

GLOBAL  
EDITION



# Modern Operating Systems

FOURTH EDITION

Andrew S. Tanenbaum • Herbert Bos

ALWAYS LEARNING

PEARSON

## ONLINE ACCESS

Thank you for purchasing a new copy of *Modern Operating Systems, Fourth Edition, Global Edition*. Your textbook includes twelve months of prepaid access to the book's Companion Website. This prepaid subscription provides you with full access to the following student support areas:

- Online Chapters
- Lab Experiments
- Online Exercises
- Simulation Exercises

Use a coin to scratch off the coating and reveal your student access code.  
Do not use a knife or other sharp object as it may damage the code.

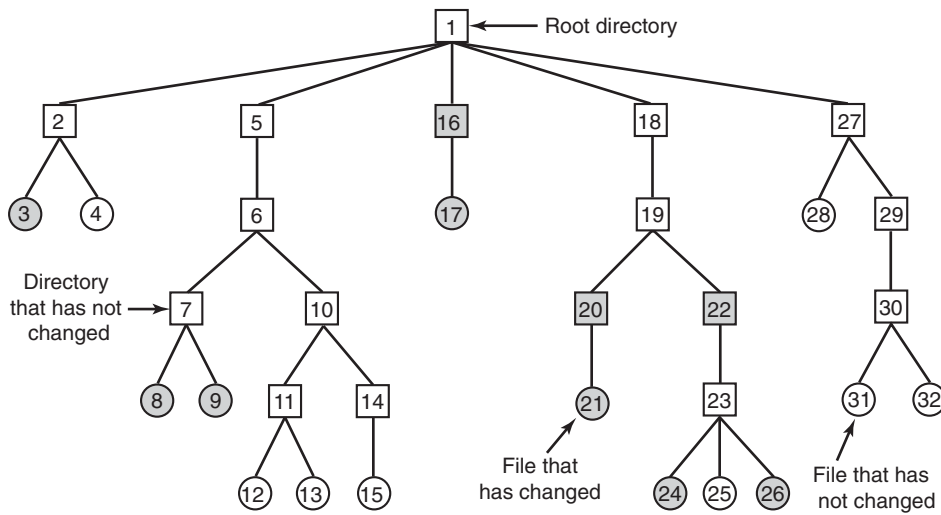
To access the *Modern Operating Systems, Fourth Edition, Global Edition*, Companion Website for the first time, you will need to register online using a computer with an Internet connection and a web browser. The process takes just a couple of minutes and only needs to be completed once.

1. Go to **[www.pearsonglobaleditions.com/Tanenbaum](http://www.pearsonglobaleditions.com/Tanenbaum)**
2. Click on your book.
3. Click on **Companion Website**.
4. Click on the **Register** button.
5. On the registration page, enter your student access code\* found beneath the scratch-off panel. Do not type the dashes. You can use lower- or uppercase.
6. Follow the on-screen instructions. If you need help at any time during the online registration process, simply click the **Need Help?** icon.
7. Once your personal Login Name and Password are confirmed, you can begin using the *Modern Operating Systems* Companion Website!

### To log in after you have registered:

You only need to register for this Companion Website once. After that, you can log in any time at **[www.pearsonglobaleditions.com/Tanenbaum](http://www.pearsonglobaleditions.com/Tanenbaum)** by providing your Login Name and Password when prompted.

\*Important: The access code can only be used once. This subscription is valid for twelve months upon activation and is not transferable. If this access code has already been revealed, it may no longer be valid. If this is the case, you can purchase a subscription by going to **[www.pearsonglobaleditions.com/Tanenbaum](http://www.pearsonglobaleditions.com/Tanenbaum)** and following the on-screen instructions.



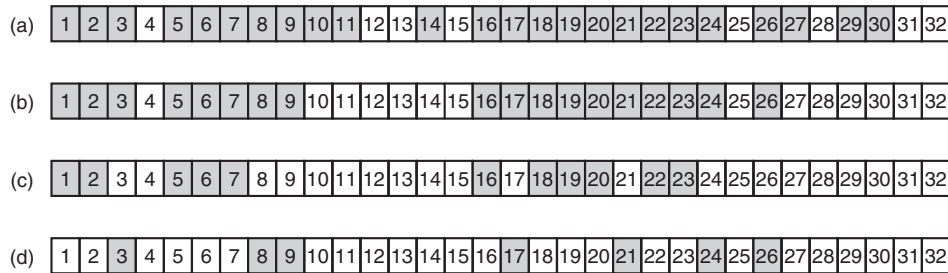
**Figure 4-25.** A file system to be dumped. The squares are directories and the circles are files. The shaded items have been modified since the last dump. Each directory and file is labeled by its i-node number.

restored first. To get their owners, modes, times, and whatever, correct, these directories must be present on the dump disk even though they themselves were not modified since the previous full dump.

The dump algorithm maintains a bitmap indexed by i-node number with several bits per i-node. Bits will be set and cleared in this map as the algorithm proceeds. The algorithm operates in four phases. Phase 1 begins at the starting directory (the root in this example) and examines all the entries in it. For each modified file, its i-node is marked in the bitmap. Each directory is also marked (whether or not it has been modified) and then recursively inspected.

At the end of phase 1, all modified files and all directories have been marked in the bitmap, as shown (by shading) in Fig. 4-26(a). Phase 2 conceptually recursively walks the tree again, unmarking any directories that have no modified files or directories in them or under them. This phase leaves the bitmap as shown in Fig. 4-26(b). Note that directories 10, 11, 14, 27, 29, and 30 are now unmarked because they contain nothing under them that has been modified. They will not be dumped. By way of contrast, directories 5 and 6 will be dumped even though they themselves have not been modified because they will be needed to restore today's changes to a fresh machine. For efficiency, phases 1 and 2 can be combined in one tree walk.

At this point it is known which directories and files must be dumped. These are the ones that are marked in Fig. 4-26(b). Phase 3 consists of scanning the i-nodes in numerical order and dumping all the directories that are marked for dumping.



**Figure 4-26.** Bitmaps used by the logical dumping algorithm.

These are shown in Fig. 4-26(c). Each directory is prefixed by the directory's attributes (owner, times, etc.) so that they can be restored. Finally, in phase 4, the files marked in Fig. 4-26(d) are also dumped, again prefixed by their attributes. This completes the dump.

Restoring a file system from the dump disk is straightforward. To start with, an empty file system is created on the disk. Then the most recent full dump is restored. Since the directories appear first on the dump disk, they are all restored first, giving a skeleton of the file system. Then the files themselves are restored. This process is then repeated with the first incremental dump made after the full dump, then the next one, and so on.

Although logical dumping is straightforward, there are a few tricky issues. For one, since the free block list is not a file, it is not dumped and hence it must be reconstructed from scratch after all the dumps have been restored. Doing so is always possible since the set of free blocks is just the complement of the set of blocks contained in all the files combined.

Another issue is links. If a file is linked to two or more directories, it is important that the file is restored only one time and that all the directories that are supposed to point to it do so.

Still another issue is the fact that UNIX files may contain holes. It is legal to open a file, write a few bytes, then seek to a distant file offset and write a few more bytes. The blocks in between are not part of the file and should not be dumped and must not be restored. Core files often have a hole of hundreds of megabytes between the data segment and the stack. If not handled properly, each restored core file will fill this area with zeros and thus be the same size as the virtual address space (e.g.,  $2^{32}$  bytes, or worse yet,  $2^{64}$  bytes).

Finally, special files, named pipes, and the like (anything that is not a real file) should never be dumped, no matter in which directory they may occur (they need not be confined to */dev*). For more information about file-system backups, see Chervenak et al., (1998) and Zwicky (1991).

### 4.4.3 File-System Consistency

Another area where reliability is an issue is file-system consistency. Many file systems read blocks, modify them, and write them out later. If the system crashes before all the modified blocks have been written out, the file system can be left in an inconsistent state. This problem is especially critical if some of the blocks that have not been written out are i-node blocks, directory blocks, or blocks containing the free list.

To deal with inconsistent file systems, most computers have a utility program that checks file-system consistency. For example, UNIX has *fsck*; Windows has *sfc* (and others). This utility can be run whenever the system is booted, especially after a crash. The description below tells how *fsck* works. *Sfc* is somewhat different because it works on a different file system, but the general principle of using the file system's inherent redundancy to repair it is still valid. All file-system checkers verify each file system (disk partition) independently of the other ones.

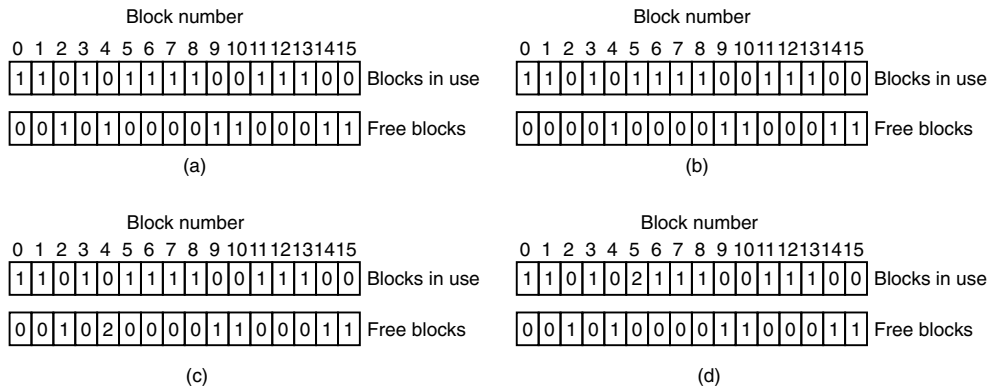
Two kinds of consistency checks can be made: blocks and files. To check for block consistency, the program builds two tables, each one containing a counter for each block, initially set to 0. The counters in the first table keep track of how many times each block is present in a file; the counters in the second table record how often each block is present in the free list (or the bitmap of free blocks).

The program then reads all the i-nodes using a raw device, which ignores the file structure and just returns all the disk blocks starting at 0. Starting from an i-node, it is possible to build a list of all the block numbers used in the corresponding file. As each block number is read, its counter in the first table is incremented. The program then examines the free list or bitmap to find all the blocks that are not in use. Each occurrence of a block in the free list results in its counter in the second table being incremented.

If the file system is consistent, each block will have a 1 either in the first table or in the second table, as illustrated in Fig. 4-27(a). However, as a result of a crash, the tables might look like Fig. 4-27(b), in which block 2 does not occur in either table. It will be reported as being a **missing block**. While missing blocks do no real harm, they waste space and thus reduce the capacity of the disk. The solution to missing blocks is straightforward: the file system checker just adds them to the free list.

Another situation that might occur is that of Fig. 4-27(c). Here we see a block, number 4, that occurs twice in the free list. (Duplicates can occur only if the free list is really a list; with a bitmap it is impossible.) The solution here is also simple: rebuild the free list.

The worst thing that can happen is that the same data block is present in two or more files, as shown in Fig. 4-27(d) with block 5. If either of these files is removed, block 5 will be put on the free list, leading to a situation in which the same block is both in use and free at the same time. If both files are removed, the block will be put onto the free list twice.



**Figure 4-27.** File-system states. (a) Consistent. (b) Missing block. (c) Duplicate block in free list. (d) Duplicate data block.

The appropriate action for the file-system checker to take is to allocate a free block, copy the contents of block 5 into it, and insert the copy into one of the files. In this way, the information content of the files is unchanged (although almost assuredly one is garbled), but the file-system structure is at least made consistent. The error should be reported, to allow the user to inspect the damage.

In addition to checking to see that each block is properly accounted for, the file-system checker also checks the directory system. It, too, uses a table of counters, but these are per file, rather than per block. It starts at the root directory and recursively descends the tree, inspecting each directory in the file system. For every i-node in every directory, it increments a counter for that file's usage count. Remember that due to hard links, a file may appear in two or more directories. Symbolic links do not count and do not cause the counter for the target file to be incremented.

When the checker is all done, it has a list, indexed by i-node number, telling how many directories contain each file. It then compares these numbers with the link counts stored in the i-nodes themselves. These counts start at 1 when a file is created and are incremented each time a (hard) link is made to the file. In a consistent file system, both counts will agree. However, two kinds of errors can occur: the link count in the i-node can be too high or it can be too low.

If the link count is higher than the number of directory entries, then even if all the files are removed from the directories, the count will still be nonzero and the i-node will not be removed. This error is not serious, but it wastes space on the disk with files that are not in any directory. It should be fixed by setting the link count in the i-node to the correct value.

The other error is potentially catastrophic. If two directory entries are linked to a file, but the i-node says that there is only one, when either directory entry is removed, the i-node count will go to zero. When an i-node count goes to zero, the

file system marks it as unused and releases all of its blocks. This action will result in one of the directories now pointing to an unused i-node, whose blocks may soon be assigned to other files. Again, the solution is just to force the link count in the i-node to the actual number of directory entries.

These two operations, checking blocks and checking directories, are often integrated for efficiency reasons (i.e., only one pass over the i-nodes is required). Other checks are also possible. For example, directories have a definite format, with i-node numbers and ASCII names. If an i-node number is larger than the number of i-nodes on the disk, the directory has been damaged.

Furthermore, each i-node has a mode, some of which are legal but strange, such as 0007, which allows the owner and his group no access at all, but allows outsiders to read, write, and execute the file. It might be useful to at least report files that give outsiders more rights than the owner. Directories with more than, say, 1000 entries are also suspicious. Files located in user directories, but which are owned by the superuser and have the SETUID bit on, are potential security problems because such files acquire the powers of the superuser when executed by any user. With a little effort, one can put together a fairly long list of technically legal but still peculiar situations that might be worth reporting.

The previous paragraphs have discussed the problem of protecting the user against crashes. Some file systems also worry about protecting the user against himself. If the user intends to type

```
rm *.o
```

to remove all the files ending with *.o* (compiler-generated object files), but accidentally types

```
rm * .o
```

(note the space after the asterisk), *rm* will remove all the files in the current directory and then complain that it cannot find *.o*. In Windows, files that are removed are placed in the recycle bin (a special directory), from which they can later be retrieved if need be. Of course, no storage is reclaimed until they are actually deleted from this directory.

#### 4.4.4 File-System Performance

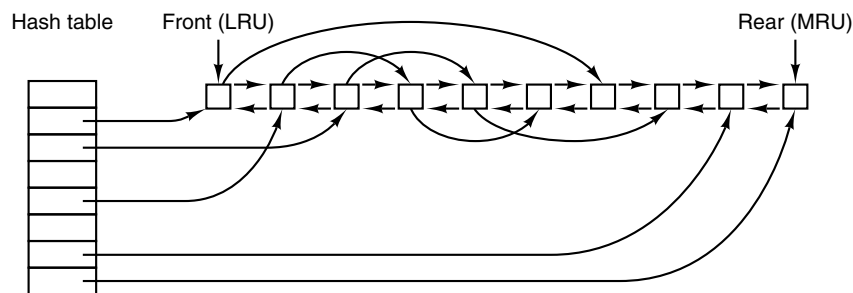
Access to disk is much slower than access to memory. Reading a 32-bit memory word might take 10 nsec. Reading from a hard disk might proceed at 100 MB/sec, which is four times slower per 32-bit word, but to this must be added 5–10 msec to seek to the track and then wait for the desired sector to arrive under the read head. If only a single word is needed, the memory access is on the order of a million times as fast as disk access. As a result of this difference in access time, many file systems have been designed with various optimizations to improve performance. In this section we will cover three of them.

## Caching

The most common technique used to reduce disk accesses is the **block cache** or **buffer cache**. (Cache is pronounced “cash” and is derived from the French *cache*, meaning to hide.) In this context, a cache is a collection of blocks that logically belong on the disk but are being kept in memory for performance reasons.

Various algorithms can be used to manage the cache, but a common one is to check all read requests to see if the needed block is in the cache. If it is, the read request can be satisfied without a disk access. If the block is not in the cache, it is first read into the cache and then copied to wherever it is needed. Subsequent requests for the same block can be satisfied from the cache.

Operation of the cache is illustrated in Fig. 4-28. Since there are many (often thousands of) blocks in the cache, some way is needed to determine quickly if a given block is present. The usual way is to hash the device and disk address and look up the result in a hash table. All the blocks with the same hash value are chained together on a linked list so that the collision chain can be followed.



**Figure 4-28.** The buffer cache data structures.

When a block has to be loaded into a full cache, some block has to be removed (and rewritten to the disk if it has been modified since being brought in). This situation is very much like paging, and all the usual page-replacement algorithms described in Chap. 3, such as FIFO, second chance, and LRU, are applicable. One pleasant difference between paging and caching is that cache references are relatively infrequent, so that it is feasible to keep all the blocks in exact LRU order with linked lists.

In Fig. 4-28, we see that in addition to the collision chains starting at the hash table, there is also a bidirectional list running through all the blocks in the order of usage, with the least recently used block on the front of this list and the most recently used block at the end. When a block is referenced, it can be removed from its position on the bidirectional list and put at the end. In this way, exact LRU order can be maintained.

Unfortunately, there is a catch. Now that we have a situation in which exact LRU is possible, it turns out that LRU is undesirable. The problem has to do with