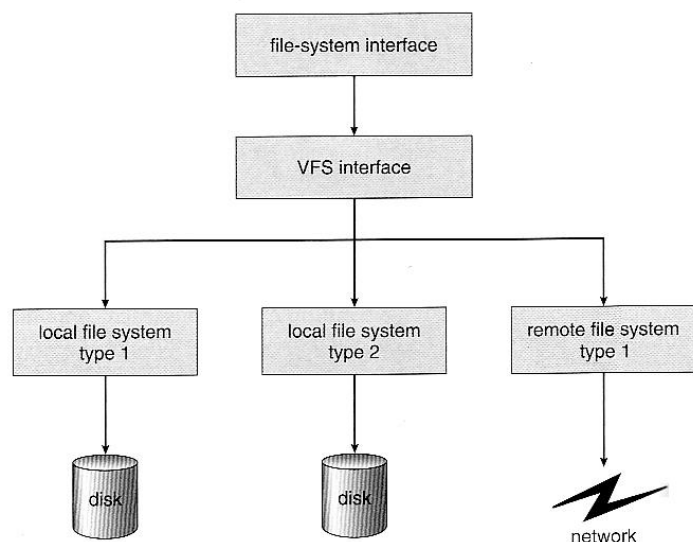


Figure 11.4 Schematic view of a virtual file system.

Figure 11.4. The first layer is the file-system interface, based on the `open()`, `read()`, `write()`, and `close()` calls and on file descriptors. The second layer is called the virtual file system (VFS) layer; it serves two important functions:

1. It separates file-system-generic operations from their implementation by defining a clean VFS interface. Several implementations for the VFS interface may coexist on the same machine, allowing transparent access to different types of file systems mounted locally.
2. The VFS provides a mechanism for uniquely representing a file throughout a network. The VFS is based on a file-representation structure, called a vnode, that contains a numerical designator for a network-wide unique file. (UNIX inodes are unique within only a single file system.) This network-wide uniqueness is required for support of network file systems.

The kernel maintains one vnode structure for each active node (file or directory).

10.9 Directory Implementation

The selection of directory-allocation and directory-management algorithms significantly affects the efficiency, performance, and reliability of the file system. Directories need to be fast to search, insert, and delete, with a minimum of wasted disk space.

a) Linear List

- A linear list is the simplest and easiest directory structure to set up, but it does have some drawbacks.
- The disadvantage of a linear list of directory entries is that finding a file requires a linear search.
- To overcome this, a software cache is implemented to store the recently accessed directory structure.
- Deletions can be done by moving all entries, flagging an entry as deleted, or by moving the last entry into the newly vacant position.
- A sorted list allows a binary search and decreases the average search time. However, the requirement that the list be kept sorted may complicate creating and deleting files,
- A linked list makes insertions and deletions into a sorted list easier, with overhead for the links.
- An advantage of the sorted list is that a sorted directory listing can be produced without a separate sort step.

b) Hash Table

- With this method, a linear list stores the directory entries, but a hash data structure is also used.
- The hash table takes a value computed from the file name and returns a pointer to the file name in the linear list. Therefore, it can greatly decrease the directory search time.

- Here **collisions** may occur. Collision is the situation where two file names hash to the same location.
- Alternatively, a chained-overflow hash table can be used. Each hash entry can be a linked list instead of an individual value, and we can resolve collisions by adding the new entry to the linked list.
- The major disadvantage with a hash table are its generally fixed size and the dependence of the hash function on that size. For example, assume that we make a linear-probing hash table that holds 64 entries. The hash function converts file names into integers from 0 to 63, for instance, by using the remainder of a division by 64. If we later try to create a 65th file, we must enlarge the directory hash table—say, to 128 entries. As a result, we need a new hash function that must map file names to the range 0 to 127, and we must reorganize the existing directory entries to reflect their new hash-function values.

10.10 Allocation Methods

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.

a) Contiguous Allocation

- **Contiguous Allocation** requires that all blocks of a file be kept together contiguously.
- Performance is very fast, because reading successive blocks of the same file generally requires no movement of the disk heads, or at most one small step to the next adjacent cylinder.
- Storage allocation is done by using one of the algorithms (first fit, best fit, worst fit).
- The allocation of blocks contiguous leads to external fragmentation.
- Problems can arise when files grow, or if the exact size of a file is unknown at creation time:
 - Over-estimation of the file's final size increases external fragmentation and wastes disk space.
 - Under-estimation may require that a file be moved or a process aborted if the file grows beyond its originally allocated space.
 - If a file grows slowly over a long time period and the total final space must be allocated initially, then a lot of space becomes unusable before the file fills the space.
- A variation is to allocate file space in large contiguous chunks, called **extents**. When a file outgrows its original extent, then an additional one block is allocated. A pointer points from last block of contiguous memory allocation to the extended chunk.

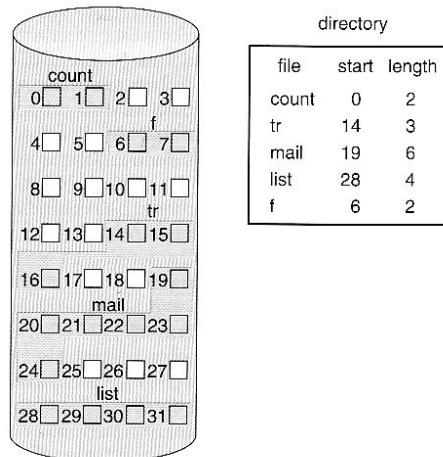


Figure 11.5 Contiguous allocation of disk space.

b) Linked Allocation

- Disk files can be stored as linked lists, with the expense of the storage space consumed by each link. (E.g. a block may be 508 bytes instead of 512.)
- Linked allocation involves no external fragmentation, does not require pre-known file sizes, and allows files to grow dynamically at any time.
- Unfortunately linked allocation is only efficient for sequential access files, as random access requires starting at the beginning of the list for each new location access.
- Allocating *clusters* of blocks reduces the space wasted by pointers, at the cost of internal fragmentation.
- Another big problem with linked allocation is reliability if a pointer is lost or damaged. Doubly linked lists provide some protection, at the cost of additional overhead and wasted space.

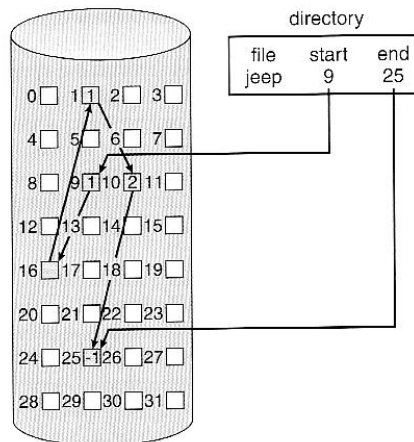


Figure 11.6 Linked allocation of disk space.

- The **File Allocation Table, FAT**, used by DOS is a variation of linked allocation, where all the links are stored in a separate table at the beginning of the disk. The benefit of this approach is that the FAT table can be cached in memory, greatly improving random access speeds.

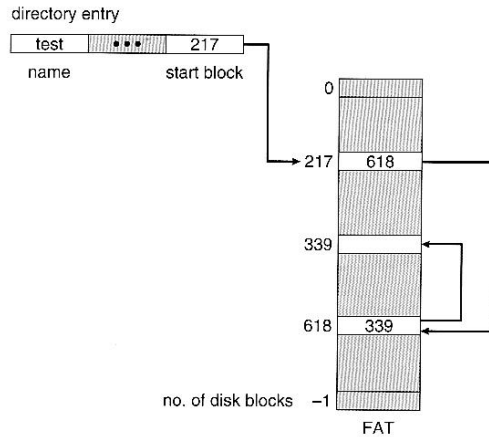


Figure 11.7 File-allocation table.

c) Indexed Allocation

- Indexed Allocation** combines all of the indexes(block numbers) for accessing each file into a common block (for that file).
- Each file will have a common block called the index block.

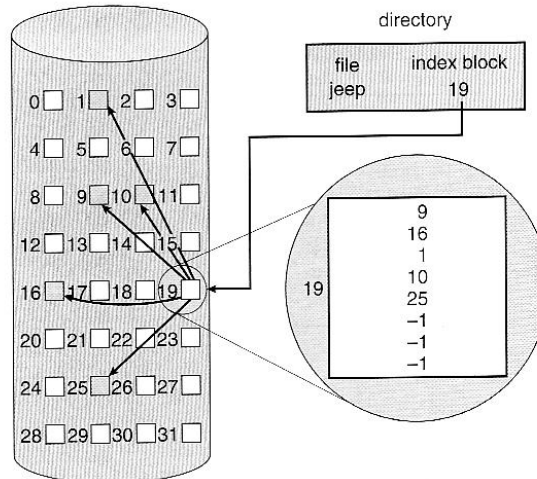


Figure 11.8 Indexed allocation of disk space.

- Some disk space is wasted (relative to linked lists or FAT tables) because an entire index block must be allocated for each file, regardless of how many data blocks the file contains. This leads to questions of how big the index block should be, and how it should be implemented. There are several approaches:

- **Linked Scheme** - An index block is one disk block, which can be read and written in a single disk operation. The first index block contains some header information, the first N block addresses, and if necessary a pointer to additional linked index blocks.
- **Multi-Level Index** - The first index block contains a set of pointers to secondary index blocks, which in turn contain pointers to the actual data blocks.
- **Combined Scheme** - This is the scheme used in UNIX inodes, in which the first 12 entries data block pointers are stored directly in the inode, and then singly, doubly, and triply indirect pointers provide access to more data blocks as needed. The advantage of this scheme is that for small files (files stored in less than 12 blocks), the data blocks are readily accessible (up to 48K with 4K block sizes); files up to about 4144K (using 4K blocks) are accessible with only a single indirect block (which can be cached), and huge files are still accessible using a relatively small number of disk accesses (larger in theory than can be addressed by a 32-bit address, which is why some systems have moved to 64-bit file pointers.)

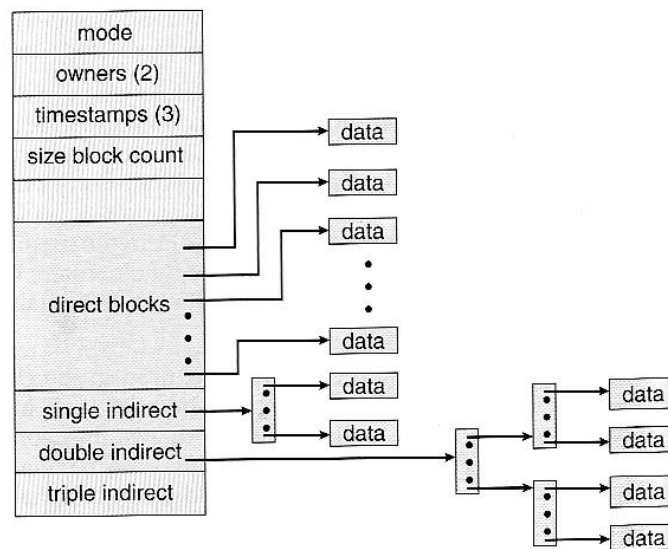


Figure 11.9 The UNIX inode.

Performance

- The optimal allocation method is different for sequential access files than for random access files, and is also different for small files than for large files.
- Some systems support more than one allocation method, which may require specifying how the file is to be used (sequential or random access) at the time it is allocated. Such systems also provide conversion utilities.
- Some systems have been known to use contiguous access for small files, and automatically switch to an indexed scheme when file sizes surpass a certain threshold.

- And of course some systems adjust their allocation schemes (e.g. block sizes) to best match the characteristics of the hardware for optimum performance.

10.11 Free-Space Management:

The space created after deleting the files can be reused. Another important aspect of disk management is keeping track of free space in memory. The list which keeps track of free space in memory is called the free-space list. To create a file, 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, is implemented in different ways as explained below.

a) Bit Vector

- Fast algorithms exist for quickly finding contiguous blocks of a given size
- One simple approach is to use a **bit vector**, in which each bit represents a disk block, set to 1 if free or 0 if allocated.

For example, consider a disk where blocks 2,3,4,5,8,9, 10,11, 12, 13, 17 and 18 are free, and the rest of the blocks are allocated. The free-space bit map would be
0011110011111100011

- Easy to implement and also very efficient in finding the first free block or ‘n’ consecutive free blocks on the disk.
- The down side is that a 40GB disk requires over 5MB just to store the bitmap.

b) Linked List

- A linked list can also be used to keep track of all free blocks.
- Traversing the list and/or finding a contiguous block of a given size are not easy, but fortunately are not frequently needed operations. Generally the system just adds and removes single blocks from the beginning of the list.
- The FAT table keeps track of the free list as just one more linked list on the table.

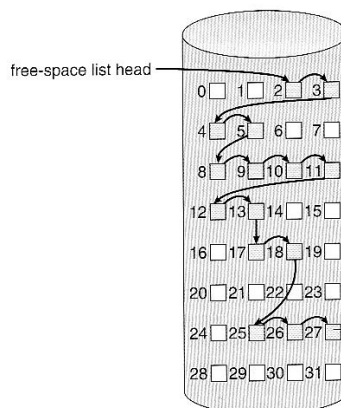


Figure 11.10 Linked free-space list on disk.

c) Grouping

- A variation on linked list free lists. It stores the addresses of n free blocks in the first free block. The first n-1 blocks are actually free. The last block contains the addresses of another n free blocks, and so on.
- The address of a large number of free blocks can be found quickly.

d) Counting

- When there are multiple contiguous blocks of free space then the system can keep track of the starting address of the group and the number of contiguous free blocks.
- Rather than keeping a list of n free disk addresses, we can keep the address of first free block and the number of free contiguous blocks that follow the first block.
- Thus the overall space is shortened. It is similar to the extent method of allocating blocks.

e) Space Maps (New)

- Sun's ZFS file system was designed for huge numbers and sizes of files, directories, and even file systems.
- The resulting data structures could be inefficient if not implemented carefully. For example, freeing up a 1 GB file on a 1 TB file system could involve updating thousands of blocks of free list bit maps if the file was spread across the disk.
- ZFS uses a combination of techniques, starting with dividing the disk up into (hundreds of) *metaslabs* of a manageable size, each having their own space map.
- Free blocks are managed using the counting technique, but rather than write the information to a table, it is recorded in a log-structured transaction record. Adjacent free blocks are also coalesced into a larger single free block.
- An in-memory space map is constructed using a balanced tree data structure, constructed from the log data.
- The combination of the in-memory tree and the on-disk log provide for very fast and efficient management of these very large files and free blocks.