

Operating Systems 2230

Computer Science & Software Engineering

Lecture 10: File System Management

A clear and obvious requirement of an operating system is the provision of a convenient, efficient, and robust filing system.

The file system is used not only to store users' programs and data, but also to support and represent significant portions of the operating system itself.

Stallings introduces the traditional file system concepts of:

fields represent the smallest logical item of data “understood” by a file-system: examples including a single integer or string. Fields may be of a fixed length (recorded by the file-system as “part of” each file), or be of variable length, where either the length is stored with the field or a sentinel byte signifies the extent.

records consist of one or more fields and are treated as a single unit by application programs needing to read or write the record. Again, records may be of fixed or variable length.

files are familiar collections of identically represented records, and are accessed by unique names. Deleting a file (by name) similarly affects its component records. Access control restricts the forms of operations that may be performed on files by various classes of users and programs.

databases consist of one or more files whose contents are strongly (logically) related. The on-disk representation of the data is considered optimal with respect to its datatypes and its expected methods of access.

Today, nearly all modern operating systems provide relatively simple file systems consisting of byte-addressable, sequential files.

The basic unit of access, the record from Stallings's taxonomy, is the byte, and access to these is simply keyed using the byte's numeric offset from the beginning of each file (the size of the variable holding this offset will clearly limit a file's extent).

The file-systems of modern operating systems, such as Linux's *ext2* and *ext3* systems and the Windows-NT File System (NTFS), provide file systems up to 4TB (terabytes, 2^{40} bytes) in size, with each file being up to 16GB.

Operating Systems and Databases

More complex arrangements of data and its access modes, such as using search keys to locate individual fixed-length records, have been "relegated" to being supported by run-time libraries and database packages (see Linux's *gdbm* information).

In commercial systems, such as for a large eCommerce system, a database management system (DBMS) will store and provide access to database information independent of an operating system's representation of files.

The DBMS manages a whole (physical) disk-drive (or a dedicated partition) and effectively assumes much of the operating system's role in managing access, security, and backup facilities. Examples include products from Oracle and Ingres.

The File Management System

The increased simplicity of the file systems provided by modern operating systems has enabled a concentration on efficiency, security and constrained access, and robustness.

Operating systems provide a layer of system-level software, using system-calls to provide services relating to the provision of files. This obviates the need for each application program to manage its own disk allocation and access.

As a kernel-level responsibility, the file management system again provides a central role in resource (buffer) allocation and process scheduling.

From the “viewpoint” of the operating system kernel itself, the file management system has a number of goals:

- to support the storage, searching, and modification of user data,
- to guarantee the correctness and integrity of the data,
- to optimise both the overall throughput (from the operating system’s global view) and response time (from the user’s view).
- to provide “transparent” access to many different device types such as hard disks, CD-ROMs, and floppy disks,
- to provide a standardised set of I/O interface routines, perhaps ameliorating the concepts of file, device and network access.

User Requirements

A further critical requirement in a multi-tasking, multi-user system is that the file management system meet users' requirements in the presence of different users.

Subject to appropriate permissions, important requirements include:

- the recording of a primary owner of each file (for access controls and accounting),
- each file should be accessed by (at least) one symbolic name,
- each user should be able to create, delete, read, and modify files,
- users should have constrained access to the files of others,
- a file's owner should be able to modify the types of access that may be performed by others,
- a facility to copy/backup files to identical or different forms of media,
- a secure logging and notification system if inappropriate access is attempted (or achieved).

Components of the File Management System

Stallings provides a generic depiction of the structure of the file management system (Figure 12.1: all figures are taken from Stallings' web-site):

Its components, from the bottom up, include:

- The *device drivers* communicate directly with I/O hardware. Device drivers commence I/O requests and receive asynchronous notification of their completion (via DMA).

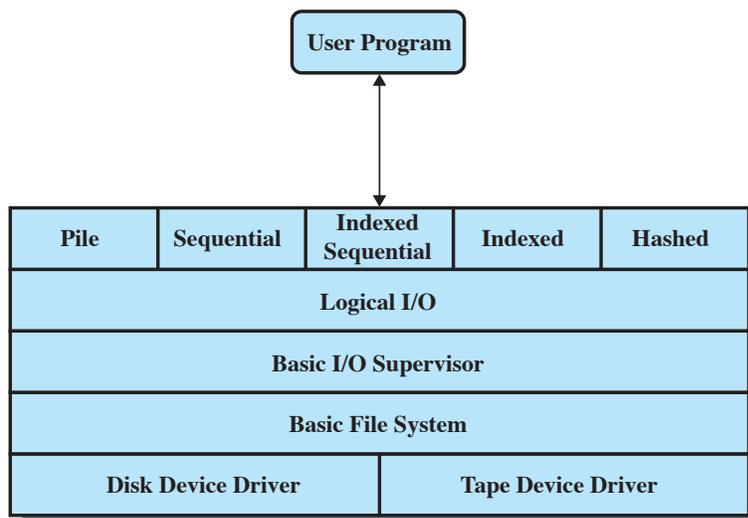


Figure 12.1 File System Software Architecture

- The *basic file system* exchanges fixed-sized pieces (blocks) of data with the device drivers, buffers those blocks in main memory, and maps blocks to their physical location.
- The *I/O supervisor* manages the choice of device, its scheduling and status, and allocates I/O buffers to processes.
- The *logical I/O* layer maps I/O requests to blocks, and maintains ownership and access information about files.

The Role of Directory Structures

All modern operating systems have adopted the hierarchical directory model to represent collections of files.

A directory itself is typically a special type of file storing information about the files (and hence directories) it contains.

Although a directory is “owned” by a user, the directory is truly owned by the operating system itself. The operating system must constrain access to important, and often hidden, information in the directory itself.

Although the “owner” of a directory may examine and (attempt to) modify it, true modification is only permitted through operating system primitives affecting the internal directory structure.

For example, deleting a file involves both deallocating any disk blocks used by that file, and removing that file’s information from its container directory.

If direct modification of the directory were possible, the file may become “unlinked” in the hierarchical structure, and thereafter be inaccessible (as it could not be named and found by name).

In general, each directory is stored as a simple file, containing the names of files within it. The directory’s “contents” may be read with simple user-level routines.

Reading Directory Information

Under both Unix and Windows-NT, user-level library routines simplify the reading of a directory’s contents. Libraries use the standard *open*, *read*, and *close* calls to examine the contents of a directory.

Under Unix:

```
#include <dirent.h>

DIR                *dirp;
struct dirent      *dp;

dirp = opendir(".");
for (dp = readdir(dirp); dp != NULL; dp = readdir(dirp))
    printf("%s\n", dp->d_name);
closedir (dirp);
```

Under Windows-NT:

```
#include <windows.h>

HANDLE             filehandle;
WIN32_FIND_DATA   finddata;

filehandle = FindFirstFile("*.*", &finddata);
if (filehandle != INVALID_HANDLE_VALUE) {
    printf("%s\n", finddata->cFileName);
    while (FindNextFile(filehandle, &finddata))
        printf("%s\n", finddata->cFileName);
}
FindClose(filehandle);
```

Note that the files will not be listed in alphabetical order; any sorting is provided by programs such as *ls* or *dir*.

File Information Structures

Early operating systems maintained information about each file in the directory entries themselves.

However, the use of the hierarchical directory model encourages a separate information structure for each file.

Each information structure is accessed by a unique key (unique within the context of its I/O device), and the role of the directory is now simply to provide file names.

As an example, Unix refers to these information structures as *inodes*, accessing them by their integral inode number on each device (themselves referred to by two integral device numbers). A distinct file is identified by its major and minor device numbers and its inode number.

Typical information contained within each information structure includes:

- the file's type (plain file, directory, etc),
- the file's size, in bytes and/or blocks,
- any limits set on the file's size,
- the primary owner of the file,
- information about other potential users of this file,
- access constraints on the owner and other users,
- dates and times of creation, last access and last modification,
- dates and times of last backup and recovery, and
- asynchronous event triggers when the file changes.

As a directory consequence, multiple file names (even from different directories) may refer to the same file. These are termed file-system links.

Reading a File's Information Structure

Under both Unix and Windows-NT, library routines again simplify access to a file's information structure:

Under Unix:

```
#include <sys/stat.h>

struct stat statbuf;

stat("filename", &statbuf);
printf("inode = %d\n", (int)statbuf.st_ino);
printf("owner's userid = %d\n", (int)statbuf.st_uid);
printf("size in bytes = %d\n", (int)statbuf.st_size);
printf("size in blocks = %d\n", (int)statbuf.st_blocks);
```

Under Windows-NT:

```
#include <windows.h>

HANDLE                filehandle;
DWORD                sizeLo, sizeHi;
BY_HANDLE_FILE_INFORMATION info;

filehandle = CreateFile("filename", GENERIC_READ,
                       FILE_SHARE_WRITE,
                       0, OPEN_EXISTING, 0, 0);
GetFileInformationByHandle(filehandle, &info);
sizeLo      = GetFileSize(filehandle, &sizeHi);
printf("number of links = %d\n", (int)info.nNumberOfLinks);
printf("size in bytes ( low 32bits) = %u\n", sizeLo);
printf("size in bytes (high 32bits) = %u\n", sizeHi);
CloseFile(filehandle);
```

File Allocation Methods — Contiguous

Of particular interest is the method that the basic file system uses to place the logical file blocks onto the physical medium (for example, into the disk's tracks and sectors).

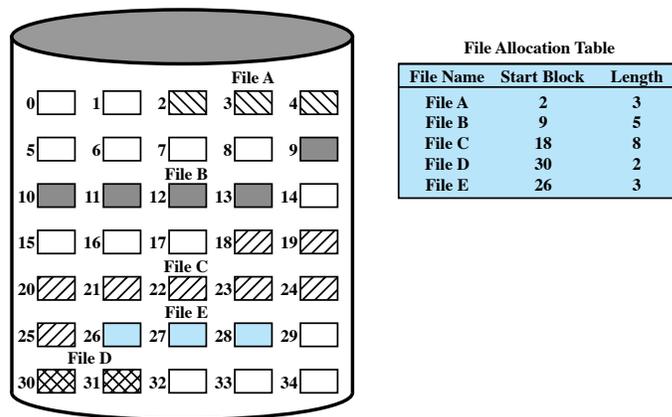


Figure 12.7 Contiguous File Allocation

The simplest policy (akin to memory partitioning in process management) is the use of a fixed or contiguous allocation. This requires that a file's maximum (ever) size be determined when the file is created, and that the file cannot grow beyond this limit (Figure 12.7).

The *file allocation table* stores, for each file, its starting block and length.

Like simple memory allocation schemes, this method suffers from both internal fragmentation (if the initial allocation is too large) and external fragmentation (as files are deleted over time).

Again, as with memory allocation, a compaction scheme is required to reduce fragmentation (“defragging” your disk).

File Allocation Methods — Chained

The opposite extreme to contiguous allocation is chained allocation.

Here, the blocks allocated to a file form a linked list (or chain), and as a file's length is extended (by appending to the file), a new block is allocated and linked to the last block in the file (Figure 12.9).

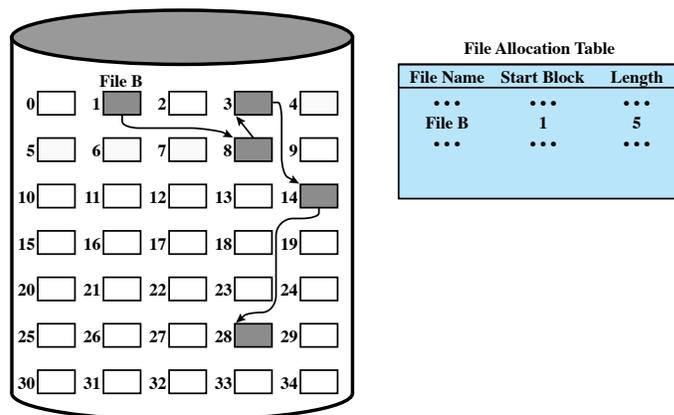


Figure 12.9 Chained Allocation

A small “pointer” of typically 32 or 64 bits is allocated within each file block to indicate the next block in the chain. Thus seeking within a file requires a read of each block to follow the pointers.

New blocks may be allocated from any free block on the disk. In particular, a file's blocks need no longer be contiguous.

File Allocation Methods — Indexed

The file allocation method of choice in both Unix and Windows is the indexed allocation method. This method was championed by the Multics operating system in 1966. The file-allocation table contains a multi-level index for each

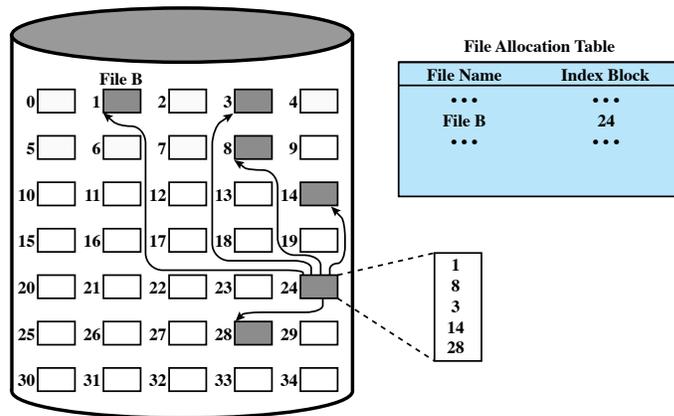


Figure 12.11 Indexed Allocation with Block Portions

file. Indirection blocks are introduced each time the total number of blocks “overflows” the previous index allocation. Typically, the indices are neither stored with the file-allocation table nor with the file, and are retained in memory when the file is opened (Figure 12.11).

Further Reading on File Systems

Analysis of the Ext2fs structure, by Louis-Dominique Dubeau, at www.csse.uwa.edu.au/teaching/CITS2230/ext2fs/.

The FAT — File Allocation Table, by O’Reilly & Associates, at www.csse.uwa.edu.au/teaching/CITS2230/fat/.

Redundant Arrays of Inexpensive Disks (RAID)

In 1987, UC Berkeley researchers Patterson, Gibson, and Katz warned the world of an impending predicament. The speed of computer CPUs and performance of RAM were growing exponentially, but mechanical disk drives were improving only incrementally.

RAID is the organisation of multiple disks into a large, high performance logical disk.

Disk arrays *stripe* data across multiple disks and access them in parallel to achieve:

- Higher data transfer rates on large data accesses, and
- Higher I/O rates on small data accesses.

Data striping also results in uniform load balancing across all of the disks, eliminating hot spots that otherwise saturate a small number of disks, while the majority of disks sit idle.

However, large disk arrays are highly vulnerable to disk failures:

- A disk array with a hundred disks is a hundred times more likely to fail than a single disk.
- An MTTF (mean-time-to-failure) of 500,000 hours for a single disk implies an MTTF of $500,000/100$, i.e. 5000 hours for a disk array with a hundred disks.

The solution to the problem of lower *reliability* in disk arrays is to improve the *availability* of the system, by employing redundancy in the form of error-correcting codes to tolerate disk failures. Do not confuse reliability and availability:

Reliability is how well a system can work without any failures in its components. If there is a failure, the system was not reliable.

Availability is how well a system can work in times of a failure. If a system is able to work even in the presence of a failure of one or more system components, the system is said to be available.

Redundancy improves the availability of a system, but cannot improve the reliability.

Reliability can only be increased by improving manufacturing technologies, or by using fewer individual components in a system.

Redundancy has costs

Every time there is a write operation, there is a change of data. This change also has to be reflected in the disks storing redundant information. This worsens the performance of writes in redundant disk arrays significantly, compared to the performance of writes in non-redundant disk arrays.

Moreover, keeping the redundant information consistent in the presence of concurrent I/O operation and the possibility of system crashes can be difficult.

The Need for RAID

The need for RAID can be summarised as:

- An array of multiple disks accessed in parallel will give greater throughput than a single disk, and
- Redundant data on multiple disks provides fault tolerance.

Provided that the RAID hardware and software perform true parallel accesses on multiple drives, there will be a performance improvement over a single disk.

With a single hard disk, you cannot protect yourself against the costs of a disk failure, the time required to obtain and install a replacement disk, reinstall the operating system, restore files from backup tapes, and repeat all the data entry performed since the last backup was made.

With multiple disks and a suitable redundancy scheme, your system can stay up and running when a disk fails, and even while the replacement disk is being installed and its data restored.

We aim to meet the following goals:

- maximise the number of disks being accessed in parallel,
- minimise the amount of disk space being used for redundant data, and
- minimise the overhead required to achieve the above goals.

Data Striping

Data striping, for improved performance, transparently distributes data over multiple disks to make them appear as a single fast, large disk. Striping improves aggregate I/O performance by allowing multiple I/O requests to be serviced in parallel.

- Multiple independent requests can be serviced in parallel by separate disks. This decreases the queuing time seen by I/O requests.
- Single multi-block requests can be serviced by multiple disks acting in coordination. This increases the effective transfer rate seen by a single request. The performance benefits increase with the number of disks in the array, but the reliability of the whole array is lowered.

Most RAID designs depend on two features:

- the granularity of data interleaving, and

- the way in which the redundant data is computed and stored across the disk array.

Fine-grained arrays interleave data in relatively small units so that all I/O requests, regardless of their size, access all of the disks in the disk array. This results in very high data transfer rate for all I/O requests but has the disadvantages that only one logical I/O request can be serviced at once, and that all disks must waste time positioning for every request.

Coarse-grained disk arrays interleave data in larger units so that small I/O requests need access only a small number of disks while large requests can access all the disks in the disk array. This allows multiple small requests to be serviced simultaneously while still allowing large requests to see the higher transfer rates from using multiple disks.

Redundancy

Since a larger number of disks lowers the overall reliability of the array of disks, it is important to incorporate redundancy in the array of disks to tolerate disk failures and allow for the continuous operation of the system without any loss of data.

Redundancy brings two problems:

- selecting the method for computing the redundant information. Most redundant disks arrays today use parity or Hamming encoding, and
- selecting a method for distribution of the redundant information across the disk array, either concentrating redundant information on a small number of disks, or distributing redundant information uniformly across all of the disks.

Such schemes are generally more desirable because they avoid hot spots and other load balancing problems suffered by schemes that do not uniformly distribute redundant information.

Non-Redundant (RAID Level 0)

A non-redundant disk array, or RAID level 0, has the lowest cost of any RAID organisation because it has no redundancy.

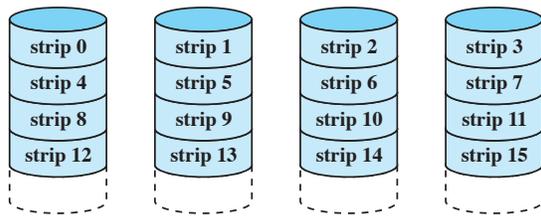
Surprisingly, it does not have the best performance. Redundancy schemes that duplicate data, such as mirroring, can perform better on reads by selectively scheduling requests on the disk with the shortest expected seek and rotational delays.

Sequential blocks of data are written across multiple disks in stripes, as shown in Figure 11.8.

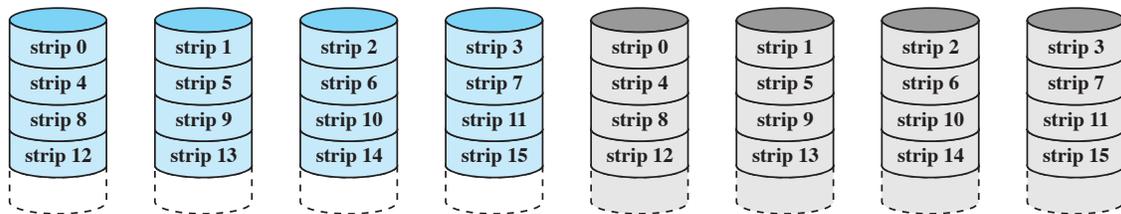
Without redundancy, any single disk failure will result in data-loss. Non-redundant disk arrays are widely used in super-computing environments where performance and capacity, rather than reliability, are the primary concerns.

The size of a data block, the *stripe width*, varies with the implementation, but is always at least as large as a disk's sector size. When reading back this sequential data, all disks can be read in parallel.

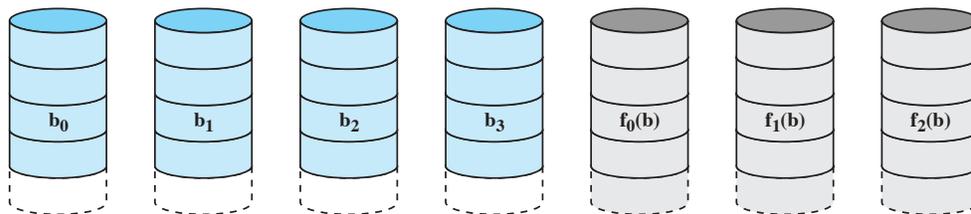
In a multi-tasking operating system, there is a high probability that even non-sequential disk accesses will keep all of the disks working in parallel.



(a) RAID 0 (non-redundant)



(b) RAID 1 (mirrored)



(c) RAID 2 (redundancy through Hamming code)

Figure 11.8 RAID Levels (page 1 of 2)

Mirrored (RAID Level 1)

The traditional solution, called *mirroring* or *shadowing*, uses twice as many disks as a non-redundant disk array.

- Whenever data is written to a disk, the same data is also written to a redundant disk in parallel, so that there are always two copies of the information.
- When data is read from a disk, it is retrieved from the disk with the shorter queuing, seek and rotational delays.
- Recovery from a disk failure is simple: the data may still be accessed from the second drive.

Mirroring is frequently used in database applications where availability and transaction time are more important than storage efficiency.

RAID-1 can achieve high I/O request rates if most requests are reads, which can be satisfied from either drive: up to twice the rate of RAID-0.

Parallel Access (RAID Level 2)

Memory systems have provided recovery from failed components with much less cost than mirroring by using *Hamming codes*. Hamming codes contain parity for distinct overlapping subsets of components. In one version of this scheme, four disks require three redundant disks, one less than mirroring.

- On a single read request, all disks are accessed simultaneously. The requested data and the error-correcting code are presented to the disk-controller, which can determine if there was an error, and correct it automatically (and hopefully report a problem with one of the disks).
- On a single write, all data disks and parity disks must be accessed.

Since the number of redundant disks is proportional to the \log_2 of the total number of the disks on the system, storage efficiency increases as the number of data disks increases.

With 32 data disks, a RAID 2 system would require 7 additional disks for a Hamming-code ECC. Not considered very economical.

Bit-Interleaved Parity (RAID Level 3)

Unlike memory component failures, disk controllers can easily identify which disk has failed. Thus, one can use a single parity disk rather than a set of parity disks to recover lost information.

In a bit-interleaved, parity disk array, data is conceptually interleaved bit-wise over the data disks, and a single parity disk is added to tolerate any single disk failure. Each read request accesses all data disks and each write request accesses all data disks and the parity disk.

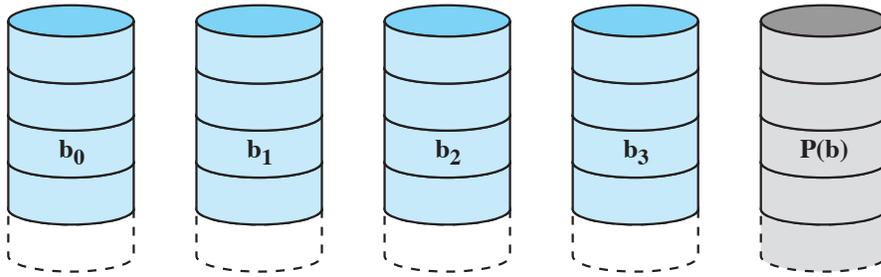
In the event of a disk failure, data reconstruction is simple. Consider an array of five drives, four data drives $D0$, $D1$, $D2$, and $D3$, and a parity drive $D4$. The parity for bit k is calculated with:

$$D4(k) = D0(k) \text{ xor } D1(k) \text{ xor } D2(k) \text{ xor } D3(k)$$

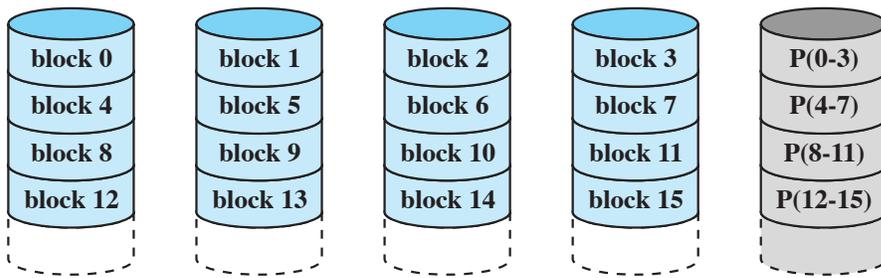
Suppose that $D1$ fails. We can add $D4(k) \text{ xor } D1(k)$ to both sides, giving:

$$D1(k) = D4(k) \text{ xor } D3(k) \text{ xor } D2(k) \text{ xor } D0(k)$$

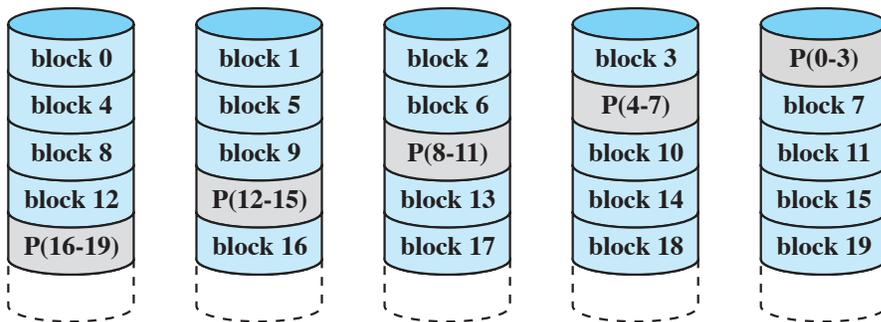
The parity disk contains only parity and no data, and cannot participate in reads, resulting in slightly lower read performance than for redundancy schemes that distribute the parity and data over all disks. Bit-interleaved, parity disk arrays are frequently used in applications that require high bandwidth but not high I/O rates.



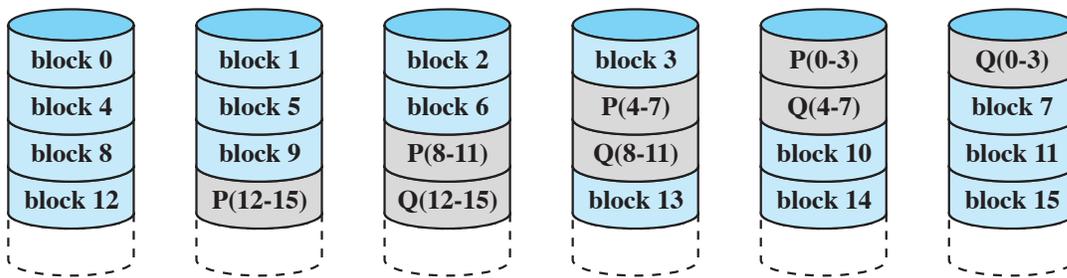
(d) RAID 3 (bit-interleaved parity)



(e) RAID 4 (block-level parity)



(f) RAID 5 (block-level distributed parity)



(g) RAID 6 (dual redundancy)

Figure 11.8 RAID Levels (page 2 of 2)

Block-Interleaved Distributed-Parity (RAID Level 5)

RAID-5 eliminates the parity disk bottleneck by distributing the parity uniformly over all of the disks. Similarly, it distributes data over all of the disks rather than over all but one.

This allows all disks to participate in servicing read operations. RAID-5 has the best small read, large write, performance of any scheme. Small write requests are somewhat inefficient compared with mirroring, due to the read-modify-write operations to update parity. This is the major performance weakness of RAID-5.

P+Q redundancy (RAID Level 6)

(Finally) RAID-6 performs two different parity calculations, stored in separate blocks on separate disks.

Because the parity calculations are different, data can be fully recovered in the event of two disk failures.