

Deadlocks

7.1 System Model

- For the purposes of deadlock discussion, a system can be modeled as a collection of limited resources, which can be partitioned into different categories, to be allocated to a number of processes, each having different needs.
- Resource categories may include memory, printers, CPUs, open files, tape drives, CD-ROMS, etc.
- By definition, all the resources within a category are equivalent, and a request of this category can be equally satisfied by any one of the resources in that category. If this is not the case (i.e. if there is some difference between the resources within a category), then that category needs to be further divided into separate categories. For example, "printers" may need to be separated into "laser printers" and "color inkjet printers".
- Some categories may have a single resource.
- In normal operation a process must request a resource before using it, and release it when it is done, in the following sequence:
 1. Request - If the request cannot be immediately granted, then the process must wait until the resource(s) it needs become available. For example the system calls `open()`, `malloc()`, `new()`, and `request()`.
 2. Use - The process uses the resource, e.g. prints to the printer or reads from the file.
 3. Release - The process relinquishes the resource. so that it becomes available for other processes. For example, `close()`, `free()`, `delete()`, and `release()`.
- For all kernel-managed resources, the kernel keeps track of what resources are free and which are allocated, to which process they are allocated, and a queue of processes waiting for this resource to become available. Application-managed resources can be controlled using mutexes or `wait()` and `signal()` calls, (i.e. binary or counting semaphores.)
- A set of processes is deadlocked when every process in the set is waiting for a resource that is currently allocated to another process in the set (and which can only be released when that other waiting process makes progress.)

DEADLOCK WITH MUTEX LOCKS

Let's see how deadlock can occur in a multithreaded Pthread program using mutex locks. The `pthread_mutex_init()` function initializes an unlocked mutex. Mutex locks are acquired and released using `pthread_mutex_lock()` and `pthread_mutex_unlock()`, respectively. If a thread attempts to acquire a locked mutex, the call to `pthread_mutex_lock()` blocks the thread until the owner of the mutex lock invokes `pthread_mutex_unlock()`.

Two mutex locks are created in the following code example:

```
/* Create and initialize the mutex locks */
pthread_mutex_t first_mutex;
pthread_mutex_t second_mutex;

pthread_mutex_init(&first_mutex, NULL);
pthread_mutex_init(&second_mutex, NULL);
```

Next, two threads—`thread_one` and `thread_two`—are created, and both these threads have access to both mutex locks. `thread_one` and `thread_two` run in the functions `do_work_one()` and `do_work_two()`, respectively, as shown below:

```
/* thread_one runs in this function */
void *do_work_one(void *param)
{
    pthread_mutex_lock(&first_mutex);
    pthread_mutex_lock(&second_mutex);
    /**
     * Do some work
     */
    pthread_mutex_unlock(&second_mutex);
    pthread_mutex_unlock(&first_mutex);

    pthread_exit(0);
}

/* thread_two runs in this function */
void *do_work_two(void *param)
{
    pthread_mutex_lock(&second_mutex);
    pthread_mutex_lock(&first_mutex);
    /**
     * Do some work
     */
    pthread_mutex_unlock(&first_mutex);
    pthread_mutex_unlock(&second_mutex);

    pthread_exit(0);
}
```

In this example, `thread_one` attempts to acquire the mutex locks in the order (1) `first_mutex`, (2) `second_mutex`, while `thread_two` attempts to acquire the mutex locks in the order (1) `second_mutex`, (2) `first_mutex`. Deadlock is possible if `thread_one` acquires `first_mutex` while `thread_two` acquires `second_mutex`.

Note that, even though deadlock is possible, it will not occur if `thread_one` can acquire and release the mutex locks for `first_mutex` and `second_mutex` before `thread_two` attempts to acquire the locks. And, of course, the order in which the threads run depends on how they are scheduled by the CPU scheduler. This example illustrates a problem with handling deadlocks: it is difficult to identify and test for deadlocks that may occur only under certain scheduling circumstances.

New Sidebar in Ninth Edition

7.2.1 Necessary Conditions

- There are four conditions that are necessary to achieve deadlock:
 1. **Mutual Exclusion** - At least one resource must be held in a non-sharable mode; If any other process requests this resource, then that process must wait for the resource to be released.
 2. **Hold and Wait** - A process must be simultaneously holding at least one resource and waiting for at least one resource that is currently being held by some other process.
 3. **No preemption** - Once a process is holding a resource (i.e. once its request has been granted), then that resource cannot be taken away from that process until the process voluntarily releases it.
 4. **Circular Wait** - A set of processes { $P_0, P_1, P_2, \dots, P_N$ } must exist such that every $P[i]$ is waiting for $P[(i + 1) \% (N + 1)]$. (Note that this condition implies the hold-and-wait condition, but it is easier to deal with the conditions if the four are considered separately.)

7.2.2 Resource-Allocation Graph

- In some cases deadlocks can be understood more clearly through the use of **Resource-Allocation Graphs**, having the following properties:
 - A set of resource categories, { $R_1, R_2, R_3, \dots, R_N$ }, which appear as square nodes on the graph. Dots inside the resource nodes indicate

specific instances of the resource. (E.g. two dots might represent two laser printers.)

- A set of processes, { P1, P2, P3, . . . , PN }
- **Request Edges** - A set of directed arcs from Pi to Rj, indicating that process Pi has requested Rj, and is currently waiting for that resource to become available.
- **Assignment Edges** - A set of directed arcs from Rj to Pi indicating that resource Rj has been allocated to process Pi, and that Pi is currently holding resource Rj.
- Note that a **request edge** can be converted into an **assignment edge** by reversing the direction of the arc when the request is granted. (However note also that request edges point to the category box, whereas assignment edges emanate from a particular instance dot within the box.)
- For example:

Figure 7.1 - Resource allocation graph

- If a resource-allocation graph contains no cycles, then the system is not deadlocked. (When looking for cycles, remember that these are **directed** graphs.) See the example in Figure 7.2 above.
- If a resource-allocation graph does contain cycles **AND** each resource category contains only a single instance, then a deadlock exists.
- If a resource category contains more than one instance, then the presence of a cycle in the resource-allocation graph indicates the *possibility* of a deadlock, but does not guarantee one. Consider, for example, Figures 7.3 and 7.4 below:

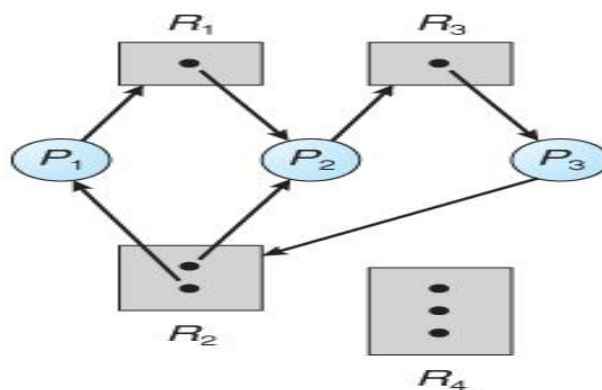


Figure 7.2 - Resource allocation graph with a deadlock

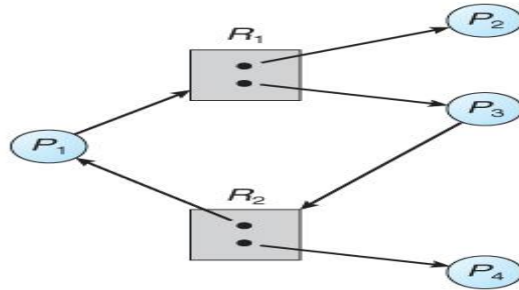


Figure 7.3 - Resource allocation graph with a cycle but no deadlock

7.3 Methods for Handling Deadlocks

- Generally speaking there are three ways of handling deadlocks:
 1. Deadlock prevention or avoidance - Do not allow the system to get into a deadlocked state.
 2. Deadlock detection and recovery - Abort a process or preempt some resources when deadlocks are detected.
 3. Ignore the problem all together - If deadlocks only occur once a year or so, it may be better to simply let them happen and reboot as necessary than to incur the constant overhead and system performance penalties associated with deadlock prevention or detection. This is the approach that both Windows and UNIX take.
- In order to avoid deadlocks, the system must have additional information about all processes. In particular, the system must know what resources a process will or may request in the future. (Ranging from a simple worst-case maximum to a complete resource request and release plan for each process, depending on the particular algorithm.)
- Deadlock detection is fairly straightforward, but deadlock recovery requires either aborting processes or preempting resources, neither of which is an attractive alternative.
- If deadlocks are neither prevented nor detected, then when a deadlock occurs the system will gradually slow down, as more and more processes become stuck waiting for resources currently held by the deadlock and by other waiting processes. Unfortunately this slowdown can be indistinguishable from a general system slowdown when a real-time process has heavy computing needs.

7.4 Deadlock Prevention

- Deadlocks can be prevented by preventing at least one of the four required conditions:

7.4.1 Mutual Exclusion

- Shared resources such as read-only files do not lead to deadlocks.
- Unfortunately some resources, such as printers and tape drives, require exclusive access by a single process.

7.4.2 Hold and Wait

- To prevent this condition processes must be prevented from holding one or more resources while simultaneously waiting for one or more others. There are several possibilities for this:
 - Require that all processes request all resources at one time. This can be wasteful of system resources if a process needs one resource early in its execution and doesn't need some other resource until much later.
 - Require that processes holding resources must release them before requesting new resources, and then re-acquire the released resources along with the new ones in a single new request. This can be a problem if a process has partially completed an operation using a resource and then fails to get it re-allocated after releasing it.
 - Either of the methods described above can lead to starvation if a process requires one or more popular resources.

7.4.3 No Preemption

- Preemption of process resource allocations can prevent this condition of deadlocks, when it is possible.
 - One approach is that if a process is forced to wait when requesting a new resource, then all other resources previously held by this process are implicitly released, (preempted), forcing this process to re-acquire the old resources along with the new resources in a single request, similar to the previous discussion.
 - Another approach is that when a resource is requested and not available, then the system looks to see what other processes currently have those resources *and* are themselves blocked waiting for some other resource. If such a process is found, then some of their resources may get preempted and added to the list of resources for which the process is waiting.
 - Either of these approaches may be applicable for resources whose states are easily saved and restored, such as registers and memory, but are generally not applicable to other devices such as printers and tape drives.

7.4.4 Circular Wait

- One way to avoid circular wait is to number all resources, and to require that processes request resources only in strictly increasing (or decreasing) order.
- In other words, in order to request resource R_j , a process must first release all R_i such that $i \geq j$.
- One big challenge in this scheme is determining the relative ordering of the different resources

7.5 Deadlock Avoidance

- The general idea behind deadlock avoidance is to prevent deadlocks from ever happening, by preventing at least one of the aforementioned conditions.
- This requires more information about each process, AND tends to lead to low device utilization. (I.e. it is a conservative approach.)
- In some algorithms the scheduler only needs to know the *maximum* number of each resource that a process might potentially use. In more complex algorithms the scheduler can also take advantage of the *schedule* of exactly what resources may be needed in what order.
- When a scheduler sees that starting a process or granting resource requests may lead to future deadlocks, then that process is just not started or the request is not granted.
- A resource allocation **state** is defined by the number of available and allocated resources, and the maximum requirements of all processes in the system.

7.5.1 Safe State

- A state is **safe** if the system can allocate all resources requested by all processes (up to their stated maximums) without entering a deadlock state.
- More formally, a state is safe if there exists a **safe sequence** of processes $\{ P_0, P_1, P_2, \dots, P_N \}$ such that all of the resource requests for P_i can be granted using the resources currently allocated to P_i and all processes P_j where $j < i$. (I.e. if all the processes prior to P_i finish and free up their resources, then P_i will be able to finish also, using the resources that they have freed up.)
- If a safe sequence does not exist, then the system is in an unsafe state, which **MAY** lead to deadlock. (All safe states are deadlock free, but not all unsafe states lead to deadlocks.)

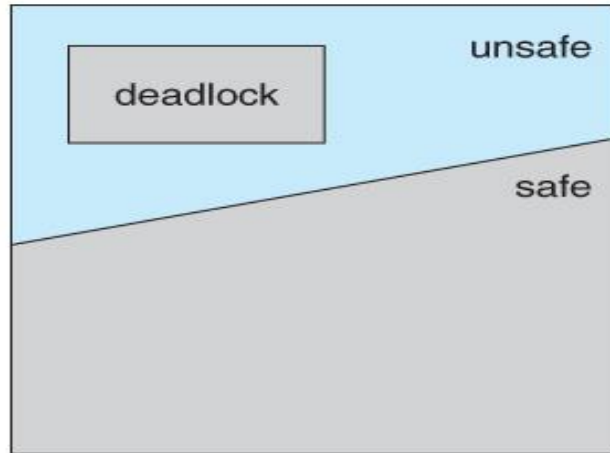


Figure 7.6 - Safe, unsafe, and deadlocked state spaces.

- For example, consider a system with 12 tape drives, allocated as follows. Is this a safe state? What is the safe sequence?

	Maximum Needs	Current Allocation
P0	10	5
P1	4	2
P2	9	2

What happens to the above table if process P2 requests and is granted one more tape drive?

- Key to the safe state approach is that when a request is made for resources, the request is granted only if the resulting allocation state is a safe one.

7.5.2 Resource-Allocation Graph Algorithm

- If resource categories have only single instances of their resources, then deadlock states can be detected by cycles in the resource-allocation graphs.
- In this case, unsafe states can be recognized and avoided by augmenting the resource-allocation graph with **claim edges**, noted by dashed lines, which point from a process to a resource that it may request in the future.
- In order for this technique to work, all claim edges must be added to the graph for any particular process before that process is allowed to request any resources. (Alternatively, processes may only make requests for resources for

which they have already established claim edges, and claim edges cannot be added to any process that is currently holding resources.)

- When a process makes a request, the claim edge $P_i \rightarrow R_j$ is converted to a request edge. Similarly when a resource is released, the assignment reverts back to a claim edge.
- This approach works by denying requests that would produce cycles in the resource-allocation graph, taking claim edges into effect.
- Consider for example what happens when process P_2 requests resource R_2 :

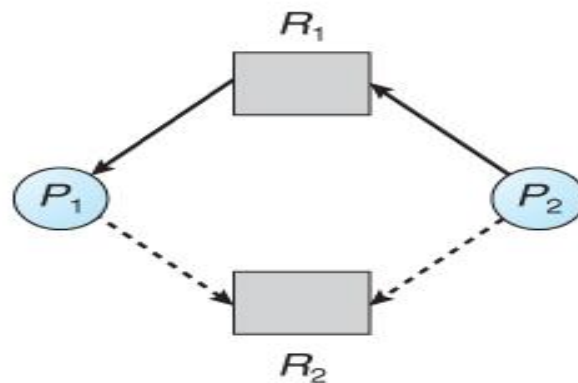


Figure 7.7 - Resource allocation graph for deadlock avoidance

- The resulting resource-allocation graph would have a cycle in it, and so the request cannot be granted.

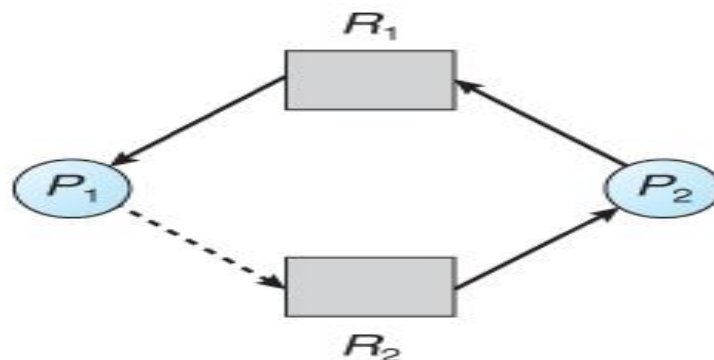


Figure 7.8 - An unsafe state in a resource allocation graph

7.5.3 Banker's Algorithm

- For resource categories that contain more than one instance the resource-allocation graph method does not work, and more complex (and less efficient) methods must be chosen.

- The Banker's Algorithm gets its name because it is a method that bankers could use to assure that when they lend out resources they will still be able to satisfy all their clients. (A banker won't loan out a little money to start building a house unless they are assured that they will later be able to loan out the rest of the money to finish the house.)
- When a process starts up, it must state in advance the maximum allocation of resources it may request, up to the amount available on the system.
- When a request is made, the scheduler determines whether granting the request would leave the system in a safe state. If not, then the process must wait until the request can be granted safely.
- The banker's algorithm relies on several key data structures: (where n is the number of processes and m is the number of resource categories.)
 - Available[m] indicates how many resources are currently available of each type.
 - Max[n][m] indicates the maximum demand of each process of each resource.
 - Allocation[n][m] indicates the number of each resource category allocated to each process.
 - Need[n][m] indicates the remaining resources needed of each type for each process. (Note that $\text{Need}[i][j] = \text{Max}[i][j] - \text{Allocation}[i][j]$ for all i, j .)
- For simplification of discussions, we make the following notations / observations:
 - One row of the Need vector, Need[i], can be treated as a vector corresponding to the needs of process i , and similarly for Allocation and Max.
 - A vector X is considered to be \leq a vector Y if $X[i] \leq Y[i]$ for all i .

7.5.3.1 Safety Algorithm

- In order to apply the Banker's algorithm, we first need an algorithm for determining whether or not a particular state is safe.
- This algorithm determines if the current state of a system is safe, according to the following steps:
 1. Let Work and Finish be vectors of length m and n respectively.
 - Work is a working copy of the available resources, which will be modified during the analysis.
 - Finish is a vector of booleans indicating whether a particular process can finish. (or has finished so far in the analysis.)

- Initialize Work to Available, and Finish to false for all elements.
 - 2. Find an i such that both (A) $\text{Finish}[i] == \text{false}$, and (B) $\text{Need}[i] < \text{Work}$. This process has not finished, but could with the given available working set. If no such i exists, go to step 4.
 - 3. Set $\text{Work} = \text{Work} + \text{Allocation}[i]$, and set $\text{Finish}[i]$ to true. This corresponds to process i finishing up and releasing its resources back into the work pool. Then loop back to step 2.
 - 4. If $\text{finish}[i] == \text{true}$ for all i , then the state is a safe state, because a safe sequence has been found.
- (JTB's Modification:
 1. In step 1. instead of making Finish an array of booleans initialized to false, make it an array of ints initialized to 0. Also initialize an int $s = 0$ as a step counter.
 2. In step 2, look for $\text{Finish}[i] == 0$.
 3. In step 3, set $\text{Finish}[i]$ to $++s$. s is counting the number of finished processes.
 4. For step 4, the test can be either $\text{Finish}[i] > 0$ for all i , or $s \geq n$. The benefit of this method is that if a safe state exists, then $\text{Finish}[]$ indicates one safe sequence (of possibly many.))

7.5.3.2 Resource-Request Algorithm (The Bankers Algorithm)

- Now that we have a tool for determining if a particular state is safe or not, we are now ready to look at the Banker's algorithm itself.
- This algorithm determines if a new request is safe, and grants it only if it is safe to do so.
- When a request is made (that does not exceed currently available resources), pretend it has been granted, and then see if the resulting state is a safe one. If so, grant the request, and if not, deny the request, as follows:
 1. Let $\text{Request}[n][m]$ indicate the number of resources of each type currently requested by processes. If $\text{Request}[i] > \text{Need}[i]$ for any process i , raise an error condition.
 2. If $\text{Request}[i] > \text{Available}$ for any process i , then that process must wait for resources to become available. Otherwise the process can continue to step 3.
 3. Check to see if the request can be granted safely, by pretending it has been granted and then seeing if the resulting state is safe. If so, grant the request, and if not, then the process must wait until its request can be

granted safely. The procedure for granting a request (or pretending to for testing purposes) is:

- Available = Available - Request
- Allocation = Allocation + Request
- Need = Need - Request

7.5.3.3 An Illustrative Example

- Consider the following situation:

	<u>Allocation</u>	<u>Max</u>	<u>Available</u>	<u>Need</u>
	A B C	A B C	A B C	A B C
P_0	0 1 0	7 5 3	3 3 2	7 4 3
P_1	2 0 0	3 2 2		1 2 2
P_2	3 0 2	9 0 2		6 0 0
P_3	2 1 1	2 2 2		0 1 1
P_4	0 0 2	4 3 3		4 3 1

- And now consider what happens if process P_1 requests 1 instance of A and 2 instances of C. ($\text{Request}[1] = (1, 0, 2)$)

	<u>Allocation</u>	<u>Need</u>	<u>Available</u>
	A B C	A B C	A B C
P_0	0 1 0	7 4 3	2 3 0
P_1	3 0 2	0 2 0	
P_2	3 0 2	6 0 0	
P_3	2 1 1	0 1 1	
P_4	0 0 2	4 3 1	

- What about requests of (3, 3, 0) by P_4 ? or (0, 2, 0) by P_0 ? Can these be safely granted? Why or why not?

7.6 Deadlock Detection

- If deadlocks are not avoided, then another approach is to detect when they have occurred and recover somehow.
- In addition to the performance hit of constantly checking for deadlocks, a policy / algorithm must be in place for recovering from deadlocks, and there is potential for lost work when processes must be aborted or have their resources preempted.

7.6.1 Single Instance of Each Resource Type

- If each resource category has a single instance, then we can use a variation of the resource-allocation graph known as a **wait-for graph**.
- A wait-for graph can be constructed from a resource-allocation graph by eliminating the resources and collapsing the associated edges, as shown in the figure below.
- An arc from P_i to P_j in a wait-for graph indicates that process P_i is waiting for a resource that process P_j is currently holding.

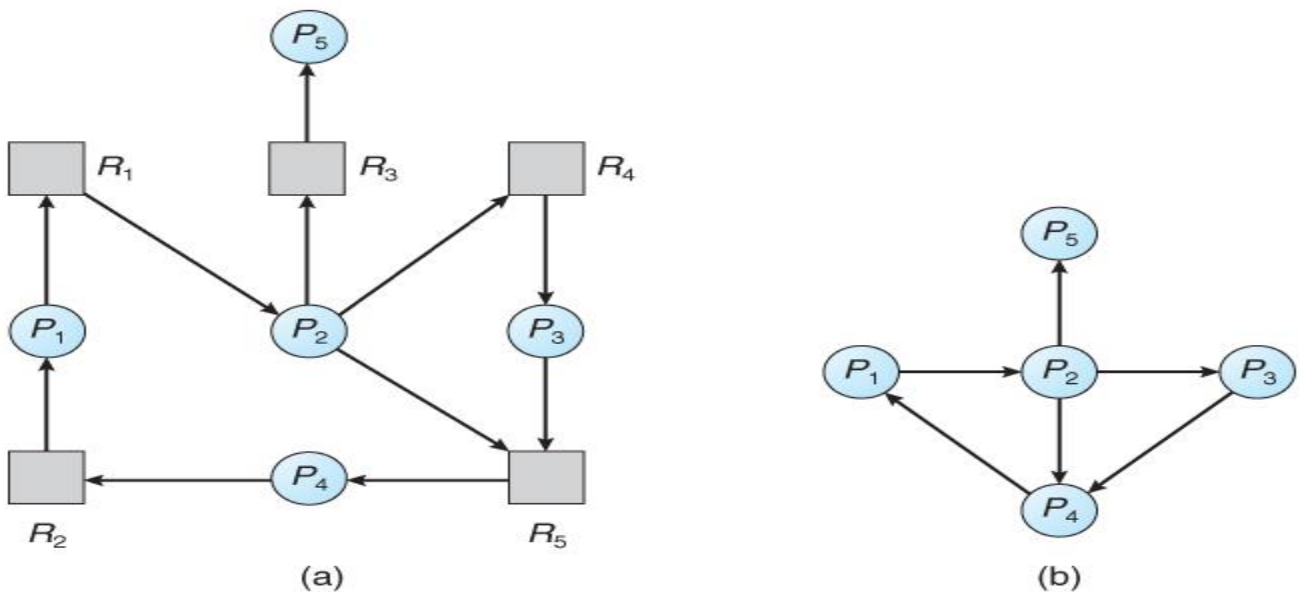


Figure 7.9 - (a) Resource allocation graph. (b) Corresponding wait-for graph

- As before, cycles in the wait-for graph indicate deadlocks.
- This algorithm must maintain the wait-for graph, and periodically search it for cycles.

7.6.2 Several Instances of a Resource Type

- The detection algorithm outlined here is essentially the same as the Banker's algorithm, with two subtle differences:
 - In step 1, the Banker's Algorithm sets $Finish[i]$ to false for all i . The algorithm presented here sets $Finish[i]$ to false only if $Allocation[i]$ is not zero. If the currently allocated resources for this process are zero, the algorithm sets $Finish[i]$ to true. This is essentially assuming that IF all of the other processes can finish, then this process can finish also. Furthermore, this algorithm is specifically looking for which processes are involved in a deadlock situation, and a process that does not have any resources allocated cannot be involved in a deadlock, and so can be removed from any further consideration.
 - Steps 2 and 3 are unchanged
 - In step 4, the basic Banker's Algorithm says that if $Finish[i] == true$ for all i , that there is no deadlock. This algorithm is more specific, by stating that if $Finish[i] == false$ for any process P_i , then that process is specifically involved in the deadlock which has been detected.
- (Note: An alternative method was presented above, in which $Finish$ held integers instead of booleans. This vector would be initialized to all zeros, and then filled with increasing integers as processes are detected which can finish. If any processes are left at zero when the algorithm completes, then there is a deadlock, and if not, then the integers in $finish$ describe a safe sequence. To modify this algorithm to match this section of the text, processes with allocation = zero could be filled in with $N, N - 1, N - 2$, etc. in step 1, and any processes left with $Finish = 0$ in step 4 are the deadlocked processes.)
- Consider, for example, the following state, and determine if it is currently deadlocked:

	<u>Allocation</u>	<u>Request</u>	<u>Available</u>
	A B C	A B C	A B C
P_0	0 1 0	0 0 0	0 0 0
P_1	2 0 0	2 0 2	
P_2	3 0 3	0 0 0	
P_3	2 1 1	1 0 0	
P_4	0 0 2	0 0 2	

- Now suppose that process P2 makes a request for an additional instance of type C, yielding the state shown below. Is the system now deadlocked?

	<u>Allocation</u>	<u>Request</u>	<u>Available</u>
	A B C	A B C	A B C
P ₀	0 1 0	0 0 0	0 0 0
P ₁	2 0 0	2 0 2	
P ₂	3 0 3	0 0 1	
P ₃	2 1 1	1 0 0	
P ₄	0 0 2	0 0 2	

7.6.3 Detection-Algorithm Usage

- When should the deadlock detection be done? Frequently, or infrequently?
- The answer may depend on how frequently deadlocks are expected to occur, as well as the possible consequences of not catching them immediately. (If deadlocks are not removed immediately when they occur, then more and more processes can "back up" behind the deadlock, making the eventual task of unblocking the system more difficult and possibly damaging to more processes.)
- There are two obvious approaches, each with trade-offs:
 1. Do deadlock detection after every resource allocation which cannot be immediately granted. This has the advantage of detecting the deadlock right away, while the minimum number of processes are involved in the deadlock. (One might consider that the process whose request triggered the deadlock condition is the "cause" of the deadlock, but realistically all of the processes in the cycle are equally responsible for the resulting deadlock.) The down side of this approach is the extensive overhead and performance hit caused by checking for deadlocks so frequently.
 2. Do deadlock detection only when there is some clue that a deadlock may have occurred, such as when CPU utilization reduces to 40% or some other magic number. The advantage is that deadlock detection is done much less frequently, but the down side is that it becomes impossible to detect the processes involved in the original deadlock, and so deadlock recovery can be more complicated and damaging to more processes.
 3. (As I write this, a third alternative comes to mind: Keep a historical log of resource allocations, since that last known time of no deadlocks. Do

deadlock checks periodically (once an hour or when CPU usage is low?), and then use the historical log to trace through and determine when the deadlock occurred and what processes caused the initial deadlock. Unfortunately I'm not certain that breaking the original deadlock would then free up the resulting log jam.)

7.7 Recovery From Deadlock

- There are three basic approaches to recovery from deadlock:
 1. Inform the system operator, and allow him/her to take manual intervention.
 2. Terminate one or more processes involved in the deadlock
 3. Preempt resources.

7.7.1 Process Termination

- Two basic approaches, both of which recover resources allocated to terminated processes:
 - Terminate all processes involved in the deadlock. This definitely solves the deadlock, but at the expense of terminating more processes than would be absolutely necessary.
 - Terminate processes one by one until the deadlock is broken. This is more conservative, but requires doing deadlock detection after each step.
- In the latter case there are many factors that can go into deciding which processes to terminate next:
 1. Process priorities.
 2. How long the process has been running, and how close it is to finishing.
 3. How many and what type of resources is the process holding. (Are they easy to preempt and restore?)
 4. How many more resources does the process need to complete.
 5. How many processes will need to be terminated
 6. Whether the process is interactive or batch.
 7. (Whether or not the process has made non-restorable changes to any resource.)

7.7.2 Resource Preemption

- When preempting resources to relieve deadlock, there are three important issues to be addressed:

1. **Selecting a victim** - Deciding which resources to preempt from which processes involves many of the same decision criteria outlined above.
2. **Rollback** - Ideally one would like to roll back a preempted process to a safe state prior to the point at which that resource was originally allocated to the process. Unfortunately it can be difficult or impossible to determine what such a safe state is, and so the only safe rollback is to roll back all the way back to the beginning. (I.e. abort the process and make it start over.)
3. **Starvation** - How do you guarantee that a process won't starve because its resources are constantly being preempted? One option would be to use a priority system, and increase the priority of a process every time its resources get preempted. Eventually it should get a high enough priority that it won't get preempted any more.

File-System Implementation

12.1 File-System Structure

- Hard disks have two important properties that make them suitable for secondary storage of files in file systems: (1) Blocks of data can be rewritten in place, and (2) they are direct access, allowing any block of data to be accessed with only (relatively) minor movements of the disk heads and rotational latency. (See Chapter 12)
- Disks are usually accessed in physical blocks, rather than a byte at a time. Block sizes may range from 512 bytes to 4K or larger.
- 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.
 - **I/O Control** consists of **device drivers**, special software programs (often written in assembly) 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,

- a.k.a. *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. Depending on the system, blocks may be referred to with a single block number, (e.g. block # 234234), or with head-sector-cylinder combinations.
 - 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. 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 (among 40 others supported.)

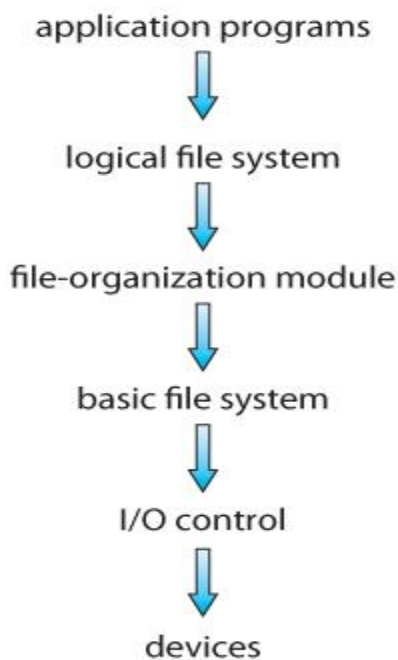


Figure 12.1 - Layered file system.

12.2 File-System Implementation

12.2.1 Overview

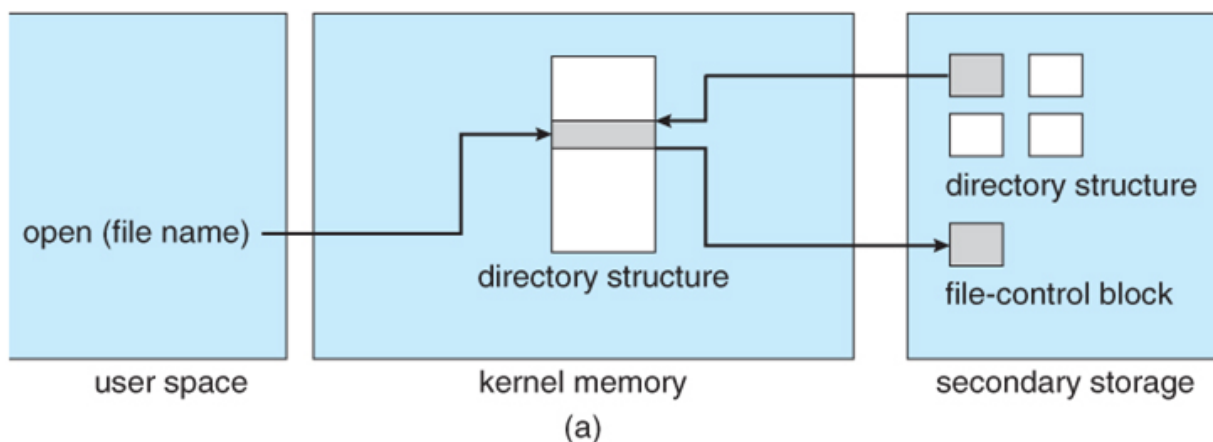
- File systems store several important data structures on the disk:
 - A **boot-control block**, (per volume) a.k.a. the **boot block** in UNIX or the **partition boot sector** in Windows contains information about how to boot the system off of this disk. This will generally be the first sector of the volume if there is a bootable system loaded on that volume, or the block will be left vacant otherwise.
 - A **volume control block**, (per volume) a.k.a. the **master file table** in UNIX or the **superblock** in Windows, which contains information such as the partition table, number of blocks on each filesystem, and pointers to free blocks and free FCB blocks.
 - 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.

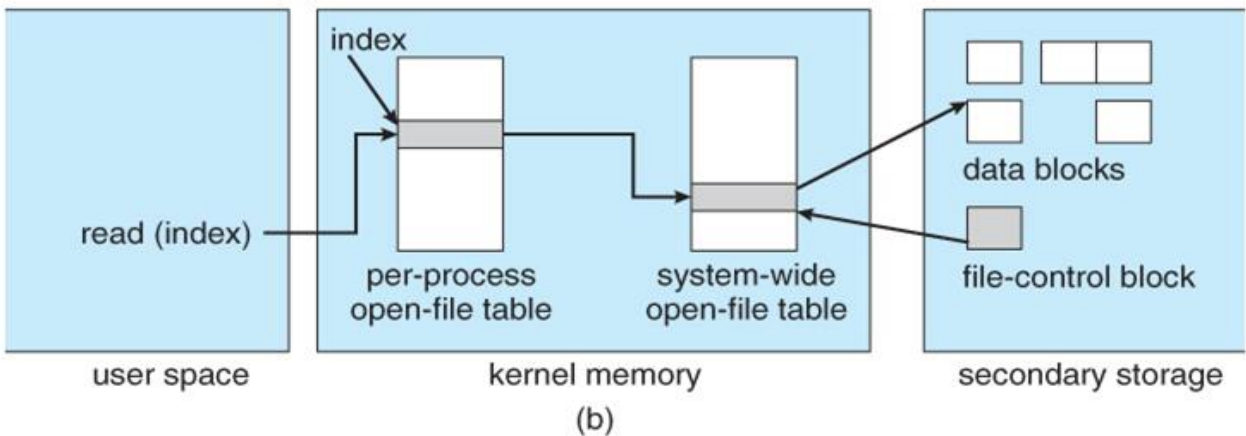
file permissions
file dates (create, access, write)
file owner, group, ACL
file size
file data blocks or pointers to file data blocks

Figure 12.2 - A typical file-control block.

- There are also several key data structures stored in memory:
 - An in-memory mount table.
 - 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.)
- Figure 12.3 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. The appropriate directory is modified with the new file name and FCB information.
 - 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.





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

12.2.2 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 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.
- 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.

12.2.3 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 a 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. See /usr/include/linux/fs.h for full details. Common operations provided include open(), read(), write(), and mmap().

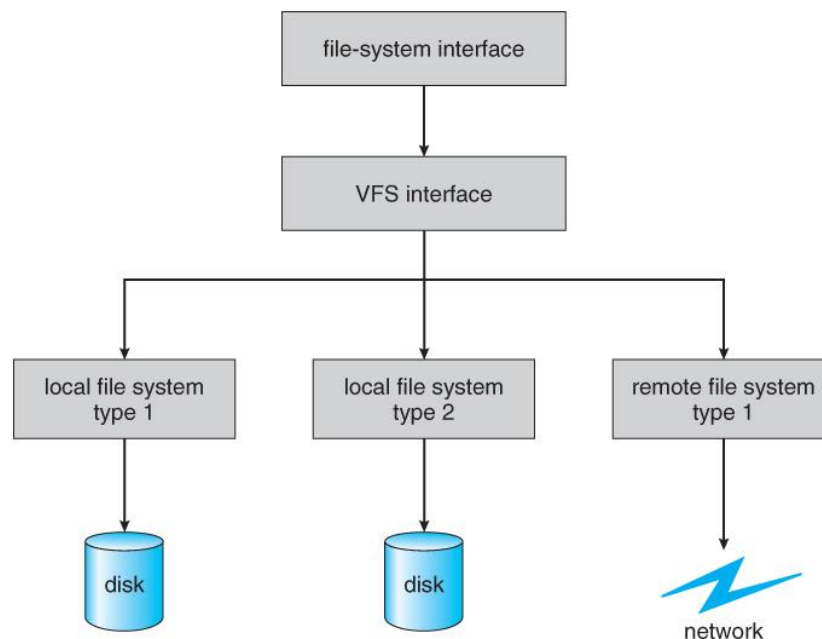


Figure 12.4 - Schematic view of a virtual file system.

12.3 Directory Implementation

- Directories need to be fast to search, insert, and delete, with a minimum of wasted disk space.

12.3.1 Linear List

- A linear list is the simplest and easiest directory structure to set up, but it does have some drawbacks.
- Finding a file (or verifying one does not already exist upon creation) requires a linear search.
- Deletions can be done by moving all entries, flagging an entry as deleted, or by moving the last entry into the newly vacant position.
- Sorting the list makes searches faster, at the expense of more complex insertions and deletions.
- A linked list makes insertions and deletions into a sorted list easier, with overhead for the links.
- More complex data structures, such as B-trees, could also be considered.

12.3.2 Hash Table

- A hash table can also be used to speed up searches.
- Hash tables are generally implemented *in addition to* a linear or other structure

12.4 Allocation Methods

- There are three major methods of storing files on disks: contiguous, linked, and indexed.

12.4.1 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 involves the same issues discussed earlier for the allocation of contiguous blocks of memory (first fit, best fit, fragmentation problems, etc.) The distinction is that the high time penalty required for moving the disk heads from spot to spot may now justify the benefits of keeping files contiguously when possible.

- (Even file systems that do not by default store files contiguously can benefit from certain utilities that compact the disk and make all files contiguous in the process.)
- 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 is allocated. (For example an extent may be the size of a complete track or even cylinder, aligned on an appropriate track or cylinder boundary.) The high-performance files system Veritas uses extents to optimize performance.

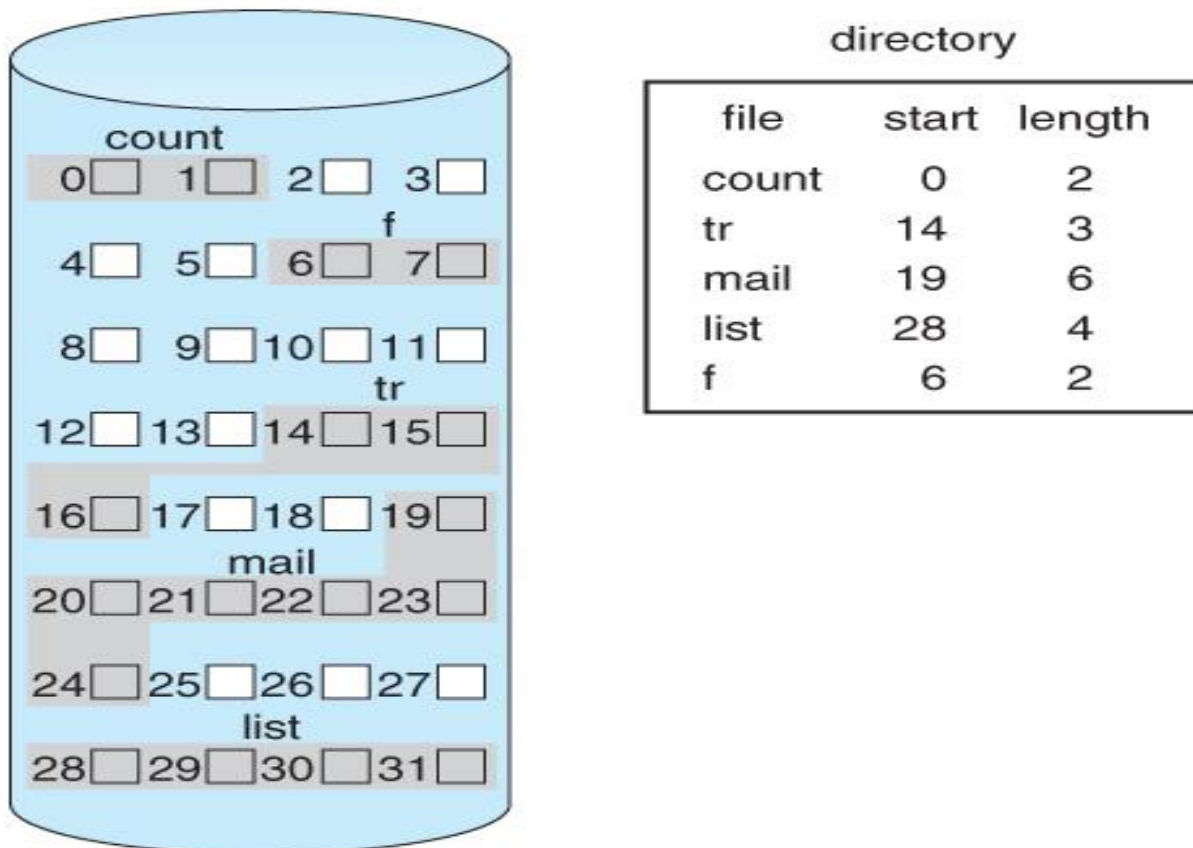


Figure 12.5 - Contiguous allocation of disk space.

12.4.2 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.

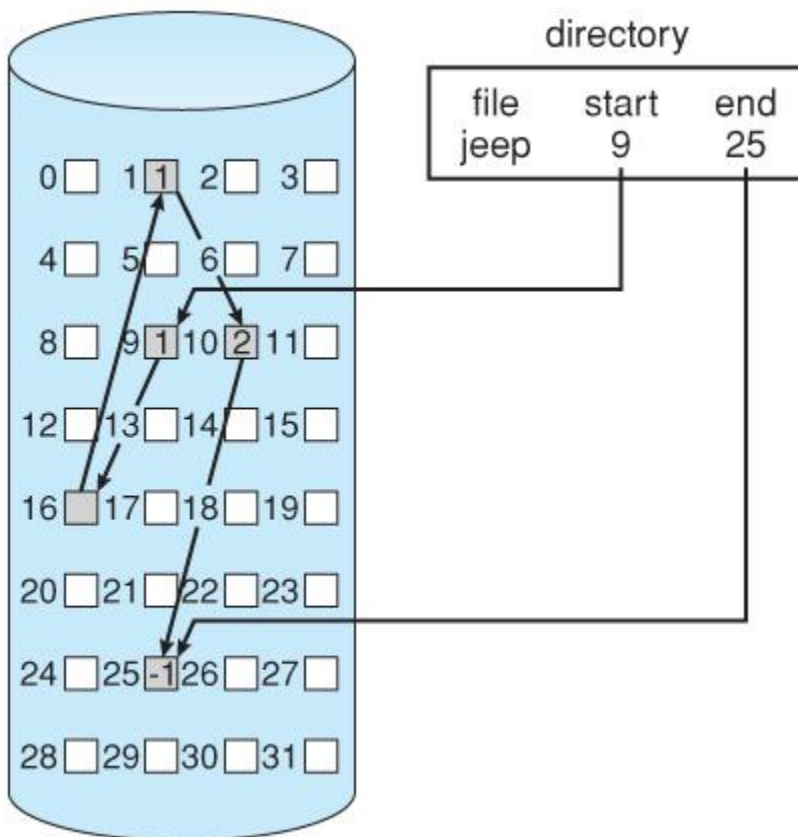


Figure 12.6 - Linked allocation of disk space.

- 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.

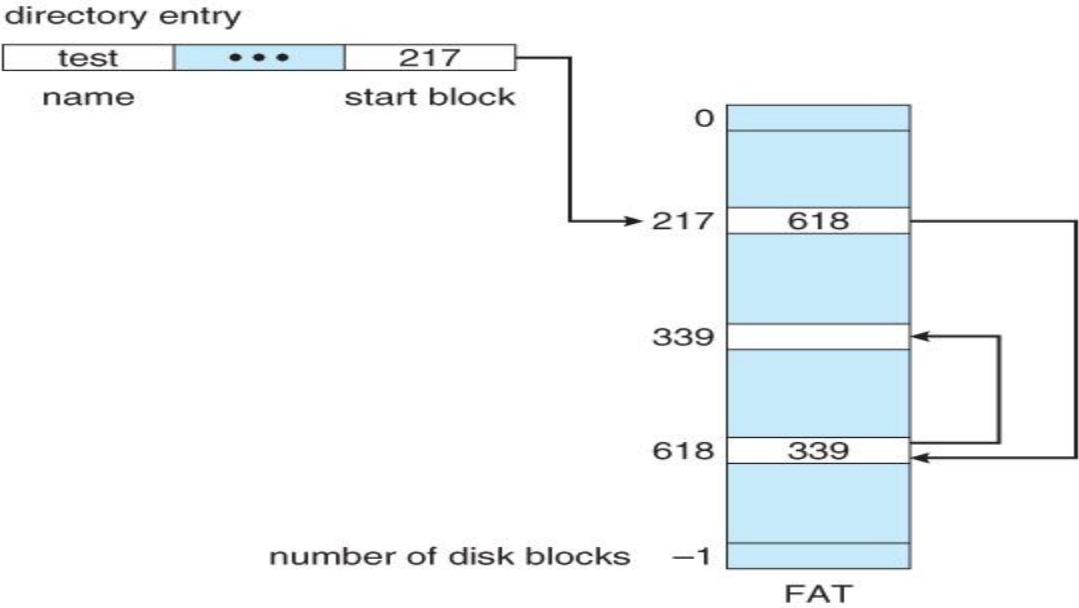


Figure 12.7 File-allocation table.

12.4.3 Indexed Allocation

- **Indexed Allocation** combines all of the indexes for accessing each file into a common block (for that file), as opposed to spreading them all over the disk or storing them in a FAT table.

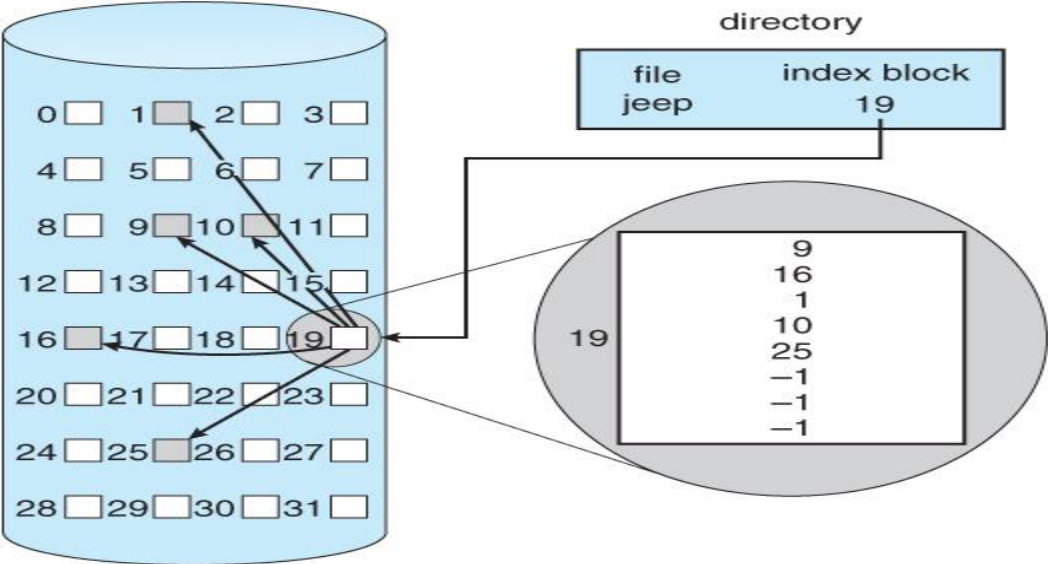


Figure 12.8 - Indexed allocation of disk space.

- 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 or so 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. (See below.) The advantage of this scheme is that for small files (which many are), 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.)

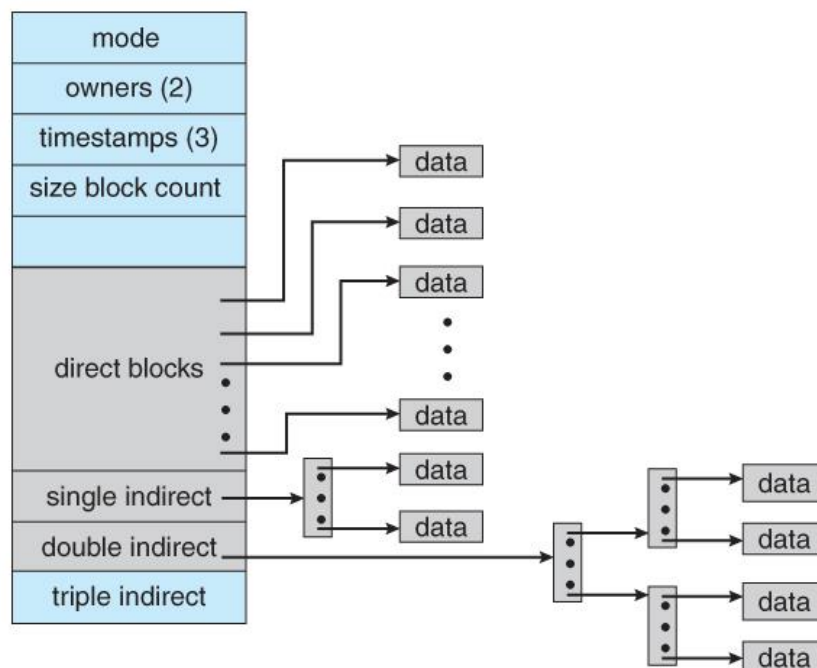


Figure 12.9 - The UNIX inode.

12.4.4 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.

12.5 Free-Space Management

- Another important aspect of disk management is keeping track of and allocating free space.

12.5.1 Bit Vector

- 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.
- Fast algorithms exist for quickly finding contiguous blocks of a given size
- The down side is that a 40GB disk requires over 5MB just to store the bitmap. (For example.)

12.5.2 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.

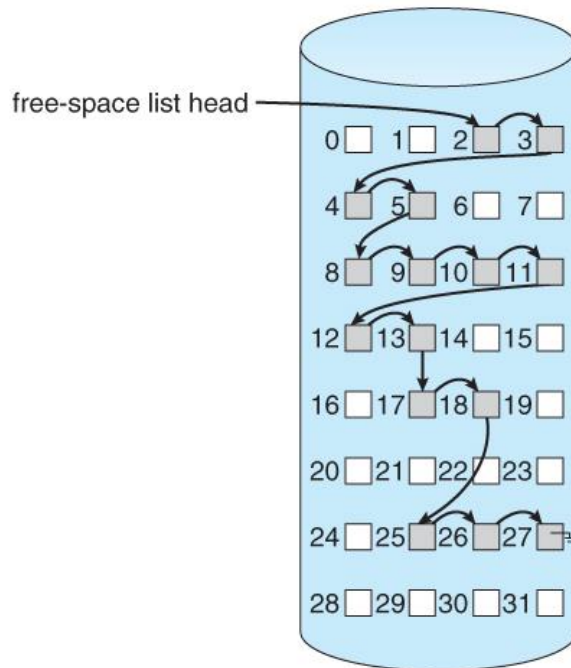


Figure 12.10 - Linked free-space list on disk.

12.5.3 Grouping

- A variation on linked list free lists is to use links of blocks of indices of free blocks. If a block holds up to N addresses, then the first block in the linked-list contains up to N-1 addresses of free blocks and a pointer to the next block of free addresses.

12.5.4 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. As long as the average length of a contiguous group of free blocks is greater than two this offers a savings in space needed for the free list. (Similar to compression techniques used for graphics images when a group of pixels all the same color is encountered.)

12.5.5 Space Maps

- 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 VERY 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.

12.6 Efficiency and Performance

12.6.1 Efficiency

- UNIX pre-allocates inodes, which occupies space even before any files are created.
- UNIX also distributes inodes across the disk, and tries to store data files near their inode, to reduce the distance of disk seeks between the inodes and the data.
- Some systems use variable size clusters depending on the file size.
- The more data that is stored in a directory (e.g. last access time), the more often the directory blocks have to be re-written.
- As technology advances, addressing schemes have had to grow as well.
 - Sun's ZFS file system uses 128-bit pointers, which should theoretically never need to be expanded. (The mass required to store 2^{128} bytes with atomic storage would be at least 272 trillion kilograms!)
- Kernel table sizes used to be fixed, and could only be changed by rebuilding the kernels. Modern tables are dynamically allocated, but that requires more complicated algorithms for accessing them.

12.6.2 Performance

- Disk controllers generally include on-board caching. When a seek is requested, the heads are moved into place, and then an entire track is read, starting from whatever sector is currently under the heads (reducing

latency.) The requested sector is returned and the unrequested portion of the track is cached in the disk's electronics.

- Some OSes cache disk blocks they expect to need again in a **buffer cache**.
- A **page cache** connected to the virtual memory system is actually more efficient as memory addresses do not need to be converted to disk block addresses and back again.
- Some systems (Solaris, Linux, Windows 2000, NT, XP) use page caching for both process pages and file data in a **unified virtual memory**.
- Figures 11.11 and 11.12 show the advantages of the **unified buffer cache** found in some versions of UNIX and Linux - Data does not need to be stored twice, and problems of inconsistent buffer information are avoided.

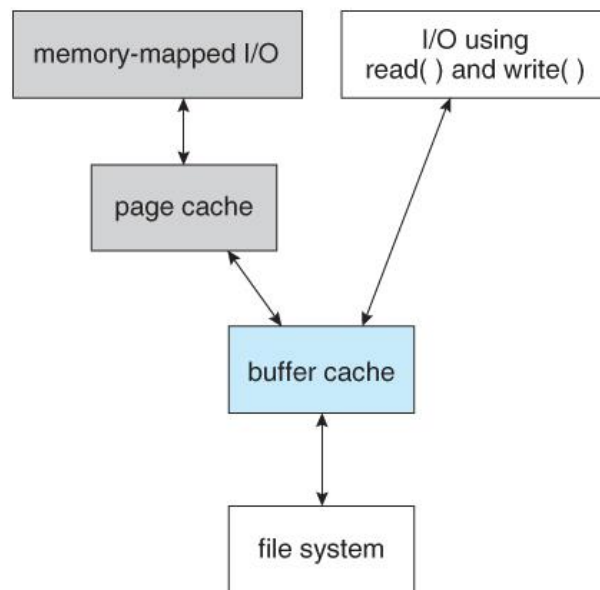


Figure 12.11 - I/O without a unified buffer cache.

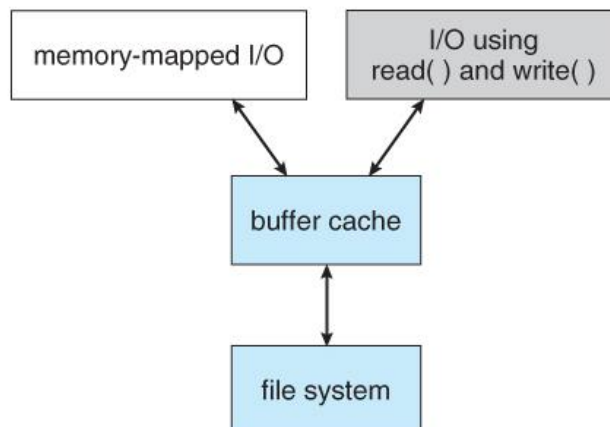


Figure 12.12 - I/O using a unified buffer cache.

- Page replacement strategies can be complicated with a unified cache, as one needs to decide whether to replace process or file pages, and how many pages to guarantee to each category of pages. Solaris, for example, has gone through many variations, resulting in *priority paging* giving process pages priority over file I/O pages, and setting limits so that neither can knock the other completely out of memory.
- Another issue affecting performance is the question of whether to implement *synchronous writes* or *asynchronous writes*. Synchronous writes occur in the order in which the disk subsystem receives them, without caching; Asynchronous writes are cached, allowing the disk subsystem to schedule writes in a more efficient order (See Chapter 12.) Metadata writes are often done synchronously. Some systems support flags to the open call requiring that writes be synchronous, for example for the benefit of database systems that require their writes be performed in a required order.
- The type of file access can also have an impact on optimal page replacement policies. For example, LRU is not necessarily a good policy for sequential access files. For these types of files progression normally goes in a forward direction only, and the most recently used page will not be needed again until after the file has been rewound and re-read from the beginning, (if it is ever needed at all.) On the other hand, we can expect to need the next page in the file fairly soon. For this reason sequential access files often take advantage of two special policies:
 - *Free-behind* frees up a page as soon as the next page in the file is requested, with the assumption that we are now done with the old page and won't need it again for a long time.
 - *Read-ahead* reads the requested page and several subsequent pages at the same time, with the assumption that those pages will be needed in the near future. This is similar to the track caching that is already performed by the disk controller, except it saves the future latency of transferring data from the disk controller memory into motherboard main memory.
- The caching system and asynchronous writes speed up disk writes considerably, because the disk subsystem can schedule physical writes to the disk to minimize head movement and disk seek times. (See Chapter 12.) Reads, on the other hand, must be done more synchronously in spite of the caching system, with the result that disk writes can counter-intuitively be much faster on average than disk reads.

12.7 Recovery

12.7.1 Consistency Checking

- The storing of certain data structures (e.g. directories and inodes) in memory and the caching of disk operations can speed up performance, but what happens in the result of a system crash? All volatile memory structures are lost, and the information stored on the hard drive may be left in an inconsistent state.
- A **Consistency Checker** (fsck in UNIX, chkdsk or scandisk in Windows) is often run at boot time or mount time, particularly if a filesystem was not closed down properly. Some of the problems that these tools look for include:
 - Disk blocks allocated to files and also listed on the free list.
 - Disk blocks neither allocated to files nor on the free list.
 - Disk blocks allocated to more than one file.
 - The number of disk blocks allocated to a file inconsistent with the file's stated size.
 - Properly allocated files / inodes which do not appear in any directory entry.
 - Link counts for an inode not matching the number of references to that inode in the directory structure.
 - Two or more identical file names in the same directory.
 - Illegally linked directories, e.g. cyclical relationships where those are not allowed, or files/directories that are not accessible from the root of the directory tree.
 - Consistency checkers will often collect questionable disk blocks into new files with names such as chk00001.dat. These files may contain valuable information that would otherwise be lost, but in most cases they can be safely deleted, (returning those disk blocks to the free list.)
- UNIX caches directory information for reads, but any changes that affect space allocation or metadata changes are written synchronously, before any of the corresponding data blocks are written to.

12.7.2 Log-Structured File Systems (was 11.8)

- **Log-based transaction-oriented** (a.k.a. **journaling**) filesystems borrow techniques developed for databases, guaranteeing that any given transaction either completes successfully or can be rolled back to a safe state before the transaction commenced:
 - All metadata changes are written sequentially to a log.
 - A set of changes for performing a specific task (e.g. moving a file) is a **transaction**.
 - As changes are written to the log they are said to be **committed**, allowing the system to return to its work.

- In the meantime, the changes from the log are carried out on the actual filesystem, and a pointer keeps track of which changes in the log have been completed and which have not yet been completed.
- When all changes corresponding to a particular transaction have been completed, that transaction can be safely removed from the log.
- At any given time, the log will contain information pertaining to uncompleted transactions only, e.g. actions that were committed but for which the entire transaction has not yet been completed.
 - From the log, the remaining transactions can be completed,
 - or if the transaction was aborted, then the partially completed changes can be undone.

12.7.3 Other Solutions (New)

- Sun's ZFS and Network Appliance's WAFL file systems take a different approach to file system consistency.
- No blocks of data are ever over-written in place. Rather the new data is written into fresh new blocks, and after the transaction is complete, the metadata (data block pointers) is updated to point to the new blocks.
 - The old blocks can then be freed up for future use.
 - Alternatively, if the old blocks and old metadata are saved, then a *snapshot* of the system in its original state is preserved. This approach is taken by WAFL.
- ZFS combines this with check-summing of all metadata and data blocks, and RAID, to ensure that no inconsistencies are possible, and therefore ZFS does not incorporate a consistency checker.

12.7.4 Backup and Restore

- In order to recover lost data in the event of a disk crash, it is important to conduct backups regularly.
- Files should be copied to some removable medium, such as magnetic tapes, CDs, DVDs, or external removable hard drives.
- A full backup copies every file on a filesystem.
- Incremental backups copy only files which have changed since some previous time.
- A combination of full and incremental backups can offer a compromise between full recoverability, the number and size of backup tapes needed, and the number of tapes that need to be used to do a full restore. For example, one strategy might be:
 - At the beginning of the month do a full backup.

- At the end of the first and again at the end of the second week, backup all files which have changed since the beginning of the month.
- At the end of the third week, backup all files that have changed since the end of the second week.
- Every day of the month not listed above, do an incremental backup of all files that have changed since the most recent of the weekly backups described above.
- Backup tapes are often reused, particularly for daily backups, but there are limits to how many times the same tape can be used.
- Every so often a full backup should be made that is kept "forever" and not overwritten.
- ***Backup tapes should be tested, to ensure that they are readable!***
- For optimal security, backup tapes should be kept off-premises, so that a fire or burglary cannot destroy both the system and the backups. There are companies (e.g. Iron Mountain) that specialize in the secure off-site storage of critical backup information.
- ***Keep your backup tapes secure - The easiest way for a thief to steal all your data is to simply pocket your backup tapes!***
- Storing important files on more than one computer can be an alternate though less reliable form of backup.
- Note that incremental backups can also help users to get back a previous version of a file that they have since changed in some way.
- Beware that backups can help forensic investigators recover e-mails and other files that users had though they had deleted!

12.8 NFS (Optional)

12.8.1 Overview

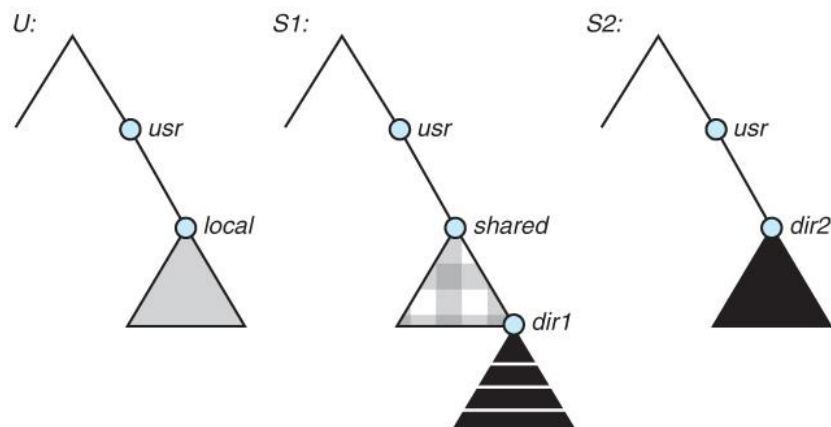


Figure 12.13 - Three independent file systems.

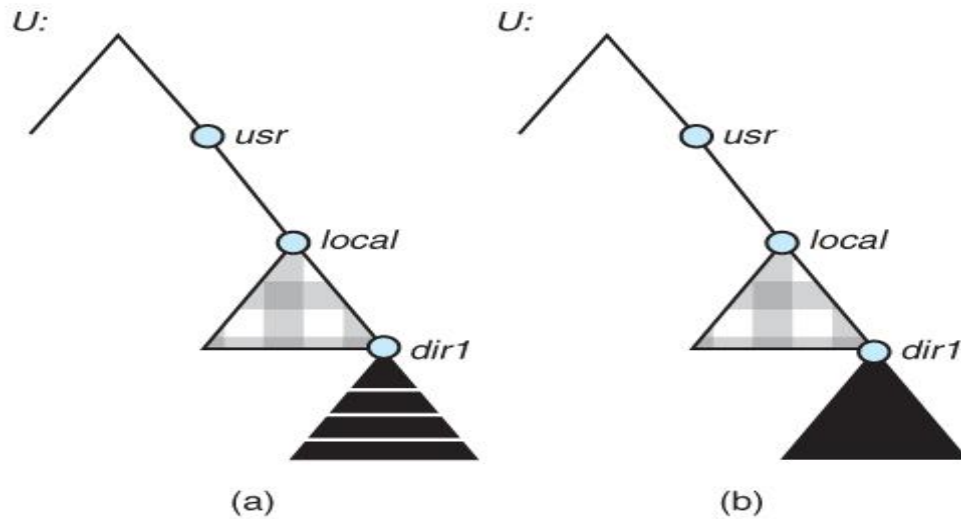


Figure 12.14 - Mounting in NFS. (a) Mounts. (b) Cascading mounts.

12.8.2 The Mount Protocol

- The NFS mount protocol is similar to the local mount protocol, establishing a connection between a specific local directory (the mount point) and a specific device from a remote system.
- Each server maintains an *export list* of the local filesystems (directory sub-trees) which are exportable, who they are exportable to, and what restrictions apply (e.g. read-only access.)
- The server also maintains a list of currently connected clients, so that they can be notified in the event of the server going down and for other reasons.
- Automount and autounmount are supported.

12.8.3 The NFS Protocol

- Implemented as a set of remote procedure calls (RPCs):
 - Searching for a file in a directory
 - REading a set of directory entries
 - Manipulating links and directories
 - Accessing file attributes
 - Reading and writing files

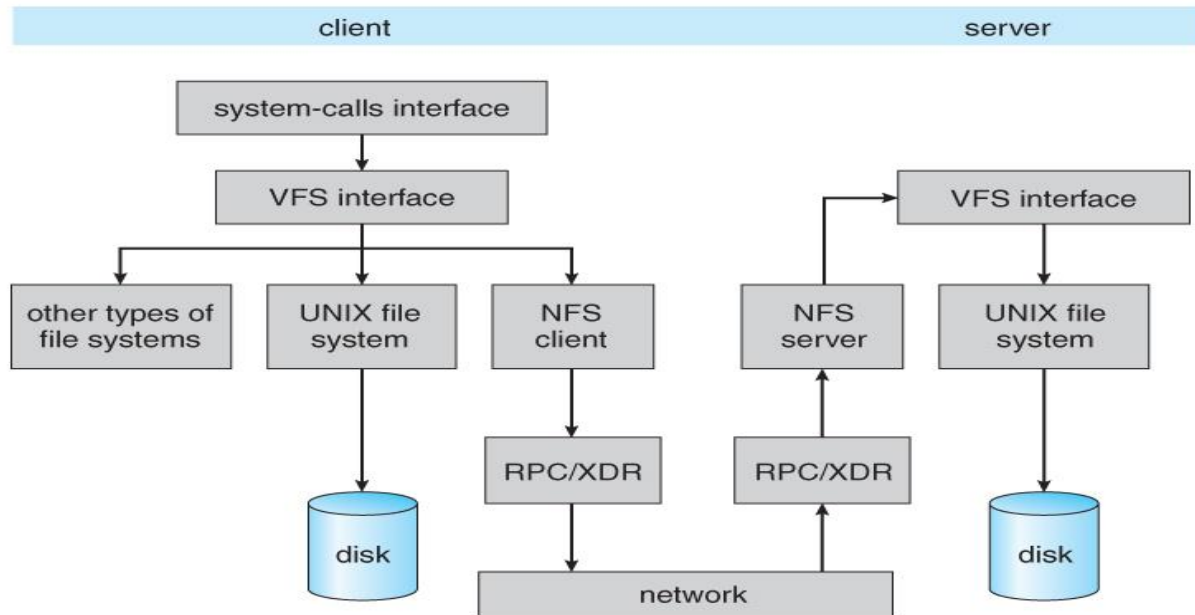


Figure 12.15 - Schematic view of the NFS architecture.

12.8.4 Path-Name Translation

11.8.5 Remote Operations

- Buffering and caching improve performance, but can cause a disparity in local versus remote views of the same file(s).

12.9 Example: The WAFL File System (Optional)

- Write Anywhere File Layout
- Designed for a specific hardware architecture.
- **Snapshots** record the state of the system at regular or irregular intervals.
 - The snapshot just copies the inode pointers, not the actual data.
 - Used pages are not overwritten, so updates are fast.
 - Blocks keep counters for how many snapshots are pointing to that block - When the counter reaches zero, then the block is considered free.

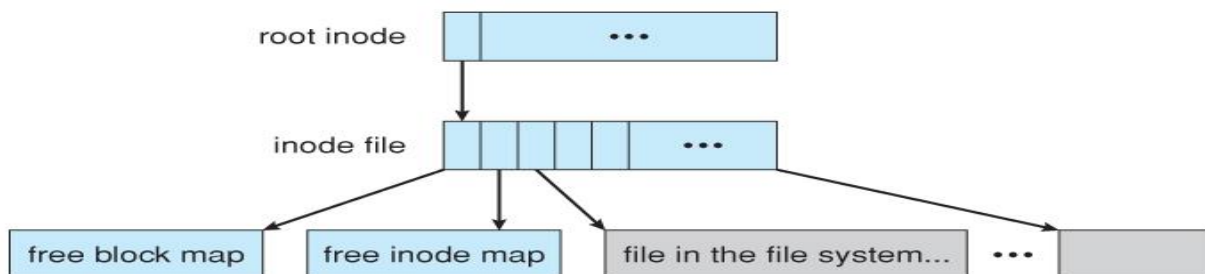
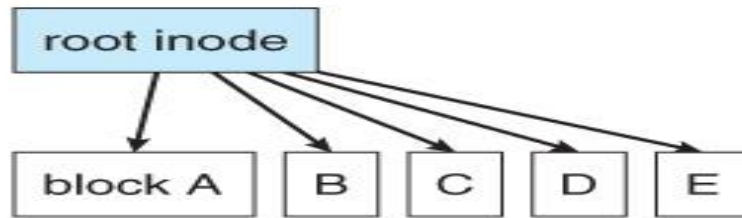
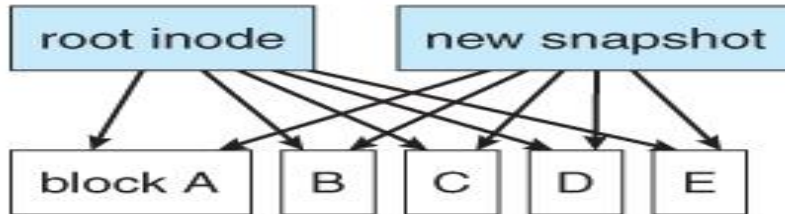


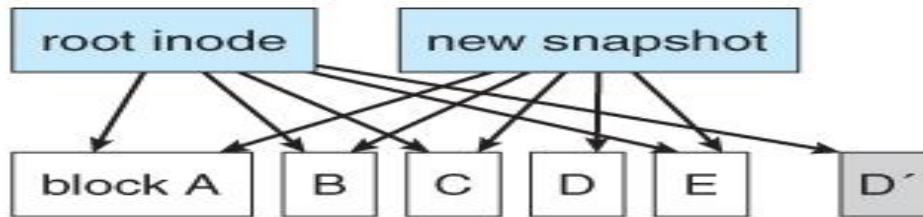
Figure 12.16 - The WAFL file layout.



(a) Before a snapshot.



(b) After a snapshot, before any blocks change.



(c) After block D has changed to D'.

Figure 12.17 - Snapshots in WAFL

Mass-Storage Structure

10.1 Overview of Mass-Storage Structure

10.1.1 Magnetic Disks

- Traditional magnetic disks have the following basic structure:
 - One or more **platters** in the form of disks covered with magnetic media. **Hard disk** platters are made of rigid metal, while "**floppy**" disks are made of more flexible plastic.
 - Each platter has two working **surfaces**. Older hard disk drives would sometimes not use the very top or bottom surface of a stack of platters, as these surfaces were more susceptible to potential damage.
 - Each working surface is divided into a number of concentric rings called **tracks**. The collection of all tracks that are the same distance from the edge of the platter, (i.e. all tracks immediately above one another in the following diagram) is called a **cylinder**.
 - Each track is further divided into **sectors**, traditionally containing 512 bytes of data each, although some modern disks occasionally use larger sector sizes. (Sectors also include a header and a trailer, including checksum information among other things. Larger sector sizes reduce the fraction of the disk consumed by headers and trailers, but increase internal fragmentation and the amount of disk that must be marked bad in the case of errors.)
 - The data on a hard drive is read by read-write **heads**. The standard configuration (shown below) uses one head per surface, each on a separate **arm**, and controlled by a common **arm assembly** which moves all heads simultaneously from one cylinder to another. (Other configurations, including independent read-write heads, may speed up disk access, but involve serious technical difficulties.)
 - The storage capacity of a traditional disk drive is equal to the number of heads (i.e. the number of working surfaces), times the number of tracks per surface, times the number of sectors per track, times the number of bytes per sector. A particular physical block of data is specified by providing the head-sector-cylinder number at which it is located.

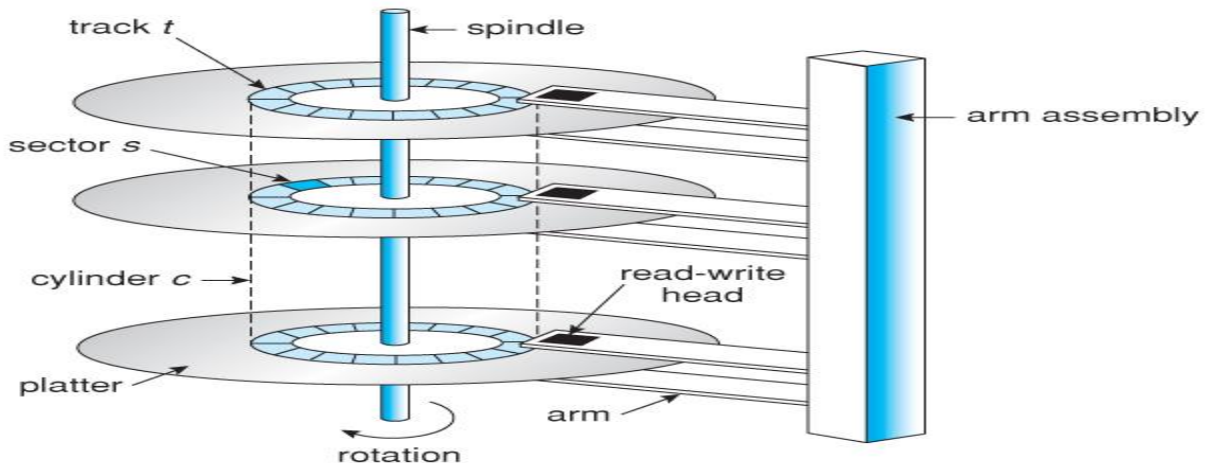


Figure 10.1 - Moving-head disk mechanism.

- In operation the disk rotates at high speed, such as 7200 rpm (120 revolutions per second.) The rate at which data can be transferred from the disk to the computer is composed of several steps:
 - The **positioning time**, a.k.a. the **seek time** or **random access time** is the time required to move the heads from one cylinder to another, and for the heads to settle down after the move. This is typically the slowest step in the process and the predominant bottleneck to overall transfer rates.
 - The **rotational latency** is the amount of time required for the desired sector to rotate around and come under the read-write head. This can range anywhere from zero to one full revolution, and on the average will equal one-half revolution. This is another physical step and is usually the second slowest step behind seek time. (For a disk rotating at 7200 rpm, the average rotational latency would be $1/2$ revolution / 120 revolutions per second, or just over 4 milliseconds, a long time by computer standards.
 - The **transfer rate**, which is the time required to move the data electronically from the disk to the computer. (Some authors may also use the term transfer rate to refer to the overall transfer rate, including seek time and rotational latency as well as the electronic data transfer rate.)
- Disk heads "fly" over the surface on a very thin cushion of air. If they should accidentally contact the disk, then a **head crash** occurs, which may or may not permanently damage the disk or even destroy it completely. For this reason it is normal to **park** the disk heads when turning a computer off, which means to move the heads off the disk or to an area of the disk where there is no data stored.
- Floppy disks are normally **removable**. Hard drives can also be removable, and some are even **hot-swappable**, meaning they can be removed while the computer is running, and a new hard drive inserted in their place.

- Disk drives are connected to the computer via a cable known as the **I/O Bus**. Some of the common interface formats include Enhanced Integrated Drive Electronics, EIDE; Advanced Technology Attachment, ATA; Serial ATA, SATA, Universal Serial Bus, USB; Fiber Channel, FC, and Small Computer Systems Interface, SCSI.
- The **host controller** is at the computer end of the I/O bus, and the **disk controller** is built into the disk itself. The CPU issues commands to the host controller via I/O ports. Data is transferred between the magnetic surface and onboard **cache** by the disk controller, and then the data is transferred from that cache to the host controller and the motherboard memory at electronic speeds.

10.1.2 Solid-State Disks - New

- As technologies improve and economics change, old technologies are often used in different ways. One example of this is the increasing used of **solid state disks, or SSDs**.
- SSDs use memory technology as a small fast hard disk. Specific implementations may use either flash memory or DRAM chips protected by a battery to sustain the information through power cycles.
- Because SSDs have no moving parts they are much faster than traditional hard drives, and certain problems such as the scheduling of disk accesses simply do not apply.
- However SSDs also have their weaknesses: They are more expensive than hard drives, generally not as large, and may have shorter life spans.
- SSDs are especially useful as a high-speed cache of hard-disk information that must be accessed quickly. One example is to store filesystem meta-data, e.g. directory and inode information, that must be accessed quickly and often. Another variation is a boot disk containing the OS and some application executables, but no vital user data. SSDs are also used in laptops to make them smaller, faster, and lighter.
- Because SSDs are so much faster than traditional hard disks, the throughput of the bus can become a limiting factor, causing some SSDs to be connected directly to the system PCI bus for example.

10.1.3 Magnetic Tapes - was 12.1.2

- Magnetic tapes were once used for common secondary storage before the days of hard disk drives, but today are used primarily for backups.
- Accessing a particular spot on a magnetic tape can be slow, but once reading or writing commences, access speeds are comparable to disk drives.
- Capacities of tape drives can range from 20 to 200 GB, and compression can double that capacity.

10.2 Disk Structure

- The traditional head-sector-cylinder, HSC numbers are mapped to linear block addresses by numbering the first sector on the first head on the outermost track as sector 0. Numbering proceeds with the rest of the sectors on that same track, and then the rest of the tracks on the same cylinder before proceeding through the rest of the cylinders to the center of the disk. In modern practice these linear block addresses are used in place of the HSC numbers for a variety of reasons:
 1. The linear length of tracks near the outer edge of the disk is much longer than for those tracks located near the center, and therefore it is possible to squeeze many more sectors onto outer tracks than onto inner ones.
 2. All disks have some bad sectors, and therefore disks maintain a few spare sectors that can be used in place of the bad ones. The mapping of spare sectors to bad sectors is managed internally to the disk controller.
 3. Modern hard drives can have thousands of cylinders, and hundreds of sectors per track on their outermost tracks. These numbers exceed the range of HSC numbers for many (older) operating systems, and therefore disks can be configured for any convenient combination of HSC values that falls within the total number of sectors physically on the drive.
- There is a limit to how closely packed individual bits can be placed on a physical media, but that limit is growing increasingly more packed as technological advances are made.
- Modern disks pack many more sectors into outer cylinders than inner ones, using one of two approaches:
 - With **Constant Linear Velocity, CLV**, the density of bits is uniform from cylinder to cylinder. Because there are more sectors in outer cylinders, the disk spins slower when reading those cylinders, causing the rate of bits passing under the read-write head to remain constant. This is the approach used by modern CDs and DVDs.
 - With **Constant Angular Velocity, CAV**, the disk rotates at a constant angular speed, with the bit density decreasing on outer cylinders. (These disks would have a constant number of sectors per track on all cylinders.)

10.3 Disk Attachment

Disk drives can be attached either directly to a particular host (a local disk) or to a network.

10.3.1 Host-Attached Storage

- Local disks are accessed through I/O Ports as described earlier.

- The most common interfaces are IDE or ATA, each of which allow up to two drives per host controller.
- SATA is similar with simpler cabling.
- High end workstations or other systems in need of larger number of disks typically use SCSI disks:
 - The SCSI standard supports up to 16 **targets** on each SCSI bus, one of which is generally the host adapter and the other 15 of which can be disk or tape drives.
 - A SCSI target is usually a single drive, but the standard also supports up to 8 **units** within each target. These would generally be used for accessing individual disks within a RAID array. (See below.)
 - The SCSI standard also supports multiple host adapters in a single computer, i.e. multiple SCSI busses.
 - Modern advancements in SCSI include "fast" and "wide" versions, as well as SCSI-2.
 - SCSI cables may be either 50 or 68 conductors. SCSI devices may be external as well as internal.
- FC is a high-speed serial architecture that can operate over optical fiber or four-conductor copper wires, and has two variants:
 - A large switched fabric having a 24-bit address space. This variant allows for multiple devices and multiple hosts to interconnect, forming the basis for the **storage-area networks, SANs**, to be discussed in a future section.
 - The **arbitrated loop, FC-AL**, that can address up to 126 devices (drives and controllers.)

10.3.2 Network-Attached Storage

- Network attached storage connects storage devices to computers using a remote procedure call, RPC, interface, typically with something like NFS filesystem mounts. This is convenient for allowing several computers in a group common access and naming conventions for shared storage.
- NAS can be implemented using SCSI cabling, or **ISCSI** uses Internet protocols and standard network connections, allowing long-distance remote access to shared files.
- NAS allows computers to easily share data storage, but tends to be less efficient than standard host-attached storage.

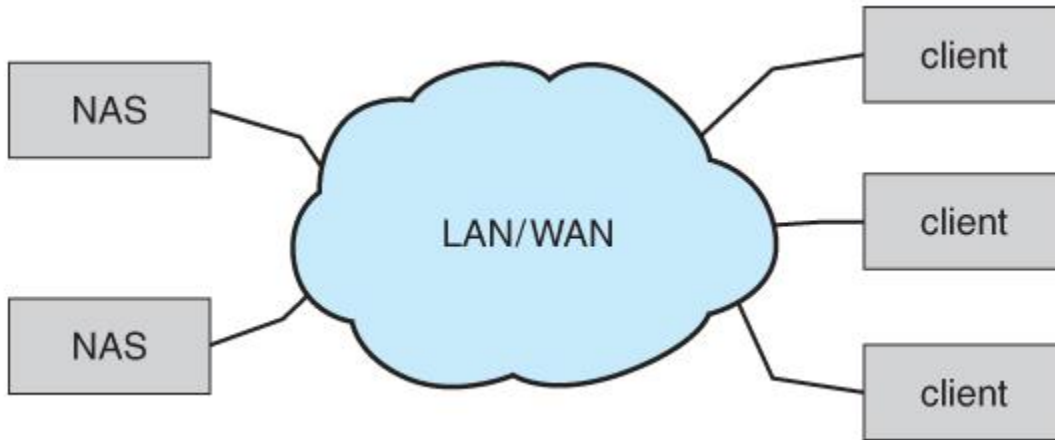


Figure 10.2 - Network-attached storage.

10.3.3 Storage-Area Network

- A **Storage-Area Network, SAN**, connects computers and storage devices in a network, using storage protocols instead of network protocols.
- One advantage of this is that storage access does not tie up regular networking bandwidth.
- SAN is very flexible and dynamic, allowing hosts and devices to attach and detach on the fly.
- SAN is also controllable, allowing restricted access to certain hosts and devices.

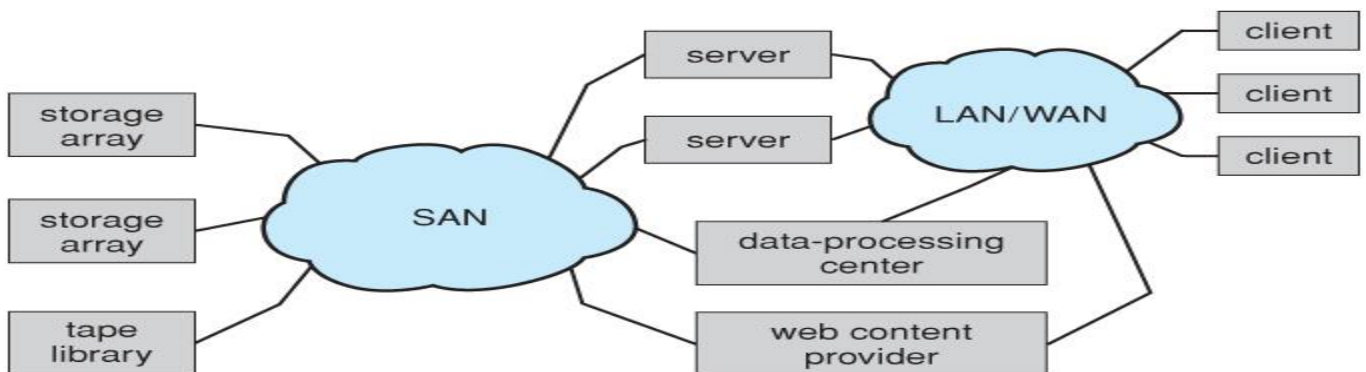


Figure 10.3 - Storage-area network.

10.4 Disk Scheduling

- As mentioned earlier, disk transfer speeds are limited primarily by **seek times** and **rotational latency**. When multiple requests are to be processed there is also some inherent delay in waiting for other requests to be processed.

- **Bandwidth** is measured by the amount of data transferred divided by the total amount of time from the first request being made to the last transfer being completed, (for a series of disk requests.)
- Both bandwidth and access time can be improved by processing requests in a good order.
- Disk requests include the disk address, memory address, number of sectors to transfer, and whether the request is for reading or writing.

10.4.1 FCFS Scheduling

- **First-Come First-Serve** is simple and intrinsically fair, but not very efficient. Consider in the following sequence the wild swing from cylinder 122 to 14 and then back to 124:



Figure 10.4 - FCFS disk scheduling.

10.4.2 SSTF Scheduling

- **Shortest Seek Time First** scheduling is more efficient, but may lead to starvation if a constant stream of requests arrives for the same general area of the disk.
- SSTF reduces the total head movement to 236 cylinders, down from 640 required for the same set of requests under FCFS. Note, however that the distance could be reduced still further to 208 by starting with 37 and then 14 first before processing the rest of the requests.

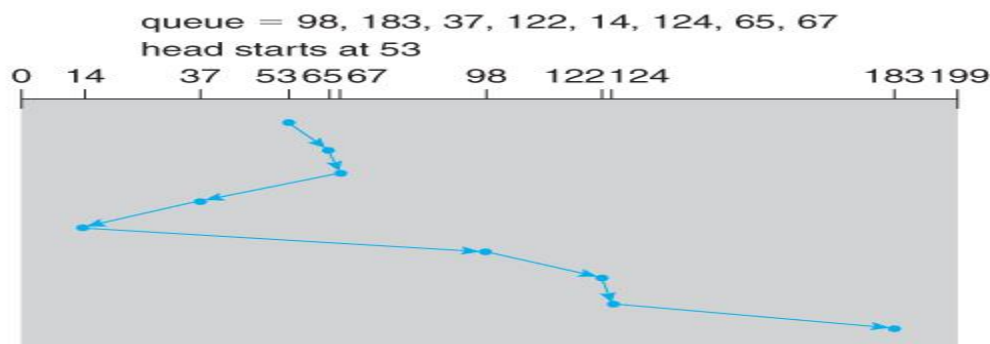


Figure 10.5 - SSTF disk scheduling.

10.4.3 SCAN Scheduling

- The **SCAN** algorithm, a.k.a. the **elevator** algorithm moves back and forth from one end of the disk to the other, similarly to an elevator processing requests in a tall building.

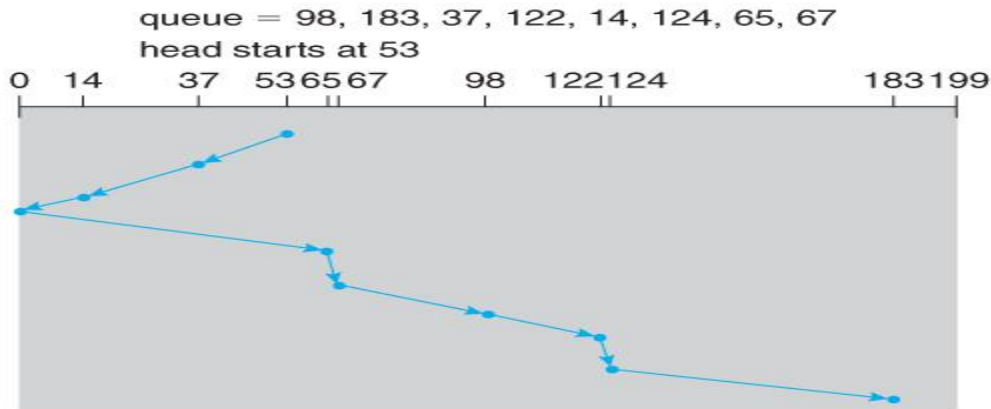


Figure 10.6 - SCAN disk scheduling.

- Under the SCAN algorithm, If a request arrives just ahead of the moving head then it will be processed right away, but if it arrives just after the head has passed, then it will have to wait for the head to pass going the other way on the return trip. This leads to a fairly wide variation in access times which can be improved upon.
- Consider, for example, when the head reaches the high end of the disk: Requests with high cylinder numbers just missed the passing head, which means they are all fairly recent requests, whereas requests with low numbers may have been waiting for a much longer time. Making the return scan from high to low then ends up accessing recent requests first and making older requests wait that much longer.

10.4.4 C-SCAN Scheduling

- The **Circular-SCAN** algorithm improves upon SCAN by treating all requests in a circular queue fashion - Once the head reaches the end of the disk, it returns to the other end without processing any requests, and then starts again from the beginning of the disk:

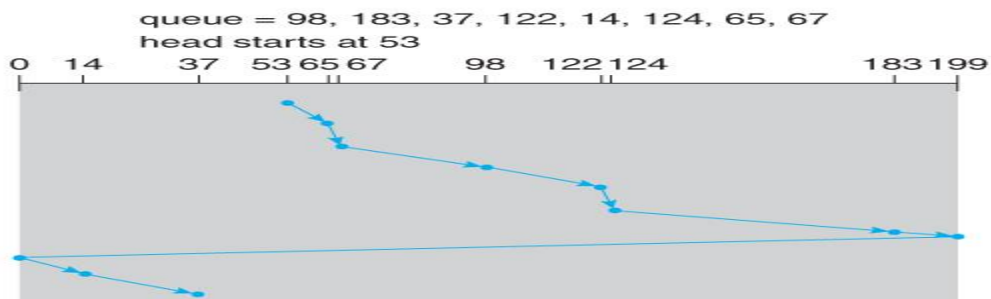


Figure 10.7 - C-SCAN disk scheduling.

12.4.5 LOOK Scheduling

- **LOOK** scheduling improves upon SCAN by looking ahead at the queue of pending requests, and not moving the heads any farther towards the end of the disk than is necessary. The following diagram illustrates the circular form of LOOK:

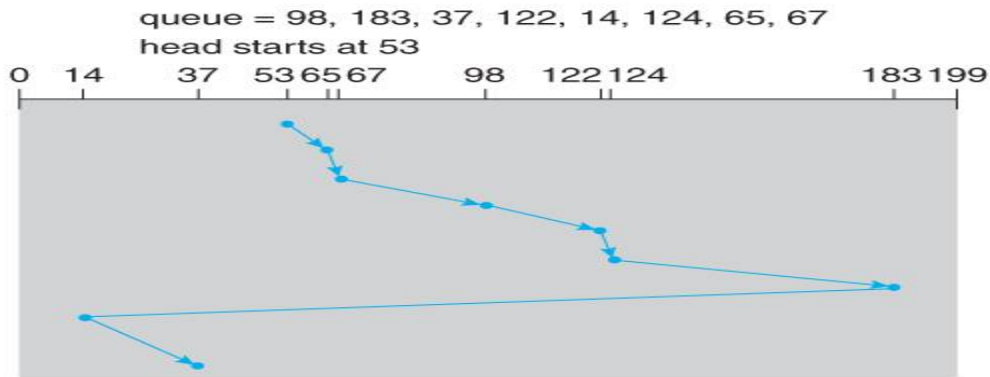


Figure 10.8 - C-LOOK disk scheduling.

10.4.6 Selection of a Disk-Scheduling Algorithm

- With very low loads all algorithms are equal, since there will normally only be one request to process at a time.
- For slightly larger loads, SSTF offers better performance than FCFS, but may lead to starvation when loads become heavy enough.
- For busier systems, SCAN and LOOK algorithms eliminate starvation problems.
- The actual optimal algorithm may be something even more complex than those discussed here, but the incremental improvements are generally not worth the additional overhead.
- Some improvement to overall filesystem access times can be made by intelligent placement of directory and/or inode information. If those structures are placed in the middle of the disk instead of at the beginning of the disk, then the maximum distance from those structures to data blocks is reduced to only one-half of the disk size. If those structures can be further distributed and furthermore have their data blocks stored as close as possible to the corresponding directory structures, then that reduces still further the overall time to find the disk block numbers and then access the corresponding data blocks.
- On modern disks the rotational latency can be almost as significant as the seek time, however it is not within the OSes control to account for that, because modern disks do not reveal their internal sector mapping schemes, (particularly when bad blocks have been remapped to spare sectors.)

- Some disk manufacturers provide for disk scheduling algorithms directly on their disk controllers, (which do know the actual geometry of the disk as well as any remapping), so that if a series of requests are sent from the computer to the controller then those requests can be processed in an optimal order.
- Unfortunately there are some considerations that the OS must take into account that are beyond the abilities of the on-board disk-scheduling algorithms, such as priorities of some requests over others, or the need to process certain requests in a particular order. For this reason Oses may elect to spoon-feed requests to the disk controller one at a time in certain situations.

10.7 RAID Structure

- The general idea behind RAID is to employ a group of hard drives together with some form of duplication, either to increase reliability or to speed up operations, (or sometimes both.)
- **RAID** originally stood for **Redundant Array of Inexpensive Disks**, and was designed to use a bunch of cheap small disks in place of one or two larger more expensive ones. Today RAID systems employ large possibly expensive disks as their components, switching the definition to **Independent** disks.

10.7.1 Improvement of Reliability via Redundancy

- The more disks a system has, the greater the likelihood that one of them will go bad at any given time. Hence increasing disks on a system actually **decreases** the **Mean Time To Failure, MTTF** of the system.
- If, however, the same data was copied onto multiple disks, then the data would not be lost unless **both** (or all) copies of the data were damaged simultaneously, which is a **MUCH** lower probability than for a single disk going bad. More specifically, the second disk would have to go bad before the first disk was repaired, which brings the **Mean Time To Repair** into play. For example if two disks were involved, each with a MTTF of 100,000 hours and a MTTR of 10 hours, then the **Mean Time to Data Loss** would be $500 * 10^6$ hours, or 57,000 years!
- This is the basic idea behind disk **mirroring**, in which a system contains identical data on two or more disks.
 - Note that a power failure during a write operation could cause both disks to contain corrupt data, if both disks were writing simultaneously at the time of the power failure. One solution is to write to the two disks in series, so that they will not both become corrupted (at least not in the same way) by a power failure. And alternate solution involves non-volatile RAM as a write cache, which is not lost in the event of a power failure and which is protected by error-correcting codes.

10.7.2 Improvement in Performance via Parallelism

- There is also a performance benefit to mirroring, particularly with respect to reads. Since every block of data is duplicated on multiple disks, read operations can be satisfied from any available copy, and multiple disks can be reading different data blocks simultaneously in parallel. (Writes could possibly be sped up as well through careful scheduling algorithms, but it would be complicated in practice.)
- Another way of improving disk access time is with **striping**, which basically means spreading data out across multiple disks that can be accessed simultaneously.
 - With **bit-level striping** the bits of each byte are striped across multiple disks. For example if 8 disks were involved, then each 8-bit byte would be read in parallel by 8 heads on separate disks. A single disk read would access 8 * 512 bytes = 4K worth of data in the time normally required to read 512 bytes. Similarly if 4 disks were involved, then two bits of each byte could be stored on each disk, for 2K worth of disk access per read or write operation.
 - **Block-level striping** spreads a filesystem across multiple disks on a block-by-block basis, so if block N were located on disk 0, then block N + 1 would be on disk 1, and so on. This is particularly useful when filesystems are accessed in **clusters** of physical blocks. Other striping possibilities exist, with block-level striping being the most common.

10.7.3 RAID Levels

- Mirroring provides reliability but is expensive; Striping improves performance, but does not improve reliability. Accordingly there are a number of different schemes that combine the principals of mirroring and striping in different ways, in order to balance reliability versus performance versus cost. These are described by different **RAID levels**, as follows: (In the diagram that follows, "C" indicates a copy, and "P" indicates parity, i.e. checksum bits.)
 1. **Raid Level 0** - This level includes striping only, with no mirroring.
 2. **Raid Level 1** - This level includes mirroring only, no striping.
 3. **Raid Level 2** - This level stores error-correcting codes on additional disks, allowing for any damaged data to be reconstructed by subtraction from the remaining undamaged data. Note that this scheme requires only three extra disks to protect 4 disks worth of data, as opposed to full mirroring. (The number of disks required is a function of the error-correcting algorithms, and the means by which the particular bad bit(s) is(are) identified.)
 4. **Raid Level 3** - This level is similar to level 2, except that it takes advantage of the fact that each disk is still doing its own error-detection, so that when an error occurs, there is no question about which disk in the array has the bad data. As a result a single parity bit is all that is needed to recover the lost data from an array of disks. Level 3 also includes striping, which improves performance. The downside with the parity approach

is that every disk must take part in every disk access, and the parity bits must be constantly calculated and checked, reducing performance. Hardware-level parity calculations and NVRAM cache can help with both of those issues. In practice level 3 is greatly preferred over level 2.

5. **Raid Level 4** - This level is similar to level 3, employing block-level striping instead of bit-level striping. The benefits are that multiple blocks can be read independently, and changes to a block only require writing two blocks (data and parity) rather than involving all disks. Note that new disks can be added seamlessly to the system provided they are initialized to all zeros, as this does not affect the parity results.
6. **Raid Level 5** - This level is similar to level 4, except the parity blocks are distributed over all disks, thereby more evenly balancing the load on the system. For any given block on the disk(s), one of the disks will hold the parity information for that block and the other N-1 disks will hold the data. Note that the same disk cannot hold both data and parity for the same block, as both would be lost in the event of a disk crash.
7. **Raid Level 6** - This level extends raid level 5 by storing multiple bits of error-recovery codes, (such as the [Reed-Solomon codes](#)), for each bit position of data, rather than a single parity bit. In the example shown below 2 bits of ECC are stored for every 4 bits of data, allowing data recovery in the face of up to two simultaneous disk failures. Note that this still involves only 50% increase in storage needs, as opposed to 100% for simple mirroring which could only tolerate a single disk failure.

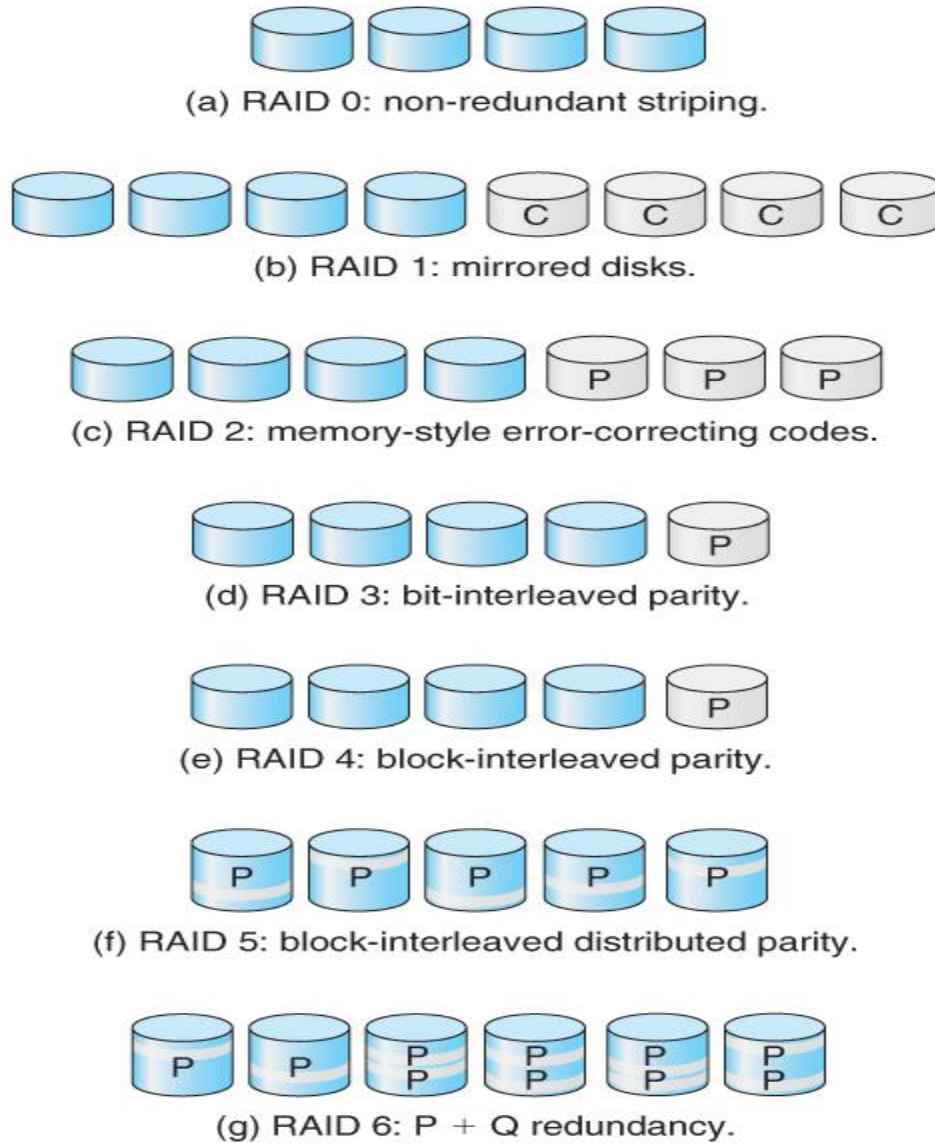


Figure 10.11 - RAID levels.

- There are also two RAID levels which combine RAID levels 0 and 1 (striping and mirroring) in different combinations, designed to provide both performance and reliability at the expense of increased cost.
 - **RAID level 0 + 1** disks are first striped, and then the striped disks mirrored to another set. This level generally provides better performance than RAID level 5.
 - **RAID level 1 + 0** mirrors disks in pairs, and then stripes the mirrored pairs. The storage capacity, performance, etc. are all the same, but there is an advantage to this approach in the event of multiple disk failures, as illustrated below:
 - In diagram (a) below, the 8 disks have been divided into two sets of four, each of which is striped, and then one stripe set is used to mirror the other set.

- If a single disk fails, it wipes out the entire stripe set, but the system can keep on functioning using the remaining set.
- However if a second disk from the other stripe set now fails, then the entire system is lost, as a result of two disk failures.
- In diagram (b), the same 8 disks are divided into four sets of two, each of which is mirrored, and then the file system is striped across the four sets of mirrored disks.
 - If a single disk fails, then that mirror set is reduced to a single disk, but the system rolls on, and the other three mirror sets continue mirroring.
 - Now if a second disk fails, (that is not the mirror of the already failed disk), then another one of the mirror sets is reduced to a single disk, but the system can continue without data loss.
 - In fact the second arrangement could handle as many as four simultaneously failed disks, as long as no two of them were from the same mirror pair.

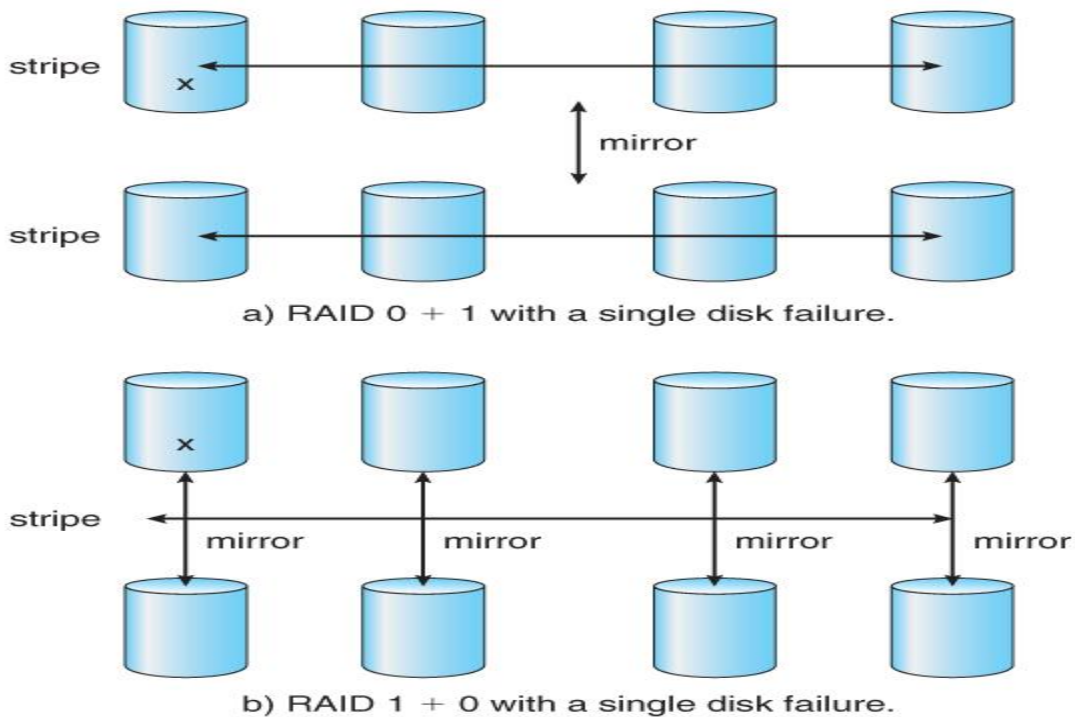


Figure 10.12 - RAID 0 + 1 and 1 + 0

10.7.4 Selecting a RAID Level

- Trade-offs in selecting the optimal RAID level for a particular application include cost, volume of data, need for reliability, need for performance, and rebuild time, the latter of which can affect the likelihood that a second disk will fail while the first failed disk is being rebuilt.
- Other decisions include how many disks are involved in a RAID set and how many disks to protect with a single parity bit. More disks in the set increases performance but increases cost. Protecting more disks per parity bit saves cost, but increases the likelihood that a second disk will fail before the first bad disk is repaired.

10.7.5 Extensions

- RAID concepts have been extended to tape drives (e.g. striping tapes for faster backups or parity checking tapes for reliability), and for broadcasting of data.

10.7.6 Problems with RAID

- RAID protects against physical errors, but not against any number of bugs or other errors that could write erroneous data.
- ZFS adds an extra level of protection by including data block checksums in all inodes along with the pointers to the data blocks. If data are mirrored and one copy has the correct checksum and the other does not, then the data with the bad checksum will be replaced with a copy of the data with the good checksum. This increases reliability greatly over RAID alone, at a cost of a performance hit that is acceptable because ZFS is so fast to begin with.

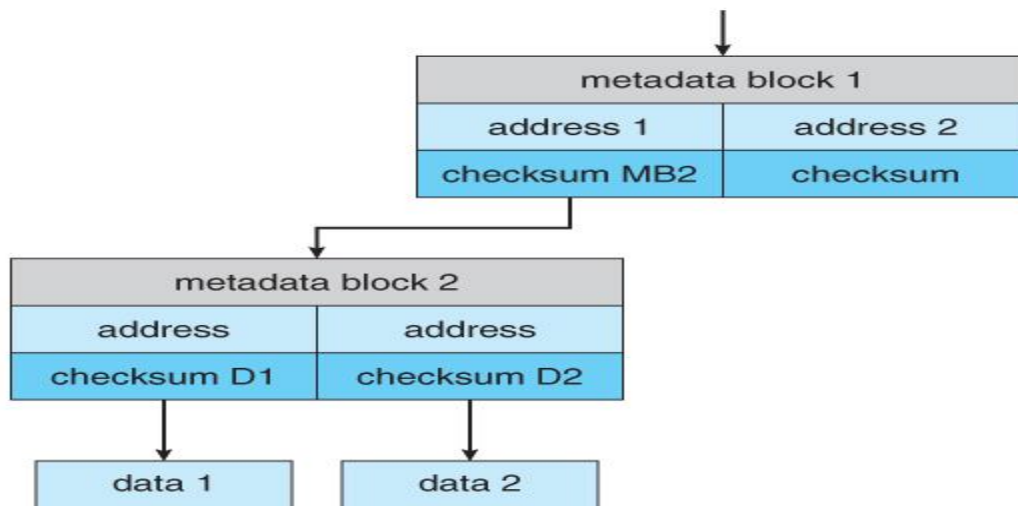


Figure 10.13 - ZFS checksums all metadata and data.

- Another problem with traditional filesystems is that the sizes are fixed, and relatively difficult to change. Where RAID sets are involved it becomes even harder to adjust filesystem sizes, because a filesystem cannot span across multiple filesystems.

- ZFS solves these problems by pooling RAID sets, and by dynamically allocating space to filesystems as needed. Filesystem sizes can be limited by quotas, and space can also be reserved to guarantee that a filesystem will be able to grow later, but these parameters can be changed at any time by the filesystem's owner. Otherwise filesystems grow and shrink dynamically as needed.

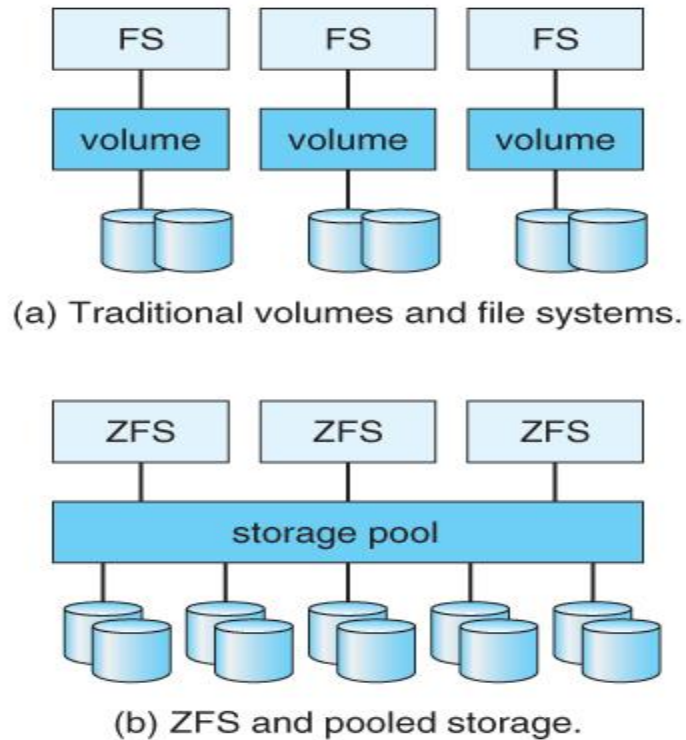


Figure 10.14 - (a) Traditional volumes and file systems. (b) a ZFS pool and file systems.

10.8 Stable-Storage Implementation (Optional)

- The concept of stable storage (first presented in chapter 6) involves a storage medium in which data is **never** lost, even in the face of equipment failure in the middle of a write operation.
- To implement this requires two (or more) copies of the data, with separate failure modes.
- An attempted disk write results in one of three possible outcomes:
 1. The data is successfully and completely written.
 2. The data is partially written, but not completely. The last block written may be garbled.
 3. No writing takes place at all.
- Whenever an equipment failure occurs during a write, the system must detect it, and return the system back to a consistent state. To do this requires two physical blocks for every logical block, and the following procedure:

1. Write the data to the first physical block.
 2. After step 1 had completed, then write the data to the second physical block.
 3. Declare the operation complete only after both physical writes have completed successfully.
- During recovery the pair of blocks is examined.
 - If both blocks are identical and there is no sign of damage, then no further action is necessary.
 - If one block contains a detectable error but the other does not, then the damaged block is replaced with the good copy. (This will either undo the operation or complete the operation, depending on which block is damaged and which is undamaged.)
 - If neither block shows damage but the data in the blocks differ, then replace the data in the first block with the data in the second block. (Undo the operation.)
 - Because the sequence of operations described above is slow, stable storage usually includes NVRAM as a cache, and declares a write operation complete once it has been written to the NVRAM.