MODULE 4: VIRTUAL MEMORY FILE-SYSTEM INTERFACE FILE-SYSTEM IMPLEMENTATION

MODULE 4: VIRTUAL MEMORY

4.1 Virtual Memory

• In many cases, the entire program is not needed.
For example:
→ Unusual error-conditions are almost never executed.
→ Arrays & lists are often allocated more memory than needed.
→ Certain options & features of a program may be used rarely.

• Benefits of executing a program that is only partially in memory.
➢ More programs could be run at the same time.
➢ Programmers could write for a large virtual-address space and need no longer use overlays.
➢ Less I/O would be needed to load/swap programs into memory, so each user program would run faster.

• Virtual Memory is a technique that allows the execution of processes that are not completely in memory (Figure 4.1).
➢ VM involves the separation of logical-memory as perceived by users from physical-memory.
➢ VM allows files and memory to be shared by several processes through page-sharing.
➢ Logical-address space can ∴ be much larger than physical-address space.

• Virtual-memory can be implemented by:
1) Demand paging and
2) Demand segmentation.

• The virtual (or logical) address-space of a process refers to the logical (or virtual) view of how a process is stored in memory.

• Physical-memory may be organized in page-frames and that the physical page-frames assigned to a process may not be contiguous.

• It is up to the MMU to map logical-pages to physical page-frames in memory.

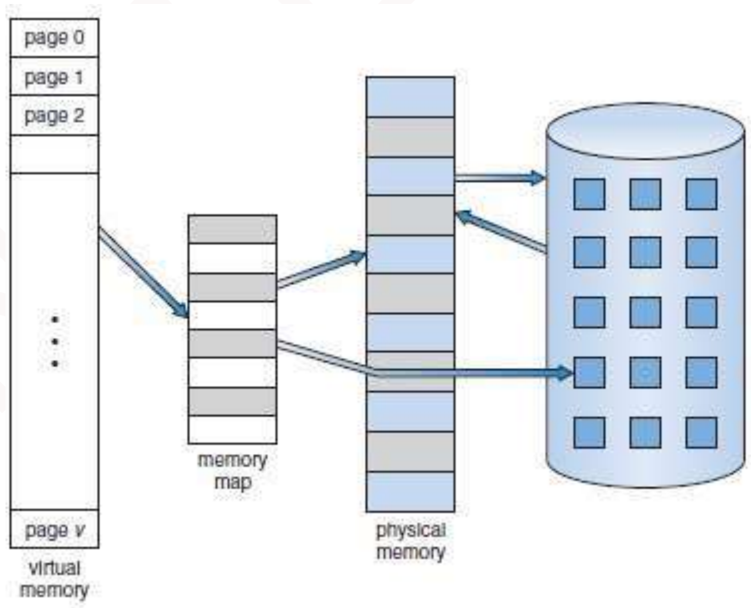


Figure 4.1 Diagram showing virtual memory that is larger than physical-memory

4.2 Demand Paging

• A demand-paging system is similar to a paging-system with swapping (Figure 4.2).
• Processes reside in secondary-memory (usually a disk).
• When we want to execute a process, we swap it into memory.
• Instead of swapping in a whole process, **lazy swapper** brings only those necessary pages into memory.

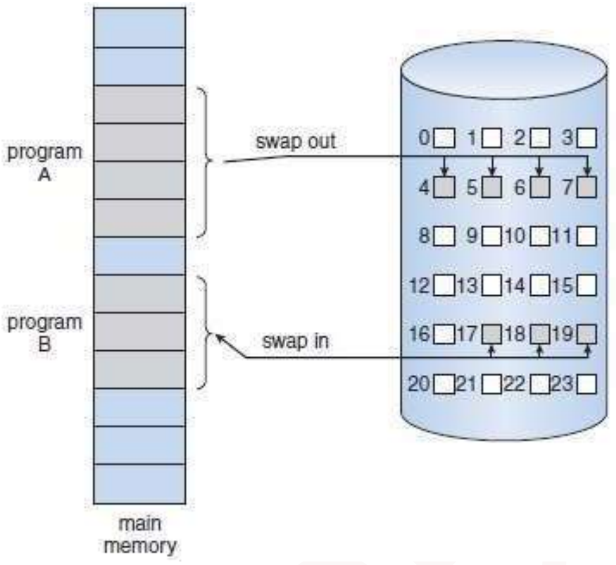


Figure 4.2 Transfer of a paged memory to contiguous disk space

4.2.1 Basic Concepts

• Instead of swapping in a whole process, the pager brings only those necessary pages into memory.
• Advantages:
1) Avoids reading into memory-pages that will not be used,
2) Decreases the swap-time and
3) Decreases the amount of physical-memory needed.
• The valid-invalid bit scheme can be used to distinguish between
→ pages that are in memory and
→ pages that are on the disk.
1) If the bit is set to **valid**, the associated page is both legal and in memory.
2) If the bit is set to **invalid**, the page either
→ is not valid (i.e. not in the logical-address space of the process) or
→ is valid but is currently on the disk (Figure 4.3).

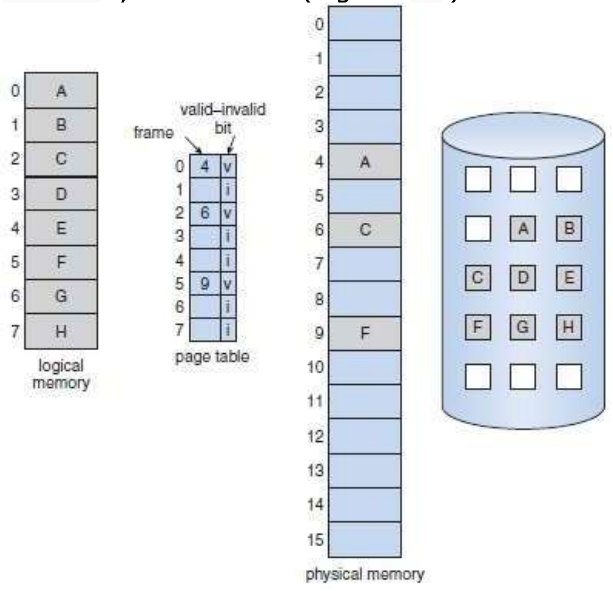


Figure 4.3 Page-table when some pages are not in main-memory

• A **page-fault** occurs when the process tries to access a page that was not brought into memory**.**
• Procedure for handling the page-fault (Figure 4.4):
1) Check an internal-table to determine whether the reference was a valid or an invalid memory access.
2) If the reference is invalid, we terminate the process.
If reference is valid, but we have not yet brought in that page, we now page it in.
3) Find a free-frame (by taking one from the free-frame list, for example).
4) Read the desired page into the newly allocated frame.
5) Modify the internal-table and the page-table to indicate that the page is now in memory.
6) Restart the instruction that was interrupted by the trap.

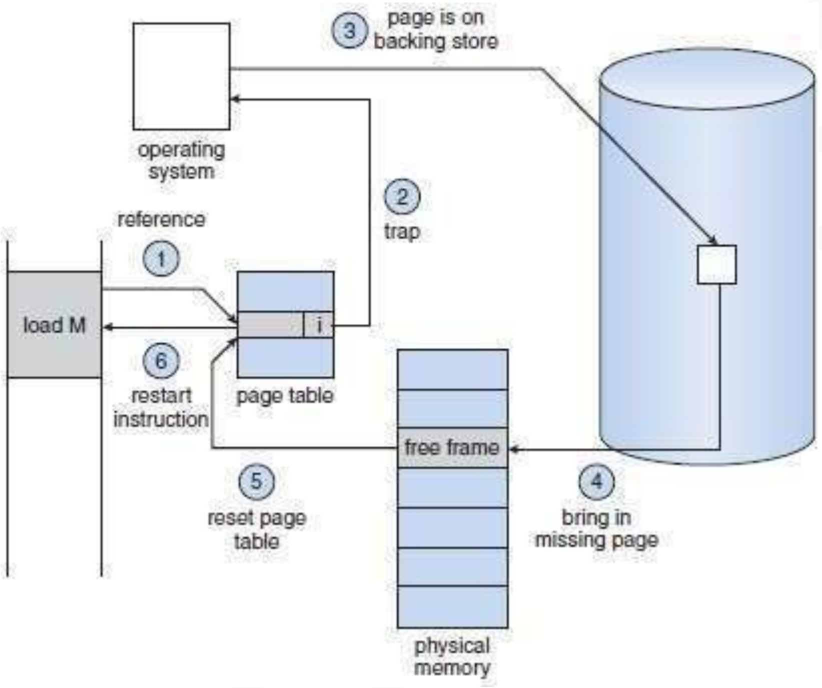


Figure 4.4 Steps in handling a page-fault

• **Pure demand paging:** Never bring pages into memory until required.
• Some programs may access several new pages of memory with each instruction, causing multiple page-faults and poor performance.
• Programs tend to have locality of reference, so this results in reasonable performance from demand paging.
• Hardware support:
1) Page-table
➢ Mark an entry invalid through a valid-invalid bit.
2) Secondary memory
➢ It holds pages that are not present in main-memory.
➢ It is usually a high-speed disk.
➢ It is known as the **swap device** (and the section of disk used for this purpose is known as swap space).

4.2.2 Performance

- Demand paging can significantly affect the performance of a computer-system.
- Let p be the probability of a page-fault ($0 \le p \le 1$).
  - if p = 0, no page-faults.
  - if p = 1, every reference is a fault.
- effective access time(EAT)=[(1 - p) *memory access]+ [p *page-fault time]
- A page-fault causes the following events to occur:
  1) Trap to the OS.
  2) Save the user-registers and process-state.
  3) Determine that the interrupt was a page-fault. '
  4) Check that the page-reference was legal and determine the location of the page on the disk.
  5) Issue a read from the disk to a free frame:
     a. Wait in a queue for this device until the read request is serviced.
     b. Wait for the device seek time.
     c. Begin the transfer of the page to a free frame.
  6) While waiting, allocate the CPU to some other user.
  7) Receive an interrupt from the disk I/O subsystem (I/O completed).
  8) Save the registers and process-state for the other user (if step 6 is executed).
  9) Determine that the interrupt was from the disk.
  10) Correct the page-table and other tables to show that the desired page is now in memory.
  11) Wait for the CPU to be allocated to this process again.
  12) Restore the user-registers, process-state, and new page-table, and then resume the interrupted instruction.

4.3 Copy-on-Write

- This technique allows the parent and child processes initially to share the same pages.
- If either process writes to a shared-page, a copy of the shared-page is created.
- For example:
  - ➢ Assume that the child process attempts to modify a page containing portions of the stack, with the pages set to be copy-on-write.
  - ➢ OS will then create a copy of this page, mapping it to the address space of the child process.
  - ➢ Child process will then modify its copied page & not the page belonging to the parent process.

4.4 Page Replacement

1) FIFO page replacement
2) Optimal page replacement
3) LRU page replacement (Least Recently Used)
4) LFU page replacement (Least Frequently Used)

4.4.1 Need for Page Replacement

• If we increase our degree of multiprogramming, we are over-allocating memory.
• While a user-process is executing, a page-fault occurs.
• The OS determines where the desired page is residing on the disk but then finds that there are no free frames on the free-frame list (Figure 4.5).
• The OS then could:
→ Terminate the user-process (Not a good idea).
→ Swap out a process, freeing all its frames, and reducing the level of multiprogramming.
→ Perform page replacement.

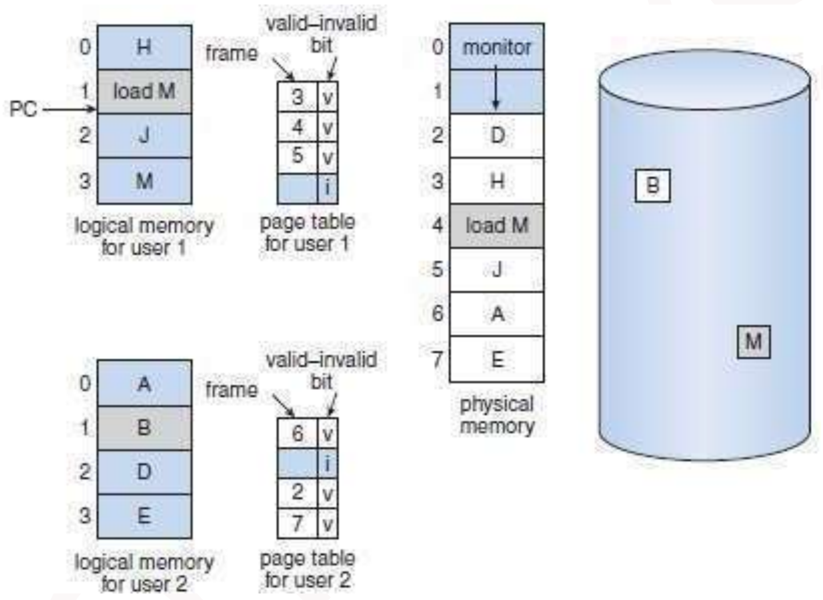


Figure 4.5 Need for page replacement

4.4.2 Basic Page Replacement

• Basic page replacement approach:
  → If no frame is free, we find one that is not currently being used and free it (Figure 4.6).

• Page replacement takes the following steps:
  1) Find the location of the desired page on the disk.
  2) Find a free frame:
    ¤ If there is a free frame, use it.
    ¤ If there is no free frame, use a page-replacement algorithm to select a victim-frame.
    ¤ Write the victim-frame to the disk; change the page and frame-tables accordingly.
  3) Read the desired page into the newly freed frame; change the page and frame-tables.
  4) Restart the user-process.

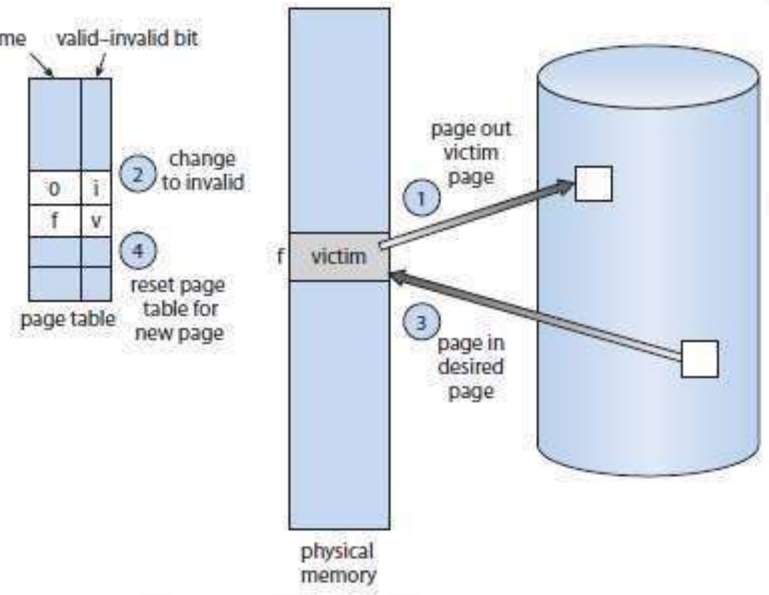


Figure 4.6 Page replacement

• Problem: If no frames are free, 2 page transfers (1 out & 1 in) are required. This situation
  → doubles the page-fault service-time and
  → increases the EAT accordingly.

Solution: Use a modify-bit (or dirty bit).

• Each page or frame has a modify-bit associated with the hardware.

• The **modify-bit** for a page is set by the hardware whenever any word is written into the page (indicating that the page has been modified).

• Working:
  1) When we select a page for replacement, we examine it's modify-bit.
  2) If the modify-bit =1, the page has been modified. So, we must write the page to the disk.
  3) If the modify-bit=0, the page has not been modified. So, we need not write the page to the disk, it is already there.

• Advantage:
  1) Can reduce the time required to service a page-fault.

• We must solve 2 major problems to implement demand paging:
  1) Develop a **Frame-allocation algorithm:**
  ➢ If we have multiple processes in memory, we must decide how many frames to allocate to each process.
  2) Develop a **Page-replacement algorithm:**
  ➢ We must select the frames that are to be replaced.

4.4.3 FIFO Page Replacement

• Each page is associated with the time when that page was brought into memory.
• When a page must be replaced, the oldest page is chosen.
• We use a **FIFO queue** to hold all pages in memory (Figure 4.7).
When a page must be replaced, we replace the page at the head of the queue
When a page is brought into memory, we insert it at the tail of the queue.
• Example: Consider the following references string with frames initially empty.

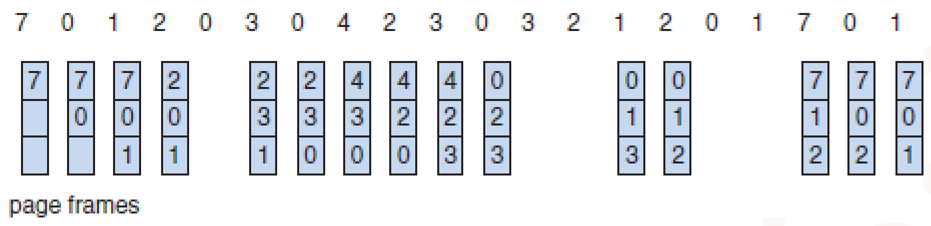


Figure 4.7 FIFO page-replacement algorithm

➢ The first three references(7, 0, 1) cause page-faults and are brought into these empty frames.
➢ The next reference(2) replaces page 7, because page 7 was brought in first.
➢ Since 0 is the next reference and 0 is already in memory, we have no fault for this reference.
➢ The first reference to 3 results in replacement of page 0, since it is now first in line.
➢ This process continues till the end of string.
➢ There are fifteen faults altogether.

• Advantage:
1) Easy to understand & program.
• Disadvantages:
1) Performance is not always good (Figure 4.8).
2) A bad replacement choice increases the page-fault rate (Belady's anomaly).
• For some algorithms, the page-fault rate may increase as the number of allocated frames increases. This is known as **Belady's anomaly**.
• Example: Consider the following reference string:
1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5
For this example, the number of faults for four frames (ten) is greater than the number of faults for three frames (nine)!

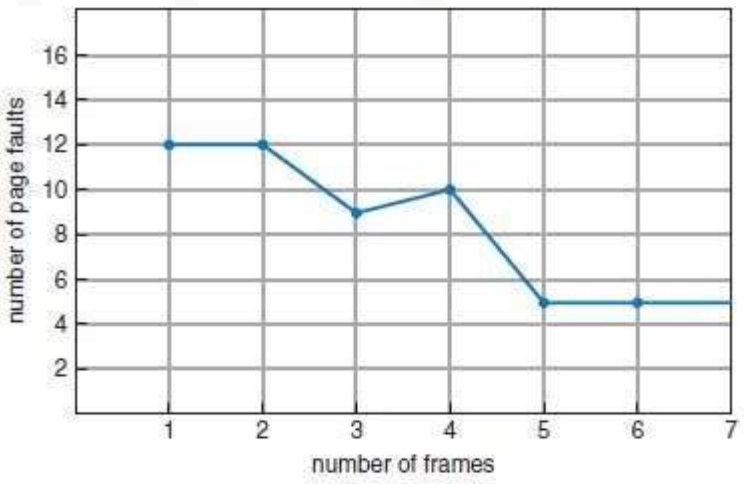


Figure 4.8 Page-fault curve for FIFO replacement on a reference string

4.4.4 Optimal Page Replacement

• Working principle: Replace the page that will not be used for the longest period of time (Figure 4.9).
• This is used mainly to solve the problem of Belady"s Anamoly.
• This has the lowest page-fault rate of all algorithms.
• Consider the following reference string:

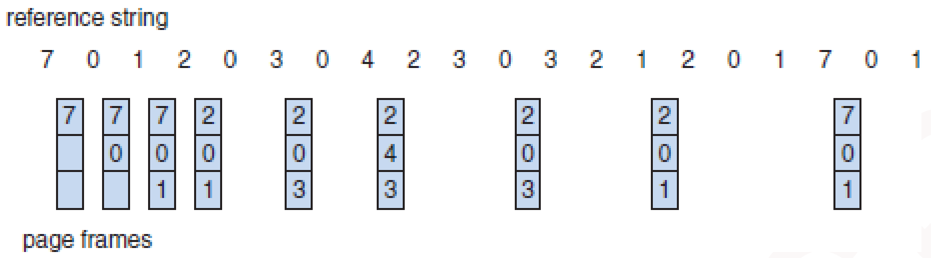


Figure 4.9 Optimal page-replacement algorithm

➢ The first three references cause faults that fill the three empty frames.
➢ The reference to page 2 replaces page 7, because page 7 will not be used until reference 18.
➢ The page 0 will be used at 5, and page 1 at 14.
➢ With only nine page-faults, optimal replacement is much better than a FIFO algorithm, which results in fifteen faults.

• Advantage:
1) Guarantees the lowest possible page-fault rate for a fixed number of frames.
• Disadvantage:
1) Difficult to implement, because it requires future knowledge of the reference string.

4.4.5 LRU Page Replacement

• The key difference between FIFO and OPT:

→ FIFO uses the time when a page was brought into memory.

→ OPT uses the time when a page is to be used.

• Working principle: Replace the page that has not been used for the longest period of time.

• Each page is associated with the time of that page's last use (Figure 4.10).

• Example: Consider the following reference string:

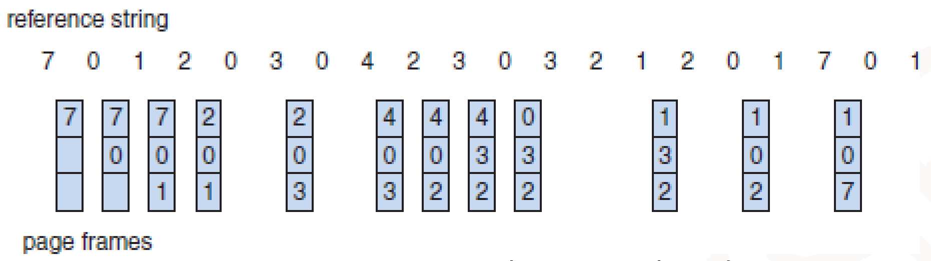


Figure 4.10 LRU page-replacement algorithm

➢ The first five faults are the same as those for optimal replacement.

➢ When the reference to page 4 occurs, LRU sees that of the three frames, page 2 was used least recently. Thus, the LRU replaces page 2.

➢ The LRU algorithm produces twelve faults.

• Two methods of implementing LRU:

1) Counters

➢ Each page-table entry is associated with a **time-of-use** field.

➢ A **counter(or logical clock)** is added to the CPU.

➢ The clock is incremented for every memory-reference.

➢ 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.

➢ We replace the page with the smallest time value.

2) Stack

➢ Keep a stack of page-numbers (Figure 4.11).

➢ Whenever a page is referenced, the page is removed from the stack and put on the top.

➢ The most recently used page is always at the top of the stack.
The least recently used page is always at the bottom.

➢ Stack is best implement by a doubly linked-list.

➢ Advantage:

1) Does not suffer from Belady's anomaly.

➢ Disadvantage:

1) Few computer systems provide sufficient h/w support for true LRU page replacement.

➢ Both LRU & OPT are called stack algorithms.

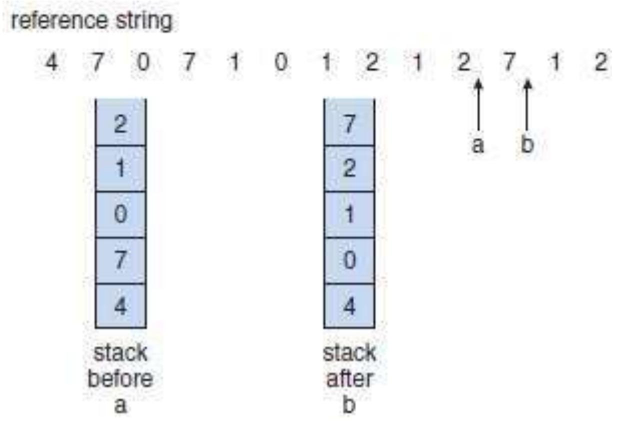


Figure 4.11 Use of a stack to record the most recent page references

4.4.6 LRU-Approximation Page Replacement

- Some systems provide a **reference bit** for each page.
- Initially, all bits are cleared(to 0) by the OS.
- As a user-process executes, the bit associated with each page referenced is set (to 1) by the hardware.
- By examining the reference bits, we can determine
  - → which pages have been used and
  - → which have not been used.
- This information is the basis for many page-replacement algorithms that approximate LRU replacement.

4.4.6.1 Additional-Reference-Bits Algorithm

- We can gain additional **ordering information** by recording the reference bits at regular intervals.
- A 8-bit byte is used for each page in a table in memory.
- At regular intervals, a timer-interrupt transfers control to the OS.
- The OS shifts the reference bit for each page into the high-order bit of its 8-bit byte.
- These 8-bit shift registers contain the history of page use, for the last eight time periods.
- Examples:
  - 00000000 - This page has not been used in the last 8 time units (800 ms).
  - 11111111 - Page has been used every time unit in the past 8 time units.
  - 11000100 has been used more recently than 01110111.
- The page with the lowest number is the LRU page, and it can be replaced.
- If numbers are equal, FCFS is used

4.4.6.2 Second-Chance Algorithm

- The number of bits of history included in the shift register can be varied to make the updating as fast as possible.
- In the extreme case, the number can be reduced to zero, leaving only the reference bit itself. This algorithm is called the **second-chance algorithm.**
- Basic algorithm is a FIFO replacement algorithm.
- Procedure:
  - ➢ When a page has been selected, we inspect its reference bit.
  - ➢ If reference bit=0, we proceed to replace this page.
  - ➢ If reference bit=1, we give the page a second chance & move on to select next FIFO page.
  - ➢ When a page gets a second chance, its reference bit is cleared, and its arrival time is reset.

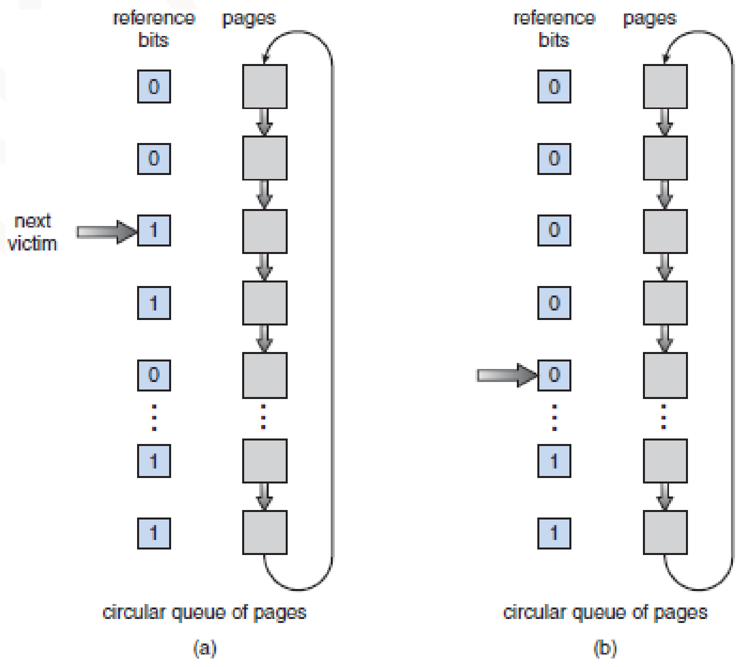


Figure 4.12 Second-chance (clock) page-replacement algorithm

• A circular queue can be used to implement the second-chance algorithm (Figure 4.12).
 ➢ A pointer (that is, a hand on the clock) 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.

4.4.6.3 Enhanced Second-Chance Algorithm

• We can enhance the second-chance algorithm by considering
 1) Reference bit and
 2) modify-bit.

• We have following 4 possible classes:
 1) **(0, 0)** neither recently used nor modified -best page to replace.
 2) **(0, 1)** not recently used hut modified-not quite as good, because the page will need to be written out before replacement.
 3) **(1, 0)** recently used but clean-probably will be used again soon.
 4) **(1, 1)** recently used and modified -probably will be used again soon, and the page will be need to be written out to disk before it can be replaced.

• Each page is in one of these four classes.
• When page replacement is called for, we examine the class to which that page belongs.
• We replace the first page encountered in the lowest nonempty class.

4.4.7 Counting-Based Page Replacement

1) LFU page-replacement algorithm
 ➢ Working principle: The page with the smallest count will be replaced.
 ➢ The reason for this selection is that an actively used page should have a large reference count.
 ➢ Problem:
  When a page is used heavily during initial phase of a process but then is never used again. Since it was used heavily, it has a large count and remains in memory even though it is no longer needed.
 Solution:
  Shift the counts right by 1 bit at regular intervals, forming an exponentially decaying average usage count.

2) MFU (Most Frequently Used) page-replacement algorithm
 ➢ Working principle: The page with the smallest count was probably just brought in and has yet to be used.

4.5 Allocation of Frames

4.5.1 Minimum Number of Frames

• We must also allocate at least a minimum number of frames. One reason for this is performance.
• As the number of frames allocated to each process decreases, the page-fault rate increases, slowing process execution.
• In addition, when a page-fault occurs before an executing instruction is complete, the instruction must be restarted.
• The minimum number of frames is defined by the computer architecture.

4.5.2 Allocation Algorithms

1) Equal Allocation

➢ We split m frames among n processes is to give everyone an equal share, m/n frames. (For ex: if there are 93 frames and five processes, each process will get 18 frames. The three leftover frames can be used as a free-frame buffer pool).

2) Proportional Allocation

➢ We can allocate available memory to each process according to its size.

• In both 1 & 2, the allocation may vary according to the multiprogramming level.
• If the multiprogramming level is increased, each process will lose some frames to provide the memory needed for the new process.
• Conversely, if the multiprogramming level decreases, the frames that were allocated to the departed process can be spread over the remaining processes.

4.5.3 Global versus Local Allocation

Global Replacement	**Local Replacement**
Allows a process to a replacement frame from the set of all frames.	Each process selects from only its own set of allocated frames.
A process may happen to select only frames allocated to other processes, thus increasing the number of frames allocated to it.	Number of frames allocated to a process does not change.
Disadvantage: A process cannot control its own page-fault rate.	Disadvantage: Might prevent a process by not making available to it other less used pages of memory.
Advantage: Results in greater system throughput.	

4.6 Thrashing

• If a process does not have "enough" pages, the page-fault rate is very high. This leads to:
→ low CPU utilization
→ operating system thinks that it needs to increase the degree of multiprogramming
→ another process added to the system.

• If the number of frames allocated to a low-priority process falls below the minimum number required, it must be suspended.

• A process is thrashing if it is spending more time paging than executing.

4.6.1 Cause of Thrashing

• Thrashing results in severe performance-problems (Figure 4.13).

• The thrashing phenomenon:
➢ As processes keep faulting, they queue up for the paging device, so CPU utilization decreases
➢ The CPU scheduler sees the decreasing CPU utilization and increases the degree of multiprogramming as a result.
➢ The new process causes even more page-faults and a longer queue!

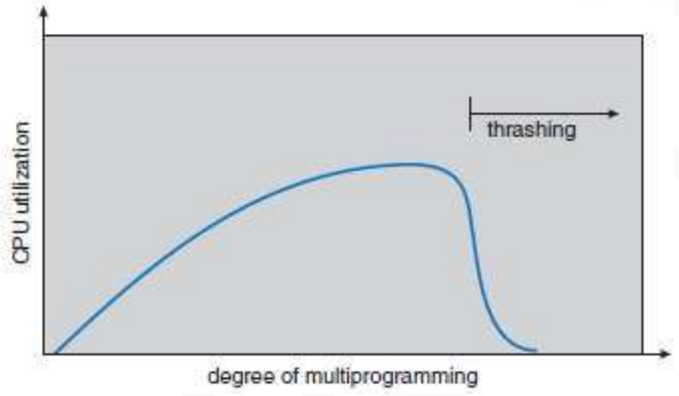


Figure 4.13 Thrashing

• Methods to avoid thrashing:

1) Use Local Replacement
➢ If one process starts thrashing, it cannot
→ steal frames from another process and
→ cause the latter to thrash as well.

2) We must provide a process with as many frames as it needs. This approach defines the **locality model** of process execution.
➢ Locality Model states that
As a process executes, it moves from locality to locality.
➢ A locality is a set of pages that are actively used together.
➢ A program may consist of several different localities, which may overlap.

EXERCISE PROBLEMS

1) Consider following page reference string:

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

For a memory with 3 frames, how many page faults would occur for

(i) LRU algorithm
(ii) FIFO algorithm and
(iii) Optimal page replacement algorithm?

Which is the most efficient among them?

Solution:

(i) LRU with 3 frames:

Frames	7	0	1	2	0	3	0	4	2	3	0	3	2	1	2	0	1	7	0	1
1	7	7	7	2	2	2	2	4	4	4	0	0	0	1	1	1	1	1	1	1
2		0	0	0	0	0	0	0	0	3	3	3	3	3	3	0	0	0	0	0
3			1	1	1	3	3	3	2	2	2	2	2	2	2	2	2	7	7	7
No. of Page faults	√	√	√	√		√		√	√	√	√			√		√		√		

No of page faults=12

(ii) FIFO with 3 frames:

Frames	7	0	1	2	0	3	0	4	2	3	0	3	2	1	2	0	1	7	0	1
1	7	0	1	2	2	3	0	4	2	3	0	0	0	1	2	2	2	7	0	1
2		7	0	1	1	2	3	0	4	2	3	3	3	0	1	1	1	2	7	0
3			7	0	0	1	2	3	0	4	2	2	2	3	0	0	0	1	2	7
No. of Page faults	√	√	√	√		√	√	√	√	√	√			√	√			√	√	√

No of page faults=15

(iii) Optimal with 3 frames:

Frames	7	0	1	2	0	3	0	4	2	3	0	3	2	1	2	0	1	7	0	1
1	7	7	7	2	2	2	2	2	2	2	2	2	2	2	2	2	2	7	7	7
2		0	0	0	0	0	0	4	4	4	0	0	0	0	0	0	0	0	0	0
3			1	1	1	3	3	3	3	3	3	3	3	1	1	1	1	1	1	1
No. of Page faults	√	√	√	√		√		√			√			√				√		

No of page faults=9

Conclusion: The optimal page replacement algorithm is most efficient among three algorithms, as it has lowest page faults i.e. 9.

2) Consider the following page reference string

1, 2, 3, 4, 2, 1, 5, 6, 2, 1, 2, 3, 7, 6, 3, 2, 1, 2, 3, 6

How many page fault would occur for the following page replacement algorithms assuming 3 and 5 frames.

(i) LRU

(ii) Optimal

Solution:

LRU with 3 frames:

Frames	1	2	3	4	2	1	5	6	2	1	2	3	7	6	3	2	1	2	3	6
1	1	1	1	4	4	4	5	5	5	1	1	1	7	7	7	2	2	2	2	2
2		2	2	2	2	2	2	6	6	6	6	3	3	3	3	3	3	3	3	3
3			3	3	3	1	1	1	2	2	2	2	2	6	6	6	1	1	1	6
No. of Page faults	√	√	√	√		√	√	√	√	√		√	√	√		√	√			√

No of page faults= 15

LRU with 5 frames:

Frames	1	2	3	4	2	1	5	6	2	1	2	3	7	6	3	2	1	2	3	6
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
2		2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2
3			3	3	3	3	3	6	6	6	6	6	6	6	6	6	6	6	6	6
4				4	4	4	4	4	4	4	4	3	3	3	3	3	3	3	3	3
5							5	5	5	5	5	5	7	7	7	7	7	7	7	7
No. of Page faults	√	√	√	√			√	√				√	√							

No of page faults= 8

Optimal with 3 frames:

Frames	1	2	3	4	2	1	5	6	2	1	2	3	7	6	3	2	1	2	3	6
1	1	1	1	1	1	1	1	1	1	1	1	3	3	3	3	3	3	3	3	6
2		2	2	2	2	2	2	2	2	2	2	2	7	7	7	2	2	2	2	2
3			3	4	4	4	5	6	6	6	6	6	6	6	6	6	1	1	1	1
No. of Page faults	√	√	√	√			√	√				√	√			√	√			√

No of page faults= 11

Optimal with 5 frames:

Frames	1	2	3	4	2	1	5	6	2	1	2	3	7	6	3	2	1	2	3	6
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
2		2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2
3			3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3
4				4	4	4	4	6	6	6	6	6	6	6	6	6	6	6	6	6
5							5	5	5	5	5	5	7	7	7	7	7	7	7	7
No. of Page faults	√	√	√	√			√	√					√							

No of page faults= 7

3) For the following page reference, calculate the page faults that occur using FIFO and LRU for 3 and 4 page frames respectively

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

Solution:

(i) LRU with 3 frames:

Frames	5	4	3	2	1	4	3	5	4	3	2	1	5
1	5	5	5	2	2	2	3	3	3	3	3	3	5
2		4	4	4	1	1	1	5	5	5	2	2	2
3			3	3	3	4	4	4	4	4	4	1	1
No. of Page faults	√	√	√	√	√	√	√	√			√	√	√

No of page faults=11

(ii) LRU with 4 frames:

Frames	5	4	3	2	1	4	3	5	4	3	2	1	5
1	5	5	5	5	1	1	1	1	1	1	2	2	2
2		4	4	4	4	4	4	4	4	4	4	4	5
3			3	3	3	3	3	3	3	3	3	3	3
4				2	2	2	2	5	5	5	5	1	1
No. of Page faults	√	√	√	√	√			√			√	√	√

No of page faults=9

(iii) LRU with 4 frames::

Frames	5	4	3	2	1	4	3	5	4	3	2	1	5
1	5	4	3	2	1	4	3	5	5	5	2	1	1
2		5	4	3	2	1	4	3	3	3	5	2	2
3			5	4	3	2	1	4	4	4	3	5	5
No. of Page faults	√	√	√	√	√	√	√	√			√	√	

No of page faults=10

(iv) FIFO with 4 frames:

Frames	5	4	3	2	1	4	3	5	4	3	2	1	5
1	5	4	3	2	1	1	1	5	4	3	2	1	5
2		5	4	3	2	2	2	1	5	4	3	2	1
3			5	4	3	3	3	2	1	5	4	3	2
4				5	4	4	4	3	2	1	5	4	3
No. of Page faults	√	√	√	√	√			√	√	√	√	√	√

No of page faults=11

4) What is Belady's anomaly? Explain with an example.

Solution:

Belady's Anomaly:

"On increasing the number of page frames, the no. of page faults do not necessarily decrease, they may also increase".

• Example: Consider the following reference string when number of frame used is 3 and 4:

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

(i) FIFO with 3 frames:

Frames	1	2	3	4	1	2	5	1	2	3	4	5
1	1	2	3	4	1	2	5	5	5	3	4	4
2		1	2	3	4	1	2	2	2	5	3	3
3			1	2	3	4	1	1	1	2	5	5
No. of Page faults	√	√	√	√	√	√	√			√	√	

No. of page faults=9

(ii) FIFO with 4 frames:

Frames	1	2	3	4	1	2	5	1	2	3	4	5
1	1	2	3	4	4	4	5	1	2	3	4	5
2		1	2	3	3	3	4	5	1	2	3	4
3			1	2	2	2	3	4	5	1	2	3
4				1	1	1	2	3	4	5	1	2
No. of Page faults	√	√	√	√			√	√	√	√	√	√

No. of page faults=10

Conclusion: With 3 frames, No. of page faults=9.
With 4 frames, No. of page faults=10.
Thus, Belady's anomaly has occurred, when no. of frames are increased from 3 to 4.

MODULE 4 (CONT.): FILE-SYSTEM INTERFACE

4.7 File Concepts

• A **file** is a named collection of related info. on secondary-storage.
• Commonly, file represents
  → program and
  → data.
• Data in file may be
  → numeric
  → alphabetic or
  → binary.
• Four types of file:
  1) **Text file**: sequence of characters organized into lines.
  2) **Source file**: sequence of subroutines & functions.
  3) **Object file**: sequence of bytes organized into blocks.
  4) **Executable file**: series of code sections.

4.7.1 File Attributes

1) Name
➢ The only information kept in human-readable form.
2) Identifier
➢ It is a unique number which identifies the file within file-system.
➢ It is in non-human-readable form.
3) Type
➢ It is used to identify different types of files.
4) Location
➢ It is a pointer to
  → device and
  → location of file.
5) Size
➢ Current-size of file in terms of bytes, words, or blocks.
➢ It also includes maximum allowed size.
6) Protection
➢ Access-control info. determines who can do
  → reading
  → writing and
  → executing.
7) Time, date, & user identification
➢ These info. can be kept for
  → creation
  → last modification and
  → last use.
➢ These data can be useful for
  → protection
  → security and
  → usage monitoring.

• Information about files are kept in the **directory-structure**, which is maintained on the disk.

4.7.2 File Operations

1) Creating a file

➢ Two steps are:
- i) Find the space in the file-system for the file.
- ii) An entry for the new file is made in the directory.

2) Writing a file

➢ Make a system-call specifying both
- → file-name and
- → info. to be written to the file.

➢ The system searches the directory to find the file's location. (The system keeps a write-pointer(wp) to the location in the file where the next write is to take place).

➢ The write-pointer must be updated whenever a write-operation occurs.

3) Reading a file

➢ Make a system-call specifying both
- → file-name and
- → location of the next block of the file in the memory.

➢ The system searches the directory to find the file's location. (The system keeps a read-pointer(rp) to the location in the file where the next read is to take place).

➢ The read-pointer must be updated whenever a read-operation occurs.

➢ Same pointer (rp & wp) is used for both read- & write-operations. This results in
- → saving space and
- → reducing system-complexity.

4) Repositioning within a file

➢ Two steps are:
- i) Search the directory for the appropriate entry.
- ii) Set the current-file-position to a given value.

➢ This file-operation is also known as **file seek**.

5) Deleting a file

➢ Two steps are:
- i) Search the directory for the named-file.
- ii) Release all file-space and erase the directory-entry.

6) Truncating a file

➢ The contents of a file are erased but its attributes remain unchanged.

➢ Only file-length attribute is set to zero.

(Most of the above file-operations involve searching the directory for the entry associated with the file. To avoid this constant searching, many systems require that an „open' system-call be used before that file is first used).

• The OS keeps a small table which contains info. about all open files (called **open-file table**).

• If a file-operation is requested, then
- → file is specified via an index into open-file table
- → so no searching is required.

• If the file is no longer actively used, then
- → process closes the file and
- → OS removes its entry in the open-file table.

• Two levels of internal tables:

1) Per-process Table

➢ Tracks all files that a process had opened.

➢ Includes access-rights to
- → file and
- → accounting info.

➢ Each entry in the table in turn points to a system-wide table

2) System-wide Table

➢ Contains process-independent info. such as
- → file-location on the disk
- → file-size and
- → access-dates.

• Information associated with an open file:

1) File-pointer

➢ Used by the system to keep track of last read-write location.

2) File-open Count

➢ The counter

→ tracks the no. of opens & closes and

→ reaches zero on the last close.

3) Disk Location of the File

➢ Location-info is kept in memory to avoid having to read it from disk for each operation.

4) Access Rights

➢ Each process opens a file in an access-mode (read, write or execute).

• **File locks** allow one process to

→ lock a file and

→ prevent other processes from gaining access to locked-file**.**

Shared Lock	**Exclusive Lock**
Similar to a reader lock.	Behaves like a writer lock.
Several processes can acquire the lock concurrently.	Only one process at a time can acquire the lock.

Mandatory	**Advisory**
OS will prevent any other process from accessing the locked-file.	OS will not prevent other process from accessing the locked-file.
OS ensures locking integrity.	It is up to software-developers to ensure that locks are appropriately acquired and released.
Used by windows OS.	Used by UNIX systems.

4.7.3 File Types

• Common technique for implementing file-types: Include the type as part of the file-name.

• Two parts of file-name (Figure 4.14):

1) Name and 2) Extension

• The system uses the extension to indicate

→ type of file and

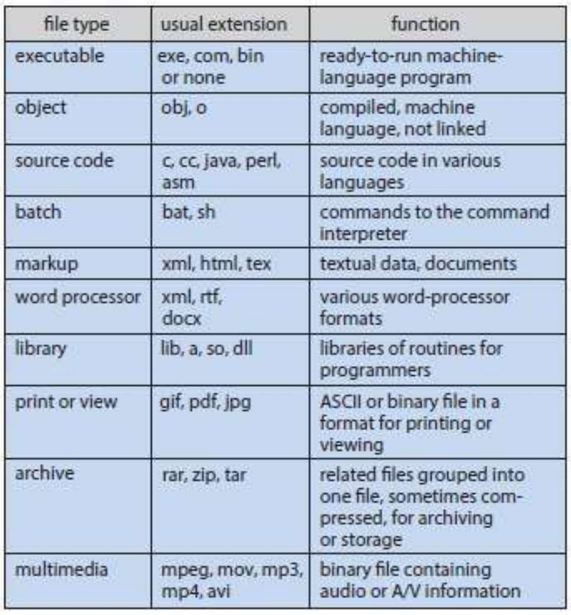
→ type of operations (read or write).

• Example:

→ Only a file with a .com, .exe, or .bat extension can be executed.

→ .com and .exe are two forms of binary executable files.

→ .bat file is a batch file containing, in ASCII format, commands to the OS.



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

Figure 4.14 Common file types

4.7.4 File Structure

- File types can be used to indicate the internal structure of the file.
- Disadvantage of supporting multiple file structures: Large size.
- All OSs must support at least one structure: an executable file
- In Mac OS, file contains 2 parts:
  - **1) Resource fork**: contains info. of interest to the user.
  - **2) Data fork**: contains program-code or data.
- Too few structures make programming inconvenient.
- Too many structures make programmer confusion.

4.7.5 Internal File Structure

- Locating an offset within a file can be complicated for the OS.
- Disk-systems typically have a well-defined block-size.
- All disk I/0 is performed in units of one block (physical record), and all blocks are the same size.
- Problem: It is unlikely that physical-record size will exactly match length of desired logical-record.
  Solution: **Packing** a number of logical-records into physical-blocks.
- Following things determine how many logical-records are in each physical-block:
  - → logical-record size
  - → physical-block size and
  - → packing technique.
- The packing can be done either by
  - → user's application program or
  - → OS.
- Disadvantage of packing:
  - → All file-systems suffer from internal fragmentation (the larger the block size, the greater the internal fragmentation).

4.8 Access Methods

4.8.1 Sequential Access

• This is based on a tape model of a file.
• This works both on
  → sequential-access devices and
  → random-access devices.
• Info. in the file is processed in order (Figure 4.15).
  For ex: editors and compilers
• Reading and writing are the 2 main operations on the file.
• File-operations:
  1) read next
  ➢ This is used to
    → read the next portion of the file and
    → advance a file-pointer, which tracks the I/O location.
  2) write next
  ➢ This is used to
    → append to the end of the file and
    → advance to the new end of file.

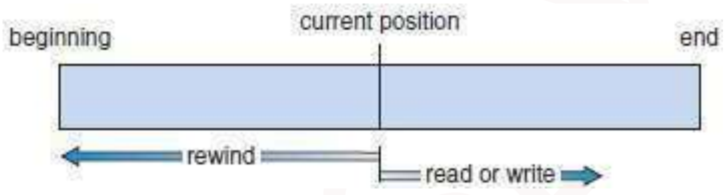
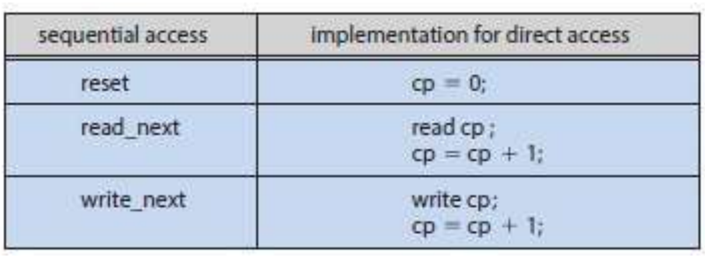


Figure 4.15 Sequential-access file

4.8.2 Direct Access (Relative Access)

• This is based on a disk model of a file (since disks allow random access to any file-block).
• A file is made up of fixed length logical records**.**
• Programs can read and write records rapidly in no particular order.
• Disadvantages:
  1) Useful for immediate access to large amounts of info.
  2) Databases are often of this type.
• File-operations include a relative block-number as parameter.
• The **relative block-number** is an index relative to the beginning of the file.
• File-operations (Figure 4.16):
    1) read n
    2) write n
      where n is the block-number
• Use of relative block-numbers:
  → allows OS to decide where the file should be placed and
  → helps to prevent user from accessing portions of file-system that may not be part of his file.



sequential access	implementation for direct access
reset	cp = 0;
read_next	read cp ; cp = cp + 1;

write_next	write cp;
	cp = cp + 1;

Figure 4.16 Simulation of sequential access on a direct-access file

4.8.3 Other Access Methods

• These methods generally involve constructing a **file-index**.
• The index contains pointers to the various blocks (like an index in the back of a book).
• To find a record in the file(Figure 4.17):
  1) First, search the index and
  2) Then, use the pointer to
    → access the file directly and
    → find the desired record.
• Problem: With large files, the index-file itself may become too large to be kept in memory.
Solution: Create an index for the index-file. (The primary index-file may contain pointers to secondary index-files, which would point to the actual data-items).

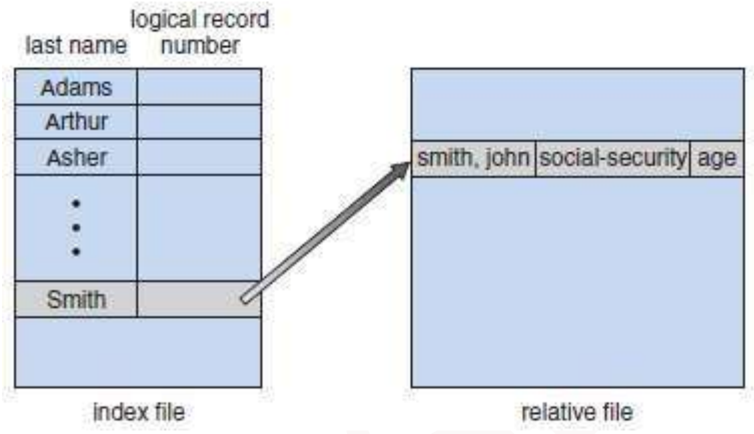


Figure 4.17 Example of index and relative files

4.9 Directory and Disk Structure

1) Single level directory
2) Two level directory
3) Tree structured directories
4) Acyclic-graph directories
5) General graph directory

4.9.1 Storage Structure

• A storage-device can be used in its entirety for a file-system.
• The storage-device can be split into 1 or more partitions (known as **slices** or **minidisk**).
• Any entity containing a file-system is known as a **volume**.
• The volume may be
→ a subset of a device or
→ a whole device.
• Each volume must also contain info. about the files in the system. This info. is kept in entries in a **device directory(** or volume table of contents).
• Device directory (or directory) records following info. for all files on that volume (Figure 4.18):
→ name
→ location
→ size and
→ type.

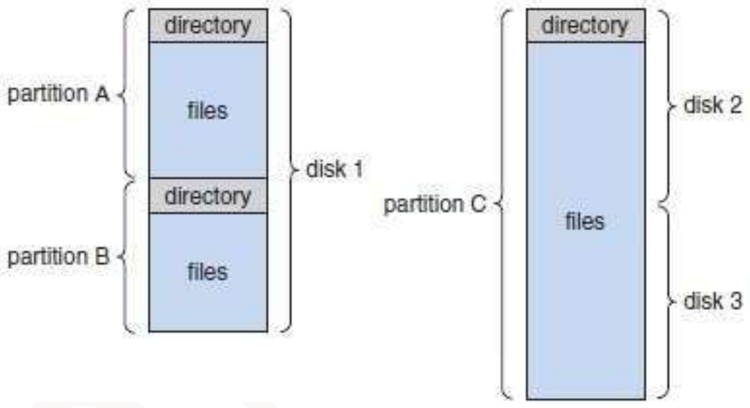


Figure 4.18: A typical file-system organization

4.9.2 Directory Overview

• Operations 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.

2) Create a File
➢ We need to be able to create and add new files to the directory.

3) Delete a File
➢ When a file is no longer needed, we want to be able to remove it from the directory.

4) List a Directory
➢ We need to be able to
→ list the files in a directory and
→ list the contents of the directory-entry for each file.

5) Rename a File
➢ Because the name of a file represents its contents to its users, we must be able to change the name when the contents or use of the file changes.

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.

4.9.3 Single Level Directory

• All files are contained in the same directory (Figure 4.19).
• Disadvantages (Limitations):
  1) Naming problem: All files must have unique names.
  2) Grouping problem: Difficult to remember names of all files, as number of files increases.

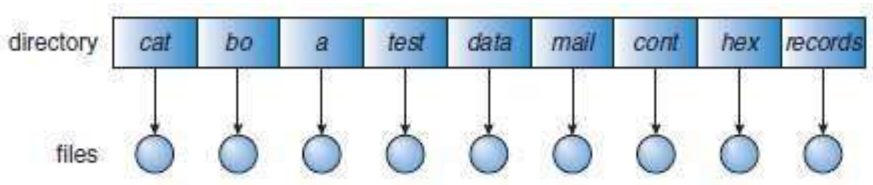


Figure 4.19 Single-level directory

4.9.4 Two Level Directory

• A separate directory for each user.
• Each user has his own UFD (user file directory).
• The UFDs have similar structures.
• Each UFD lists only the files of a single user.
• When a user job starts, the system's MFD is searched (MFD=master file directory).
• The MFD is indexed by user-name.
• Each entry in MFD points to the UFD for that user (Figure 4.20).

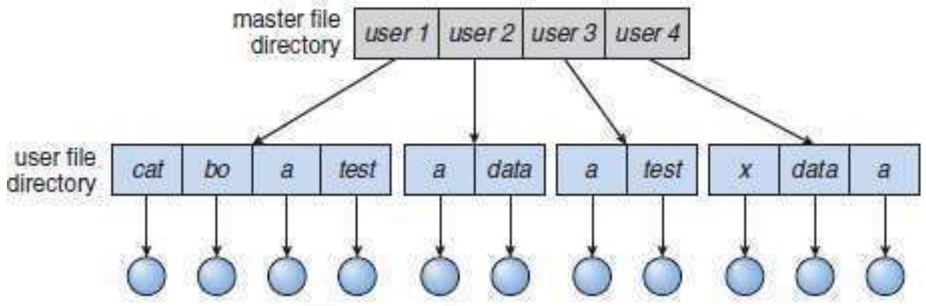


Figure 4.20 Two-level directory-structure

• To create a file for a user,
  the OS searches only that user's UFD to determine whether another file of that name exists.
• To delete a file,
  the OS limits its search to the local UFD. (Thus, it cannot accidentally delete another user's file that has the same name).
• Advantages:
  1) No filename-collision among different users.
  2) Efficient searching.
• Disadvantage:
  1) Users are isolated from one another and can't cooperate on the same task.

4.9.5 Tree Structured Directories

• Users can create their own subdirectories and organize files (Figure 4.21).
• A tree is the most common directory-structure.
• The tree has a root directory.
• Every file in the system has a unique path-name.

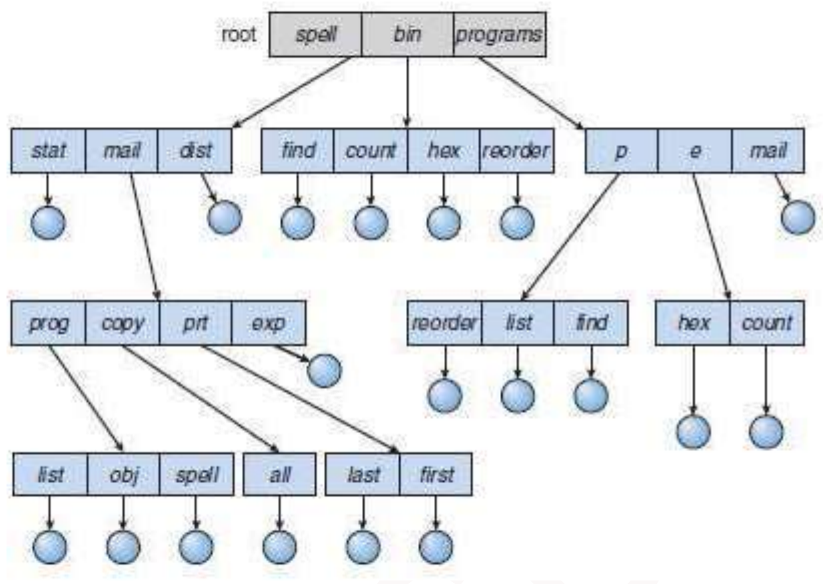


Figure 4.21 Tree-structured directory-structure

• A directory contains a set of files (or subdirectories).
• A directory is simply another file, but it is treated in a special way.
• In each directory-entry, one bit defines as
  file (0) or
  subdirectory (1).
• Path-names can be of 2 types:
• Two types of path-names:
  1) **Absolute path-name** begins at the root.
  2) **Relative path-name** defines a path from the current directory.
• How to delete directory?
  1) To delete an **empty directory**:
    → Just delete the directory.
  2) To delete a **non-empty directory**:
    → First, delete all files in the directory.
    → If any subdirectories exist, this procedure must be applied recursively to them.
• Advantage:
  1) Users can be allowed to access the files of other users.
• Disadvantages:
  1) A path to a file can be longer than a path in a two-level directory.
  2) Prohibits the sharing of files (or directories).

4.9.6 Acyclic Graph Directories

• The directories can share subdirectories and files (Figure 4.22).
  (An **acyclic graph** means a graph with no cycles).
• The same file (or subdirectory) may be in 2 different directories.
• Only one shared-file exists, so any changes made by one person are immediately visible to the other.

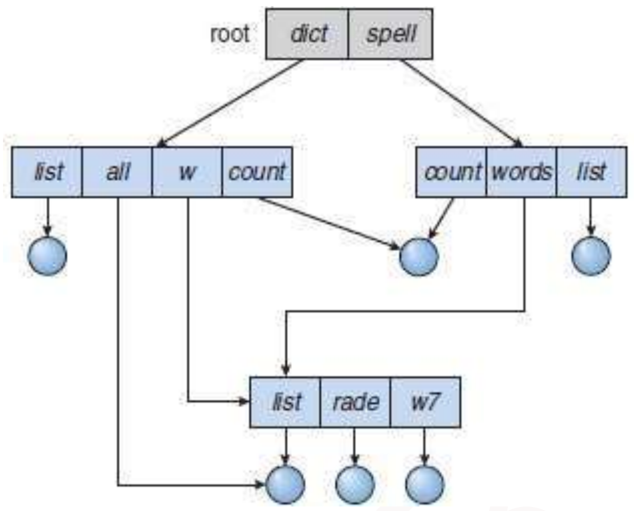


Figure 4.22 Acyclic-graph directory-structure

• Two methods to implement shared-files(or subdirectories):
  1) Create a new directory-entry called a link.
    A link is a pointer to another file (or subdirectory).
  2) Duplicate all info. about shared-files in both sharing directories.
• Two problems:
  1) A file may have multiple absolute path-names.
  2) Deletion may leave dangling-pointers to the non-existent file.

Solution to deletion problem:
  1) Use backpointers: Preserve the file until all references to it are deleted.
  2) With symbolic links, remove only the link, not the file. If the file itself is deleted, the link can be removed.

4.9.7 General Graph Directory

• Problem: If there are cycles, we want to avoid searching components twice (Figure 4.23).
  Solution: Limit the no. of directories accessed in a search.

• Problem: With cycles, the reference-count may be non-zero even when it is no longer possible to refer to a directory (or file). (A value of 0 in the reference count means that there are no more references to the file or directory, and the file can be deleted).
Solution: Garbage-collection scheme can be used to determine when the last reference has been deleted.

• Garbage collection involves

1) First pass
   → traverses the entire file-system and
   → marks everything that can be accessed.

2) A second pass collects everything that is not marked onto a list of free-space.

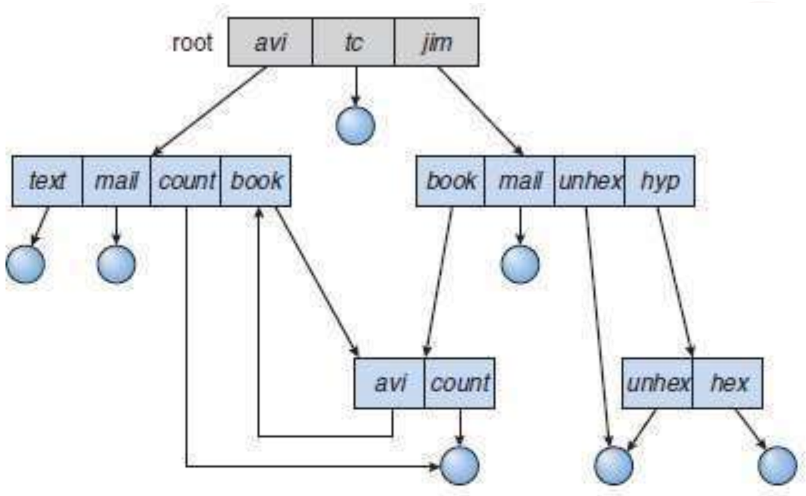


Figure 4.23 General graph directory

4.10 File System Mounting

• A file-system must be mounted before it can be available to processes on the system (Figure 4.24).
• **Mount-point** is the location in the file-structure where the file-system is to be attached.
• Procedure:
  1) OS is given
    → name of the device and
    → mount-point (Figure 4.25).
  2) OS verifies that the device contains a valid file-system.
  3) OS notes in its directory-structure that a file-system is mounted at specified mount-point.

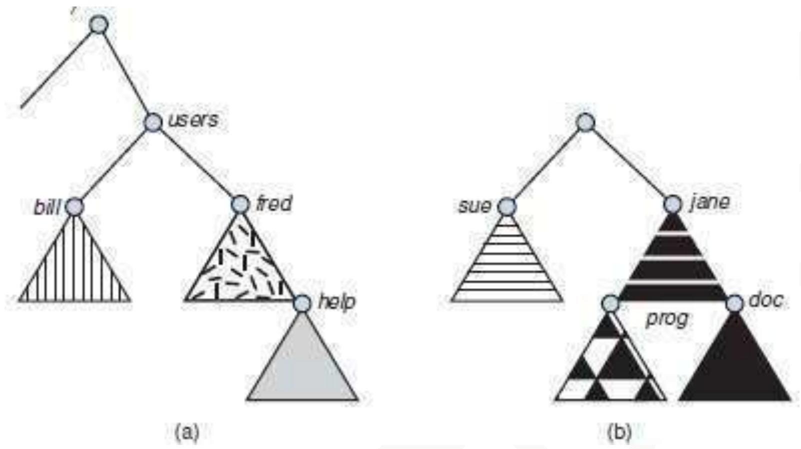


Figure 4.24 File system. (a) Existing system. (b) Unmounted volume

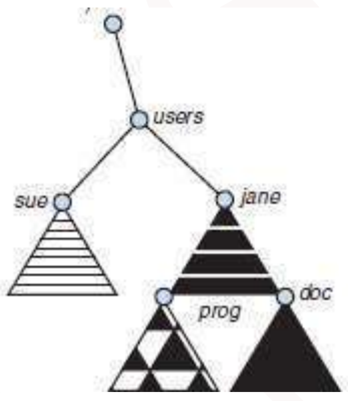


Figure 4.25 Mount point

4.11 File Sharing

• Sharing of files on multi-user systems is desirable.
• Sharing may be done through a protection scheme.
• On distributed systems, files may be shared across a network.
• Network File-system (NFS) is a common distributed file-sharing method.

4.11.1 Multiple Users

• File-sharing can be done in 2 ways:
 1) The system can allow a user to access the files of other users by default or
 2) The system may require that a user specifically grant access.
• To implement file-sharing, the system must maintain more file- & directory-attributes than on a single-user system.
• Most systems use concepts of file owner and group.
 1) Owner
 ➢ The user who
  → may change attributes & grant access and
  → has the most control over the file (or directory).
 ➢ Most systems implement owner attributes by managing a list of user-names and user IDs
 2) Group
 ➢ The group attribute defines a subset of users who can share access to the file.
 ➢ Group functionality can be implemented as a system-wide list of group-names and group IDs.
• Exactly which operations can be executed by group-members and other users is definable by the file's owner.
• The owner and group IDs of a file
 → are stored with the other file-attributes.
 → can be used to allow/deny requested operations.

4.11.2 Remote File Systems

• Allows a computer to mount 1 or more file-systems from 1 or more remote-machines.
• Three methods:
 1) **Manually via programs like FTP**.
 2) **Automatically DFS** (Distributed file-system): remote directories are visible from a local machine.
 3) **Semi-automatically via www** (World Wide Web): A browser is needed to gain access to the remote files, and separate operations (a wrapper for ftp) are used to transfer files.
• ftp is used for both anonymous and authenticated access.
• Anonymous access allows a user to transfer files without having an account on the remote system.

4.11.2.1 Client Server Model

• Allows clients to mount remote file-systems from servers.
• The machine containing the files is called the **server**.
 The machine seeking access to the files is called the **client**.
• A server can serve multiple clients, and
 A client can use multiple servers.
• The server specifies which resources (files) are available to which clients.
• A client can be specified by a network-name such as an IP address.
• Disadvantage:
 1) Client identification is more difficult.
• In UNIX and its NFS (network file-system), authentication takes place via the client networking info., by default.
• Once the remote file-system is mounted, file-operation requests are sent to the server via the DFS protocol.

4.11.2.2 Distributed Information Systems

• Provides unified access to the info. needed for remote computing.
• The DNS (domain name system) provides hostname-to-networkaddress translations.
• Other distributed info. systems provide username/password space for a distributed facility.

4.11.2.3 Failure Modes

• Local file-systems can fail for a variety of reasons such as
→ failure of disk (containing the file-system)
→ corruption of directory-structure &
→ cable failure.
• Remote file-systems have more failure modes because of the complexity of network-systems.
• The network can be interrupted between 2 hosts. Such interruptions can result from
→ hardware failure
→ poor hardware configuration or
→ networking implementation issues.
• DFS protocols allow delaying of file-system operations to remote-hosts, with the hope that the remote-host will become available again.
• To implement failure-recovery, some kind of state info. may be maintained on both the client and the server.

4.11.3 Consistency Semantics

• These represent an important criterion of evaluating file-systems that supports file-sharing.
• These specify how multiple users of a system are to access a shared-file simultaneously.
• In particular, they specify when modifications of data by one user will be observed by other users.
• These semantics are typically implemented as code with the file-system.
• These are directly related to the process-synchronization algorithms.
• A successful implementation of complex sharing semantics can be found in the Andrew file-system (AFS).

UNIX Semantics

➢ UNIX file-system (UFS) uses the following consistency semantics:
1) Writes to an open-file by a user are visible immediately to other users who have this file opened.
2) One mode of sharing allows users to share the pointer of current location into a file. Thus, the advancing of the pointer by one user affects all sharing users.

➢ A file is associated with a single physical image that is accessed as an exclusive resource.
➢ Contention for the single image causes delays in user processes.

Session Semantics

➢ The AFS uses the following consistency semantics:
1) Writes to an open file by a user are not visible immediately to other users that have the same file open.
2) Once a file is closed, the changes made to it are visible only in sessions starting later. Already open instances of the file do not reflect these changes.

➢ A file may be associated temporarily with several (possibly different) images at the same time.
➢ Consequently, multiple users are allowed to perform both read and write accesses concurrently on their images of the file, without delay.
➢ Almost no constraints are enforced on scheduling accesses.

Immutable Shared Files Semantics

➢ Once a file is declared as shared by its creator, it cannot be modified.
➢ An immutable file has 2 key properties:
1) File-name may not be reused and
2) File-contents may not be altered.

➢ Thus, the name of an immutable file signifies that the contents of the file are fixed.
➢ The implementation of these semantics in a distributed system is simple, because the sharing is disciplined

4.12 Protection

• When info. is stored 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.
• For a small single-user system, we might provide protection by physically removing the floppy disks and locking them in a desk drawer.
• File owner/creator should be able to control:
→ what can be done
→ by whom.

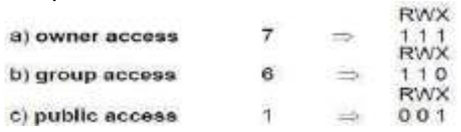
4.12.1 Types of Access

• Systems that do not permit access to the files of other users do not need protection. This is too extreme, so controlled-access is needed.
• Following operations may be controlled:

1) Read
➢ Read from the file.
2) Write
➢ Write or rewrite the file.
3) Execute
➢ Load the file into memory and execute it.
4) Append
➢ Write new info. at the end of the file.
5) Delete
➢ Delete the file and tree its space for possible reuse.
6) List
➢ List the name and attributes of the file.

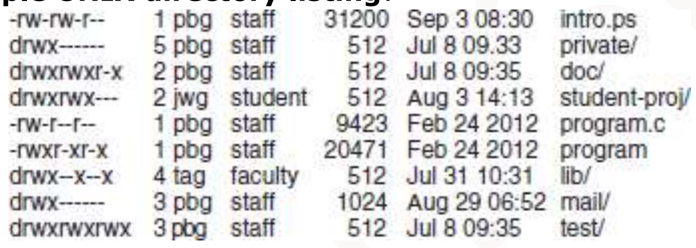
4.12.2 Access Control

• Common approach to protection problem: make access dependent on identity of user.
• Files can be associated with an ACL (access-control list) which specifies
→ username and
→ types of access for each user.
• Problems:
1) Constructing a list can be tedious.
2) Directory-entry now needs to be of variable-size, resulting in more complicated space management.
Solution: These problems can be resolved by combining ACLs with an „owner, group, universe' access-control scheme
• To reduce the length of the ACL, many systems recognize 3 classifications of users:
1) Owner
➢ The user who created the file is the owner.
2) Group
➢ A set of users who are sharing the file and need similar access is a group.
3) Universe
➢ All other users in the system constitute the universe.
• Samples:



			RWX
a) owner access	7	⇒	1 1 1
			RWX
b) group access	6	⇒	1 1 0
			RWX
c) public access	1	⇒	0 0 1

E.g. rwx bits indicate which users have permission to read/write/execute
A Sample UNIX directory listing:



```
-rw-rw-r--    1 pbg  staff     31200  Sep 3 08:30   intro.ps
drwx------    5 pbg  staff       512  Jul 8 09.33   private/
drwxrwxr-x    2 pbg  staff       512  Jul 8 09:35   doc/
drwxrwx---    2 jwg  student     512  Aug 3 14:13   student-proj/
-rw-r--r--    1 pbg  staff      9423  Feb 24 2012   program.c
-rwxr-xr-x    1 pbg  staff     20471  Feb 24 2012   program
drwx--x--x    4 tag  faculty     512  Jul 31 10:31  lib/
drwx------    3 pbg  staff      1024  Aug 29 06:52  mail/
drwxrwxrwx    3 pbg  staff       512  Jul 8 09:35   test/
```

4.12.3 Other Protection Approaches

• A password can be associated with each file.
• Disadvantages:
1) The no. of passwords you need to remember may become large.
2) If only one password is used for all the files, then all files are accessible if it is discovered.
3) Commonly, only one password is associated with all of the user's files, so protection is all-or-nothing.
• In a multilevel directory-structure, we need to provide a mechanism for directory protection.
• The directory operations that must be protected are different from the File-operations:
1) Control creation & deletion of files in a directory.
2) Control whether a user can determine the existence of a file in a directory.

MODULE 4 (CONT.): FILE-SYSTEM IMPLEMENTATION

4.13 File System Structure

• Disks provide the bulk of secondary-storage on which a file-system is maintained.

• The disk is a suitable medium for storing multiple files. This is because

1) A disk can be rewritten in place.

➢ It is possible to

→ read a block from the disk

→ modify the block and

→ write the block into the disk.

2) A disk can access directly any block of information.

➢ It is possible to access any file either sequentially or randomly.

➢ 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.

➢ Each block has one or more sectors.

➢ Depending on the disk drive, sector-size varies from 32 bytes to 4096 bytes.

➢ The usual size is 512 bytes.

• File-systems provide efficient and convenient access to the disk by allowing data to be stored, located, and retrieved easily

• Design problems of file-systems:

1) Defining how the file-system should look to the user.

2) Creating algorithms & data-structures to map the logical file-system onto the physical secondary-storage devices.

• The file-system itself is generally composed of many different levels.

Every level in design uses features of lower levels to create new features for use by higher levels.

4.13.1 Layered File System

• Levels of the file-system(Figure 4.26):

1) I/O Control (Lowest level)

➢ Consists of device-drivers & interrupt handlers to transfer info. between main-memory & disk.

➢ A device-driver can be thought of as a translator.

Its input consists of high-level commands.

Its output consists of low-level instructions.

2) Basic File-system

➢ Needed only to issue basic commands to the appropriate device-driver to read & write blocks on the disk.

3) File-organization Module

➢ Knows about files and their logical & physical blocks.

➢ Translates logical-block address to physical-block address.

4) Logical File-system

➢ Manages metadata information. i.e. **Metadata** includes all of the file-system structure except the actual data.

➢ Manages the directory-structure.

➢ Maintains file-structure via FCB (File Control Blocks). i.e. **FCB** contains info. about the file, including

→ ownership

→ permissions and

→ location of the file.

• Advantages of layered structure:

1) Duplication of code is minimized.

2) I/O control can be used by multiple file-systems.

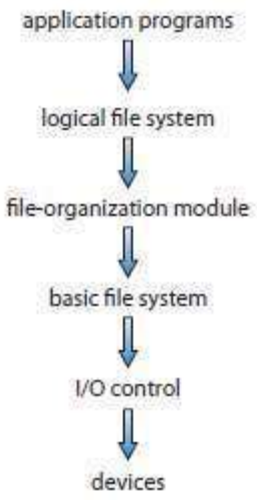


Figure 4.26 Layered file system

4.14 File System Implementation

4.14.1 Overview

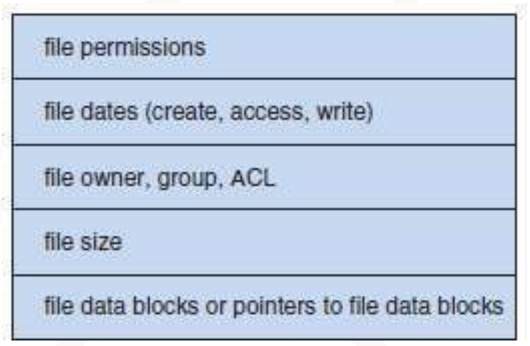
• On-disk & in-memory structures are used to implement a file-system.
• On-disk structures include (Figure 4.27):

1) Boot Control Block
➢ Contains info. needed to boot an OS from the partition.
➢ 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.

2) Partition Control Block
➢ Contains partition-details like
→ no. of blocks
→ size of blocks and
→ free-block count.
➢ In UFS, this is called a superblock.
In NTFS, it is stored in the master file table.

3) Directory-structure
➢ Used to organize the files.
➢ In UFS, this includes file-names and associated inode-numbers.
In NTFS, it is stored in the master file table.

4) FCB (file control block)
➢ Contains file-details including
→ file-permissions
→ ownership
→ file-size and
→ location of data-blocks.



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

Figure 4.27 A typical file-control block

• In-memory structures are used for both file-system management and performance improvement via caching: The structures may include:

1) In-memory Mount Table
➢ Contains info. about each mounted partition.

2) An in-memory Directory-structure
➢ Holds directory info. of recently accessed directories.

3) System-wide Open-file Table
➢ Contains a copy of the FCB of each open file

4) Per-process Open-file Table
➢ Contains a pointer to the appropriate entry in the system-wide open-file table.

• **Buffers** hold file-system blocks when they are being read from disk or written to disk.
• To create a new file, a program calls the LFS (logical file-system).
The „LFS' knows the format of the directory-structures.
• To create a new file, the LFS
1) Allocates a new FCB.
2) Reads the appropriate directory into memory.
3) Updates LFS with the new file-name and FCB.
4) Writes LFS back to the disk (Figure 4.28).

• After a file has been created, it can be used for I/O.

1) First the file must be opened.
2) FCB is copied to a system-wide open-file table in memory.
3) An entry is made in the **per-process** open-file table, with a pointer to the entry in the **system-wide** open-file table.
4) The open call returns a pointer to the appropriate entry in the per-process file-system table.
5) All file operations are then performed via this pointer.
6) When a process closes the file
    i) The per-process table entry is removed.
    ii) The system-wide entry"s open count is decremented.

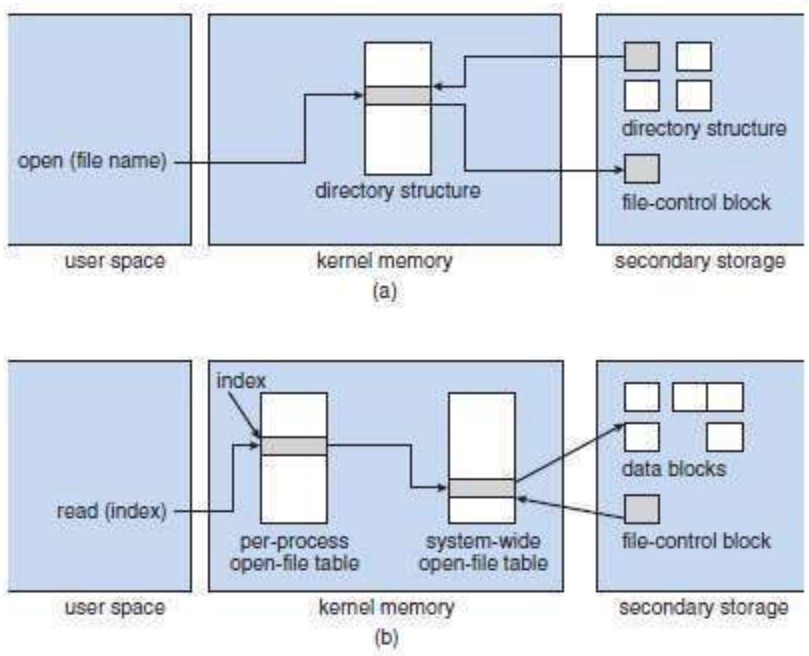


Figure 4.28 In-memory file-system structures. (a) File open. (b) File read

4.14.2 Partitions & Mounting

• Disk layouts can be:

1) A disk can be divided into multiple partitions or
2) A partition can span multiple disks (RAID).

• Each partition can either be:

1) **Raw** i.e. containing no file-system or
2) **Cooked** i.e. containing a file-system.

• **Boot info.** is a sequential series of blocks, loaded as an image into memory.

➢ Execution of the image starts at a predefined location, such as the first byte.

• The boot info. has its own format, because

→ at boot time the system does not have device-drivers loaded and
→ . „. the system cannot interpret the file-system format.

• Steps for mounting:

1) The root partition containing the kernel is mounted at boot time.
2) Then, the OS verifies that the device contains a valid file-system.
3) Finally, the OS notes in its in-memory mount table structure that
    i) A file-system is mounted and
    ii) Type of the file-system.

4.14.3 Virtual File Systems

• The OS allows multiple types of file-systems to be integrated into a directory-structure.
• Three layers (Figure 4.29):

1) File-system Interface

➢ This is based on the open(), read(), writeO and closeO calls on file descriptors.

2) File-system (VFS) Interface

➢ This serves 2 functions:

i) Separates file-system basic operations from their implementation by defining a clean VFS interface.

ii) The VFS is based on a file-representation structure called a **vnode**.

vnode contains a numerical designator for a network-wide unique file.

3) Local File-system

➢ Local files are distinguished according to their file-system types.

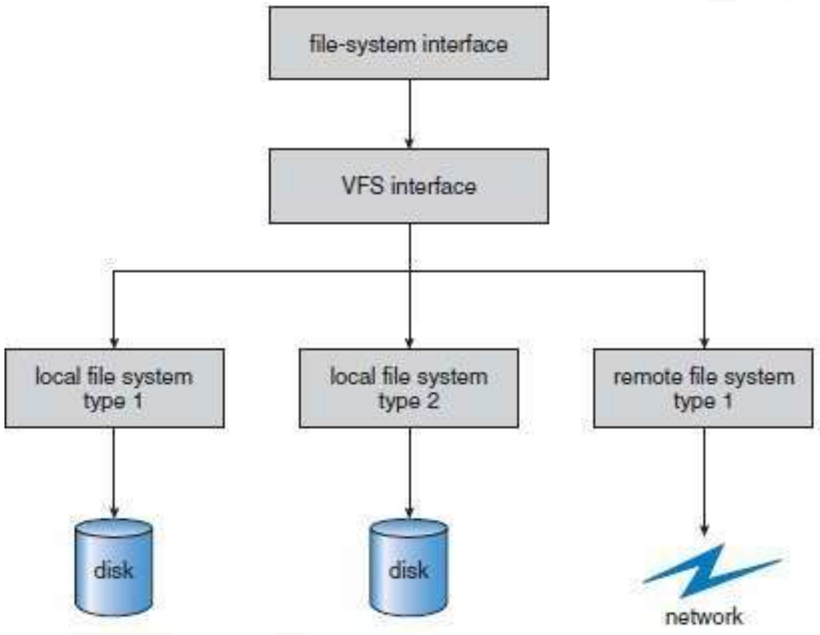


Figure 4.29 Schematic view of a virtual file system

4.15 Directory Implementation

1) Linear-list
2) Hash-table

4.15.1 Linear List

• A linear-list of file-names has pointers to the data-blocks.
• To create a new file:
  1) First search the directory to be sure that no existing file has the same name.
  2) Then, add a new entry at the end of the directory.
• To delete a file:
  1) Search the directory for the named-file and
  2) Then release the space allocated to the file.
• To reuse the directory-entry, there are 3 solutions:
  1) Mark the entry as unused (by assigning it a special name).
  2) Attach the entry to a list of free directory entries.
  3) Copy the last entry in the directory into the freed location & to decrease length of directory.
• Problem: Finding a file requires a linear-search which is slow to execute.
  Solutions:
  1) A cache can be used to store the most recently used directory information.
  2) A sorted list allows a binary search and decreases search time.
• Advantage:
  1) Simple to program.
• Disadvantage:
  1) Time-consuming to execute.

4.15.2 Hash Table

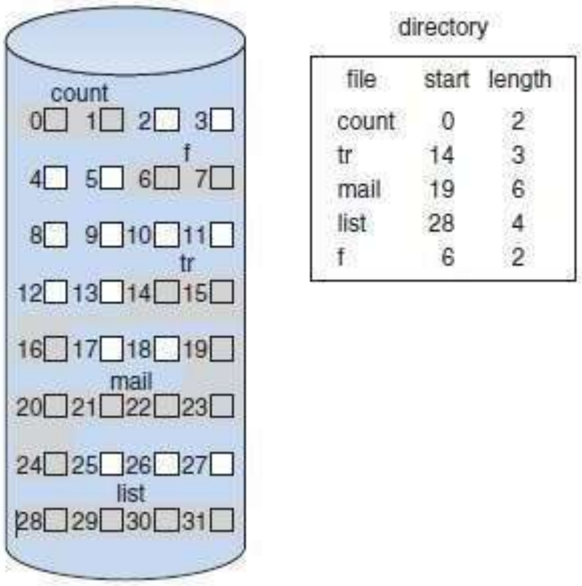
• A linear-list stores the directory-entries. In addition, a hash data-structure is also used.
• The hash-table
  → takes a value computed from the file name and
  → returns a pointer to the file name in the linear-list.
• Advantages:
  1) Decrease the directory search-time.
  2) Insertion & deletion are easy.
• Disadvantages:
  1) Some provision must be made for collisions i.e. a situation in which 2 file-names hash to the same location.
  2) Fixed size of hash-table and the dependence of the hash function on that size.

4.16 Allocation Methods

• The direct-access nature of disks allows us flexibility in the implementation of files.
• In almost every case, many files are stored on the same disk.
• Main problem:
    How to allocate space to the files so that
        → disk-space is utilized effectively and
        → files can be accessed quickly.
• Three methods of allocating disk-space:
    1) Contiguous
    2) Linked and
    3) Indexed
• Each method has advantages and disadvantages.
• Some systems support all three (Data General's RDOS for its Nova line of computers).

4.16.1 Contiguous Allocation

• Each file occupies a set of contiguous-blocks on the disk (Figure 4.30).
• Disk addresses define a linear ordering on the disk.
• The number of disk seeks required for accessing contiguously allocated files is minimal.
• Both sequential and direct access can be supported.
• Problems:
    1) Finding space for a new file
    ➢ External fragmentation can occur.
    2) Determining how much space is needed for a file.
    ➢ If you allocate too little space, it can't be extended.
        Two solutions:
        i) The user-program can be terminated with an appropriate error-message. The user must then allocate more space and run the program again.
        ii) Find a larger hole,
            copy the contents of the file to the new space and
                release the previous space.
• To minimize these drawbacks:
    1) A contiguous chunk of space can be allocated initially and
    2) Then when that amount is not large enough, another chunk of contiguous space (known as an „extent") is added.



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

Figure 4.30 Contiguous allocation of disk-space

4.16.2 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 (Figure 4.31).
• To create a new file, just create a new entry in the directory (each directory-entry has a pointer to the disk-block of the file).

1) A **write** to the file causes a free block to be found. This new block is then written to and linked to the eof (end of file).
2) A **read** to the file causes moving the pointers from block to block.

• Advantages:

1) No external fragmentation, and any free block on the free-space list can be used to satisfy a request.
2) The size of the file doesn't need to be declared on creation.
3) Not necessary to compact disk-space.

• Disadvantages:

1) Can be used effectively only for sequential-access files.
2) Space required for the pointers.
Solution: Collect blocks into multiples (called „clusters") & allocate clusters rather than blocks.
3) Reliability: Problem occurs if a pointer is lost(or damaged).
Partial solutions: i) Use doubly linked-lists.
ii) Store file name and relative block-number in each block.

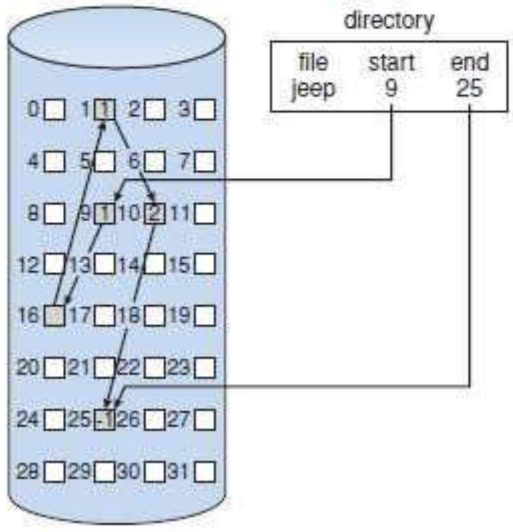


Figure 4.31 Linked allocation of disk-space

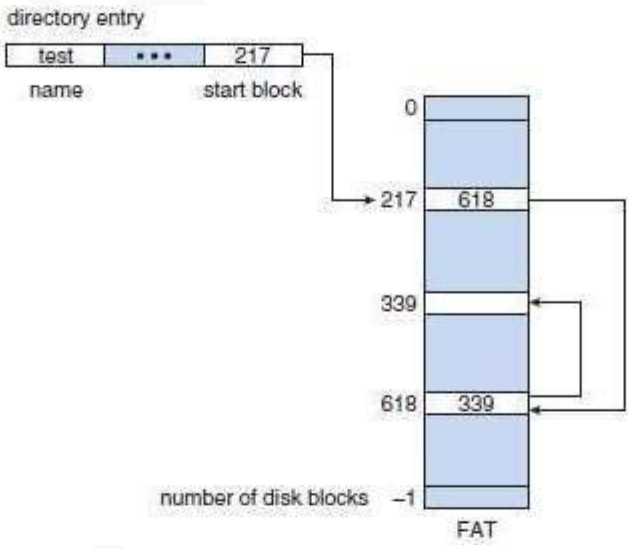


Figure 4.32 File-allocation table

• FAT is a variation on linked allocation (FAT=File Allocation Table).
• A section of disk at the beginning of each partition is set aside to contain the table (Figure 4.32).
• The table

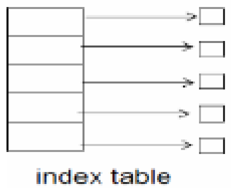
→ has one entry for each disk-block and
→ is indexed by block-number.

• The directory-entry contains the block-number of the first block in the file.
• The table entry indexed by that block-number then contains the block-number of the next block in the file.
• This chain continues until the last block, which has a special end-of-file value as the table entry.
• Advantages:

1) Cache can be used to reduce the no. of disk head seeks.
2) Improved access time, since the disk head can find the location of any block by reading the info in the FAT.

4.16.3 Indexed Allocation

• Solves the problems of linked allocation (without a FAT) by bringing all the pointers together into an index block.
• Each file has its own index block, which is an array of disk-block addresses.



Logical view of the Index Table

• The ith entry in the index block points to the ith file block (Figure 4.33).
• The directory contains the address of the index block.

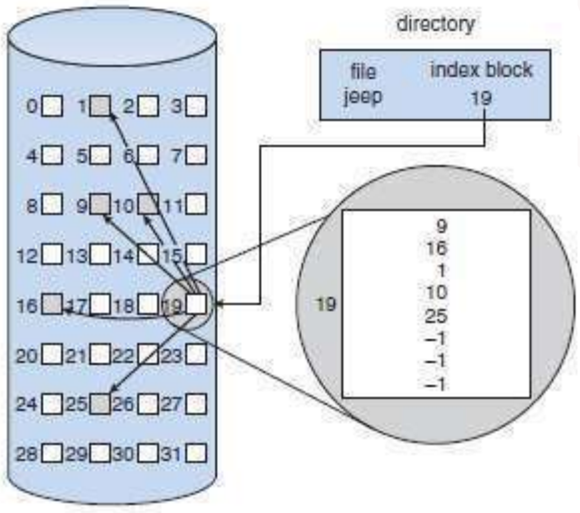


Figure 4.33 Indexed allocation of disk space

• When the file is created, all pointers in the index-block are set to nil.
• When writing the ith block, a block is obtained from the free-space manager, and its address put in the ith index-block entry,
• Problem: If the index block is too small, it will not be able to hold enough pointers for a large file,
Solution: Three mechanisms to deal with this problem:

1) Linked Scheme

➢ To allow for large files, link several index blocks,

2) Multilevel Index

➢ A first-level index block points to second-level ones, which in turn point to the file blocks,

3) Combined Scheme

➢The first few pointers point to direct blocks (i.e. they contain addresses of blocks that contain data of the file).

➢ The next few pointers point to indirect blocks.

• Advantage:
1) Supports direct access, without external fragmentation,
• Disadvantages:
1) Suffer from wasted space,
2) The pointer overhead of the index block is generally greater than the pointer overhead of linked allocation,
3) Suffer from performance problems,

4.16.4 Performance

Contiguous Allocation

1Adv)Requires only one access to get a disk-block
2Adv) We can calculate immediately the disk address of the next block and read it directly
3Adv) Good for direct access

Linked Allocation

1Adv) Good for sequential access
1Dis) Not be used for an application requiring direct access

Indexed Allocation

1Adv) If the index block is already in memory, then the access can be made directly
1Dis) keeping the index block in memory requires considerable space

(Adv → Advantage Dis → Disadvantage)

4.17 Free Space Management

• A **free-space list** keeps track of free disk-space (i.e. those not allocated to some file or directory).
• To create a file,
  1) We search the free-space list for the required amount of space.
  2) Allocate that space to the new file.
  3) This space is then removed from the free-space list.
• To delete a file, its disk-space is added to the free-space list.

1) Bit Vector

• The free-space list is implemented as a bit map/bit vector.
• Each block is represented by a bit.
  1) If the block is free, the bit is 1.
  2) If the block is allocated, the bit is 0.
• For example, consider a disk where blocks 2, 3, 4, 5 and 7 are free and the rest of the blocks are allocated. The free-space bit map will be
  00111101
• Advantage:
  1) Relative simplicity & efficiency in finding the first free block, or „n' consecutive free blocks.
• Disadvantages:
  1) Inefficient unless the entire vector is kept in main memory.
  2) The entire vector is written to disc occasionally for recovery.

2) Linked List

• The basic idea:
  1) Link together all the free disk-blocks (Figure 4.34).
  2) Keep a pointer to the first free block in a special location on the disk.
  3) Cache the block in memory.
• The first block contains a pointer to the next free one, etc.
• Disadvantage:
  1) Not efficient, because to traverse the list, each block is read.
• Usually the OS simply needs a free block, and uses the first one.

3) Grouping

• The addresses of n free blocks are stored in the 1st free block.
• The first n-1 of these blocks are actually free.
• The last block contains addresses of another n free blocks, etc.
• Advantage:
  1) Addresses of a large no of free blocks can be found quickly.

4) Counting

• Takes advantage of the fact that, generally, several contiguous blocks may be allocated/freed simultaneously.
• Keep the address of the first free block and the number „n' of free contiguous blocks that follow the first block.
• Each entry in the free-space list then consists of a disk address and a count.

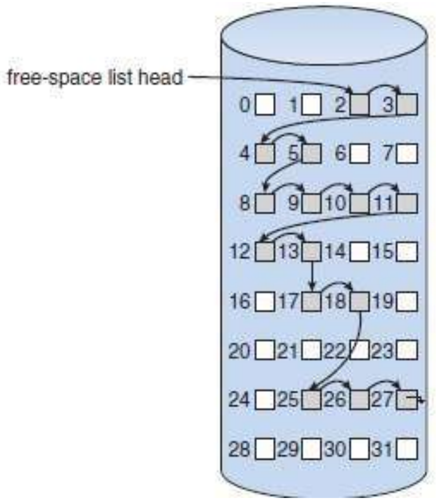


Figure 4.34 Linked free-space list on disk