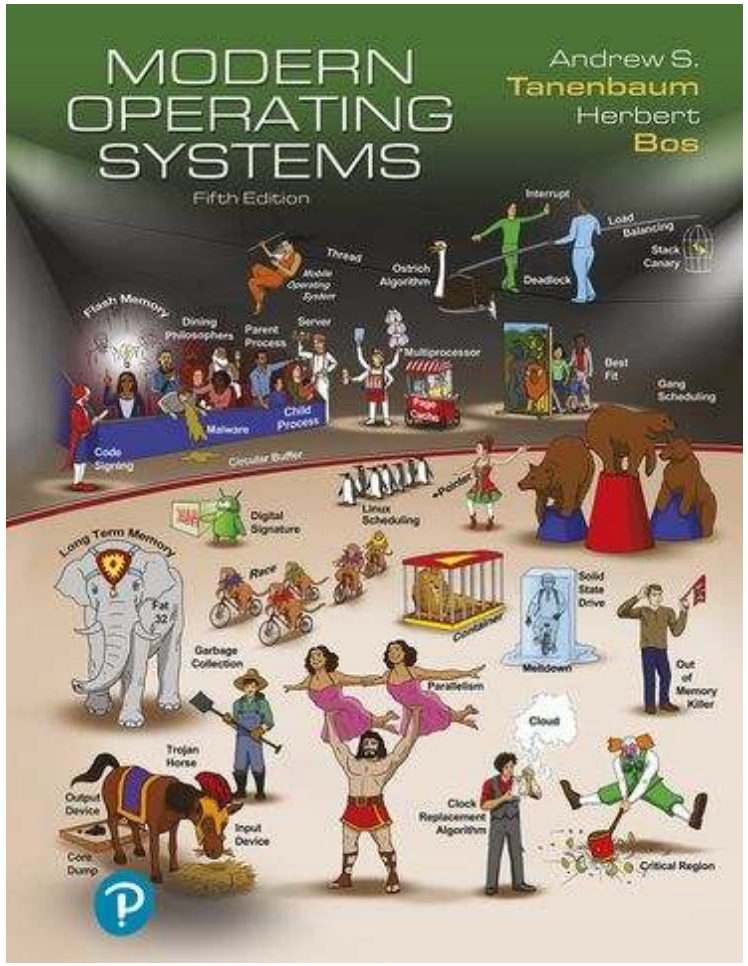
Modern Operating Systems

Fifth Edition



Chapter 4

File Systems

File Systems (1 of 4)

Essential requirements for long-term information storage:

1. It must be possible to store a very large amount of information.
2. Information must survive termination of process using it.
3. Multiple processes must be able to access information concurrently.

File Systems (2 of 4)

Think of a disk as a linear sequence of fixed-size blocks and supporting two operations:

1. Read block **k**.
2. Write block **k**

File Systems (3 of 4)

Questions that quickly arise:

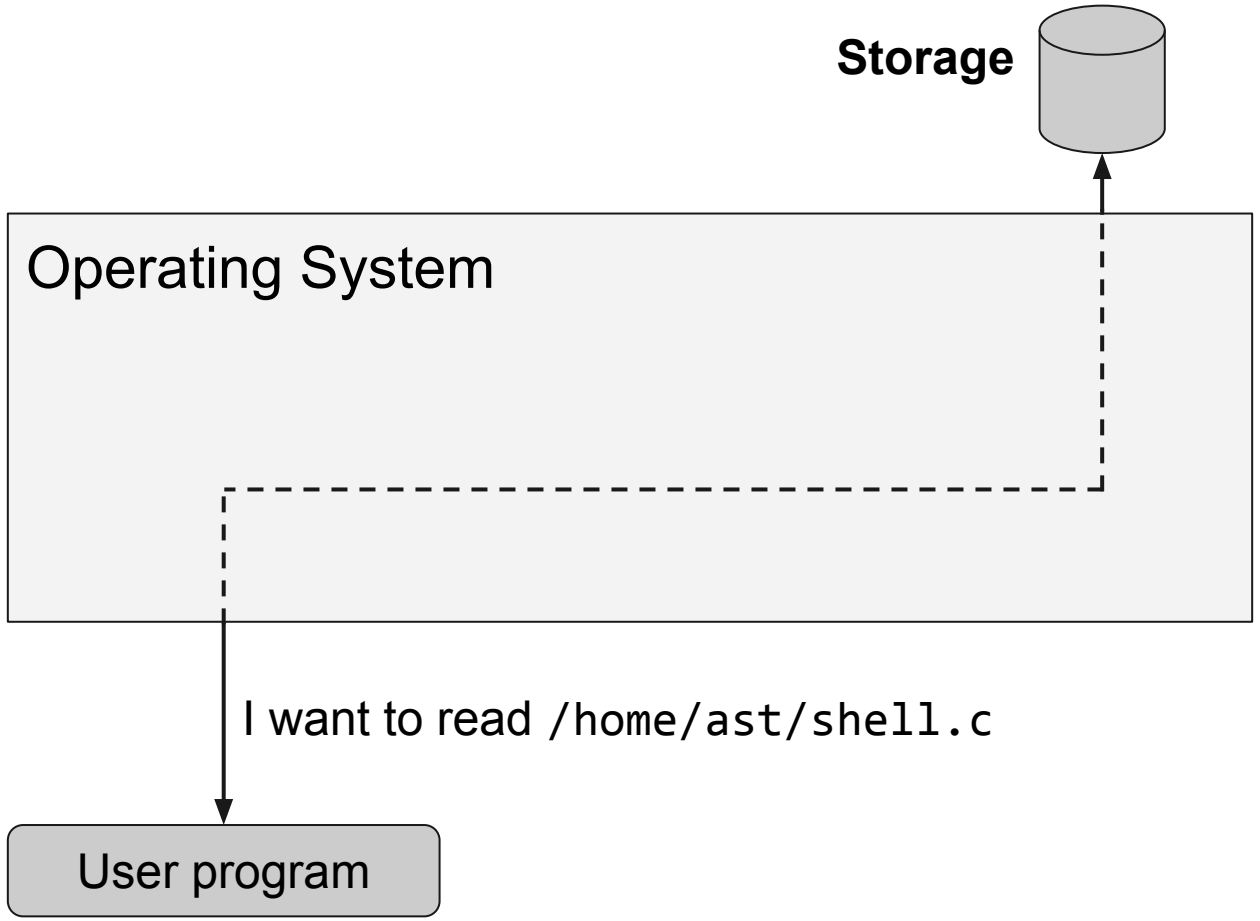
1. How do you find information?
2. How do you keep one user from reading another user's data?
3. How do you know which blocks are free?

File Systems (4 of 4)

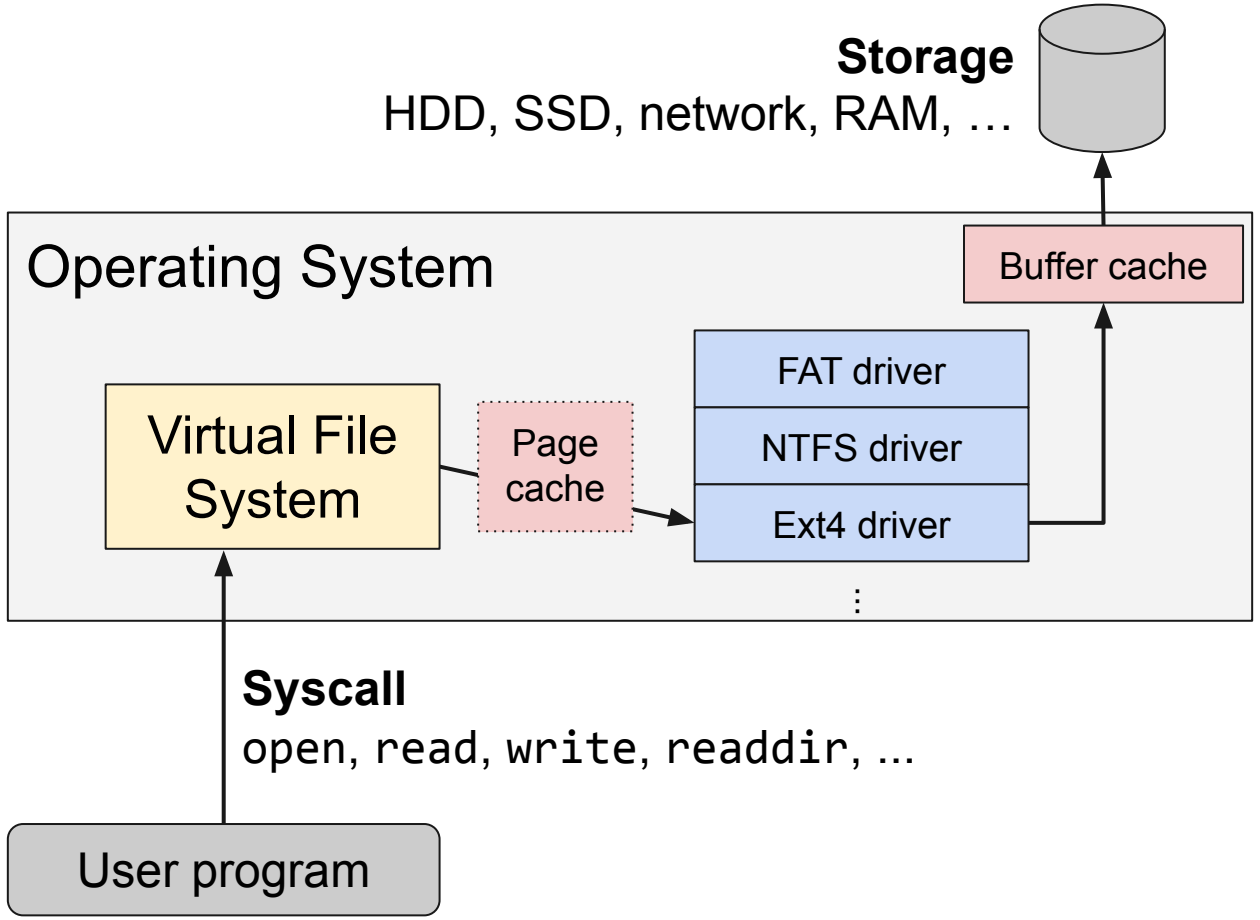
File Systems are:

- A way to organize and (persistently) store information
- An abstraction over storage devices:
  - Hard disk, SSD, network, RAM, …
- Organized in files and (typically) directories.
- Examples:
  - FAT12/FAT16: MS-DOS
  - NTFS: Windows
  - Ext4: Linux
  - APFS: macOS/iOS

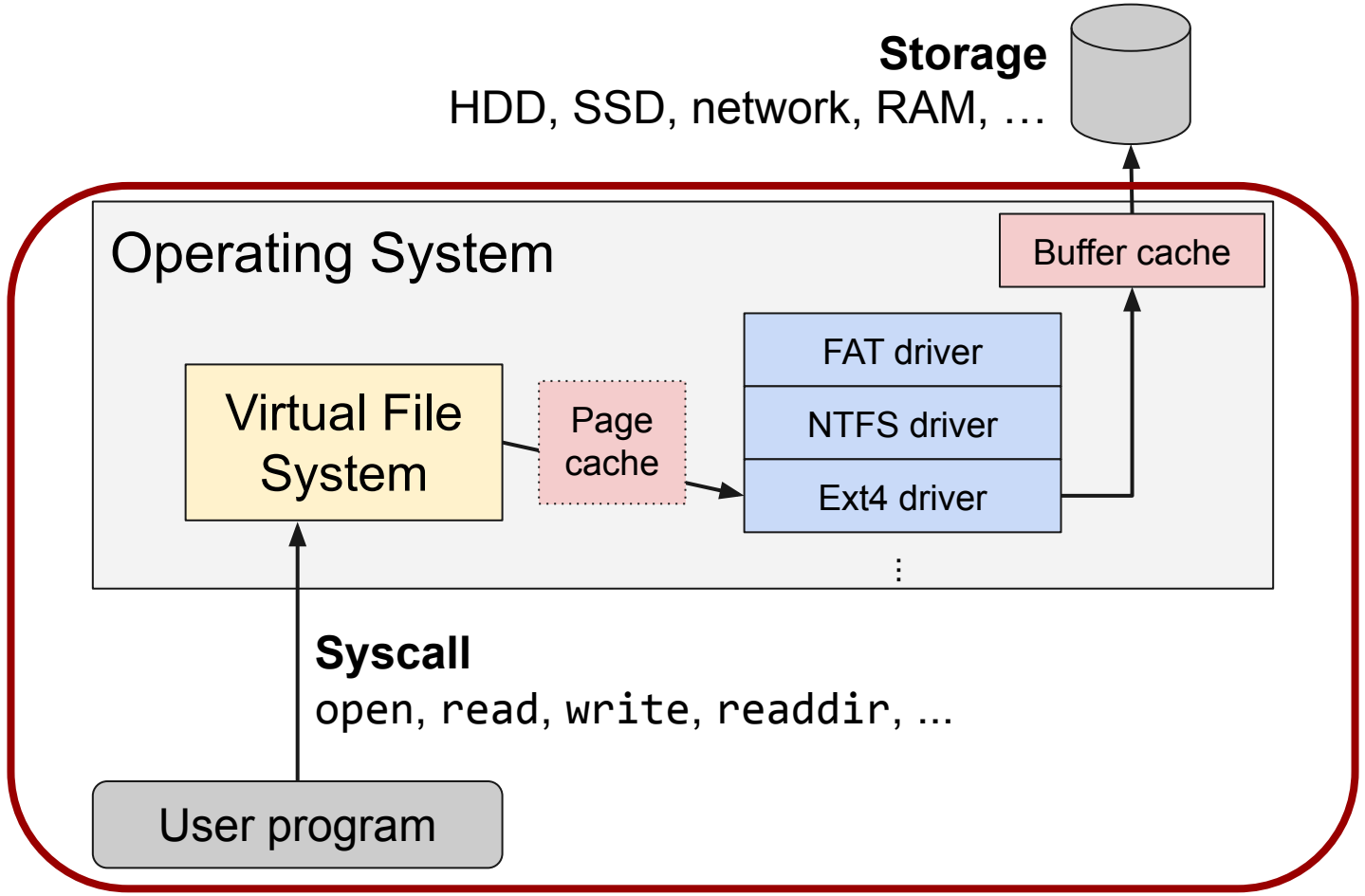
Overview



Overview

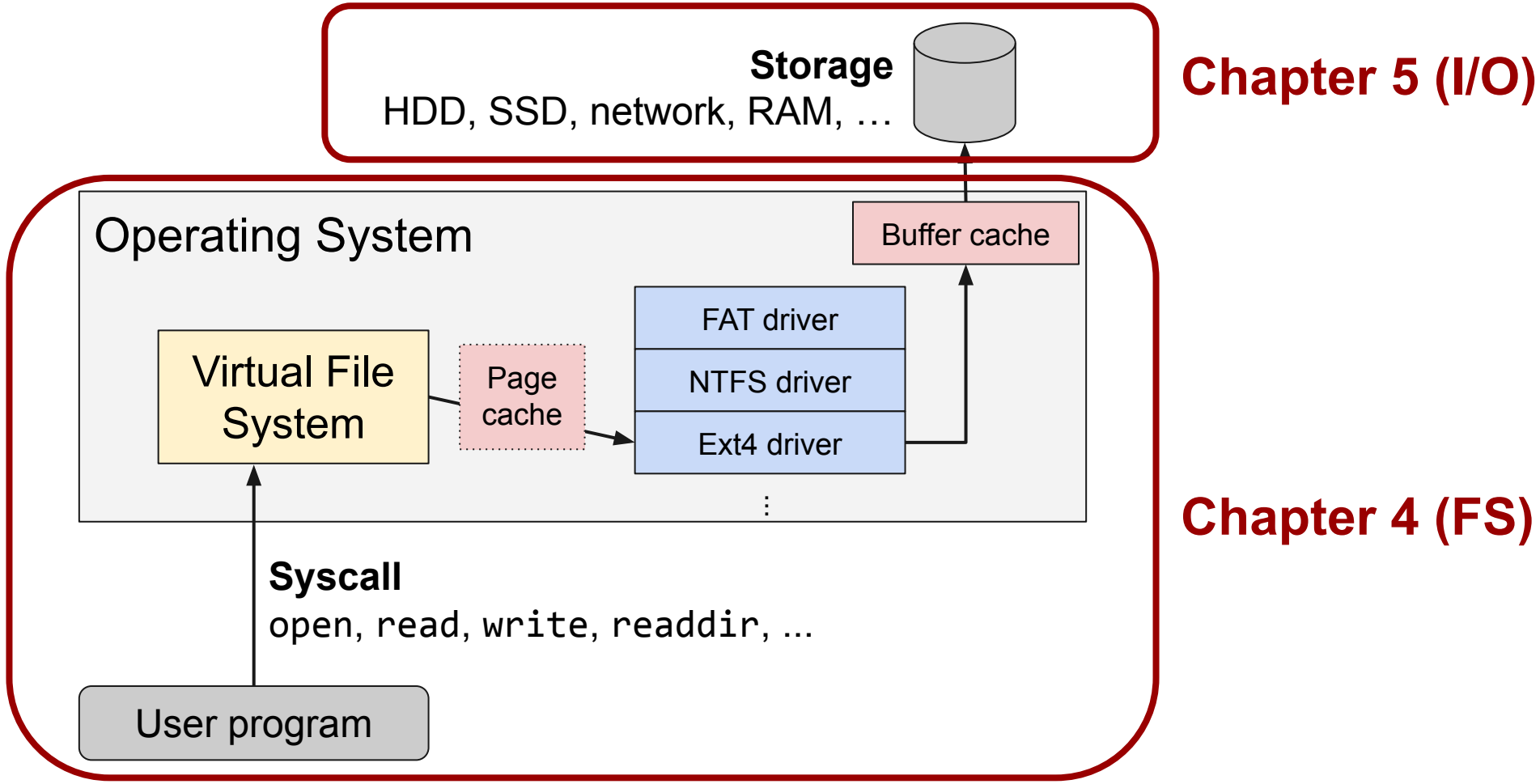


Overview



Chapter 4 (FS)

Overview



Files

- Abstract storage nodes
- File access:
  - Sequential vs. random access
- File types:
  - Regular files, directories, soft links
  - Special files (e.g., device files, metadata files)
- File structure:
  - OS' perspective: Files as streams of bytes
  - Program's perspective: Archives, Executables, etc.

File Structure

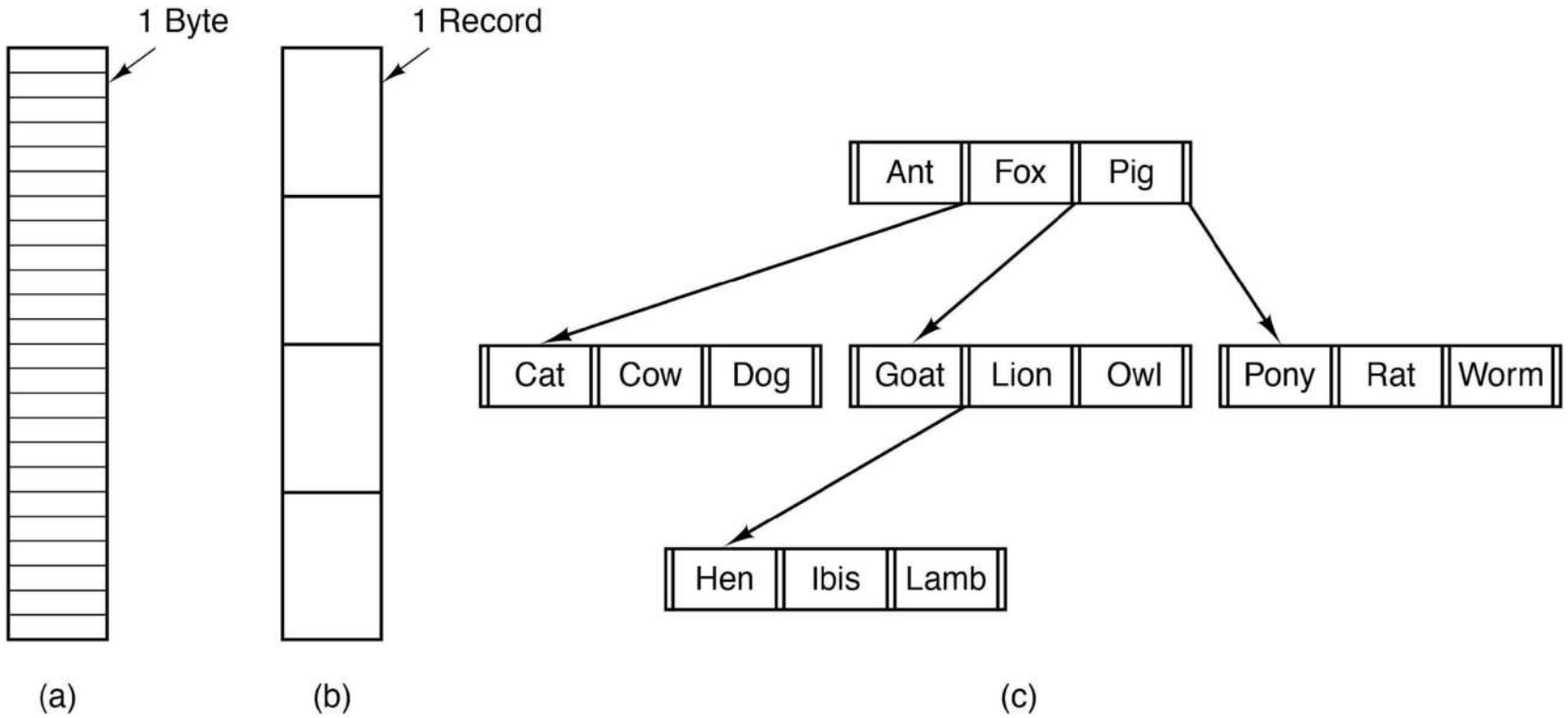


Figure 4-2. Three kinds of files. (a) Byte sequence. (b) Record sequence. (c) Tree.

Quiz

When does the OS itself care about the contents of a file?

File Naming

- Different file systems have different limitations/conventions for file names
- File extensions
- File name length
  - FAT12: 8.3 characters (later extended to 255)
  - Ext4: 255 characters
- Special characters in file names
  - FAT12: No "*/:<>?\| and more
  - Ext4: No '\0' and '/', or the special names "." and ".."
- Case sensitivity

File Naming

Figure 4.1 Some typical file extensions.

Extension	Meaning
.bak	Backup file
.c	C source program
.gif	Compuserve Graphical Interchange Format image
.html	World Wide Web HyperText Markup Language document
.jpg	Still picture encoded with the JPEG standard
.mp3	Music encoded in MPEG layer 3 audio format
.mpg	Movie encoded with the MPEG standard
.o	Object file (compiler output, not yet linked)
.pdf	Portable Document Format file
.ps	PostScript file
.tex	Input for the TEX formatting program
.txt	General text file
.zip	Compressed archive

Quiz

```
$ cat hello.c
#include <stdio.h>
void main() {
   printf("Hello, world\n");
}
$ gcc -o hello.pdf hello.c

$ file hello.pdf
hello.pdf: ELF 64-bit LSB shared object, x86-64, version 1 (SYSV) [...]

$ ./hello.pdf
Hello, world
```

How does the OS know `hello.pdf` is not really a PDF file?

File Types

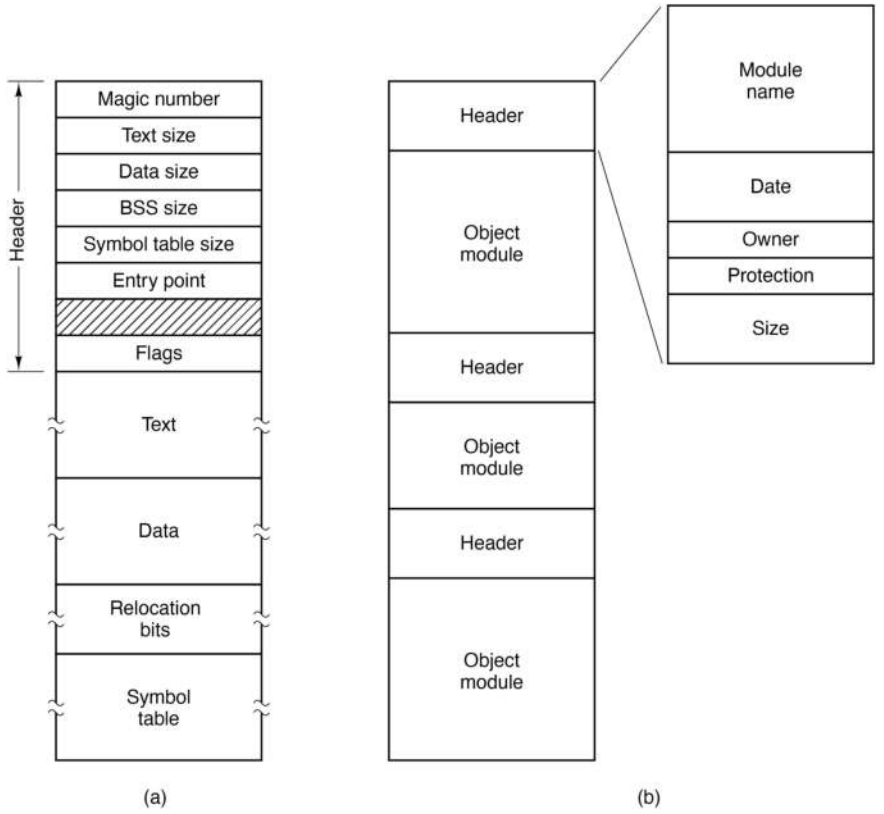


Figure 4-3. (a) An executable file. (b) An archive

File Attributes (1 of 2)

Figure 4-5. Some possible file attributes.

Attribute	Meaning
Protection	Who can access the file and in what way
Password	Password needed to access the file
Creator	ID of the person who created the file
Owner	Current owner
Read-only flag	0 for read/write; 1 for read only
Hidden flag	0 for normal; 1 for do not display in listings
System flag	0 for normal files; 1 for system file
Archive flag	0 for has been backed up; 1 for needs to be backed up
ASCII/binary flag	0 for ASCII file; 1 for binary file
Random access flag	0 for sequential access only; 1 for random access

File Attributes (2 of 2)

Figure 4-5. Some possible file attributes.

Attribute	Meaning
Temporary flag	0 for normal; 1 for delete file on process exit
Lock flags	0 for unlocked; nonzero for locked
Record length	Number of bytes in a record
Key position	Offset of the key within each record
Key length	Number of bytes in the key field
Creation time	Date and time the file was created
Time of last access	Date and time the file was last accessed
Time of last change	Date and time the file was last changed
Current size	Number of bytes in the file
Maximum size	Number of bytes the file may grow to

File Operations (1 of 2)

1. Create
2. Delete
3. Open
4. Close
5. Read
6. Write
7. Append
8. Seek

File Operations (2 of 2)

9. Get attributes
10. Set attributes
11. Rename

File Operations - UNIX

- Opening and reading file:

```
int fd = open("foo.txt", O_RDONLY);
char buf[512];
ssize_t bytes_read = read(fd, buf, 512);
close(fd);
printf("read %zd: %s\n", bytes_read, buf);
```

- Opening a file returns a handle (file descriptor) for future operations.
- Any function may return an error, e.g.:
  - `-ENOENT`: File does not exist
  - `-EBADF`: Bad file descriptor

File Operations - UNIX

- Seeking in files:

```
int fd = open("foo.txt", O_RDONLY);
lseek(fd, 128, SEEK_CUR);
char buf[8];
read(fd, buf, 8);
close(fd);
```

- Move current position in file forwards or backwards.

File Operations - UNIX

- Writing files:

```
int fd = open("foo.txt", O_WRONLY | O_CREAT | O_TRUNC);
char buf[] = "Hi there";
write(fd, buf, strlen(buf));
close(fd);
```

- `O_CREAT`: Create file if it does not exist
- `O_TRUNC`: "Truncate" file to size 0 if it exists (i.e., throw away current file contents)

File Operations - UNIX

- Removing file:
  - `unlink(“foo.txt”);`
- Renaming file:
  - `rename(“foo.txt”, “bar.txt”);`
- Change file permission attribute:
  - `chmod(“foo.txt”, 0755);`
- Change file owner attribute:
  - `chown(“foo.txt”, uid, gid);`

Example Program Using File System Calls (1 of 3)

```
/* File copy program. Error checking and reporting is minimal. */

#include <sys/types.h>                  /* include necessary header files */
#include <fcntl.h>
#include <stdlib.h>
#include <unistd.h>

int main(int argc, char *argv[]);       /* ANSI prototype */

#define BUF_SIZE 4096                   /* use a buffer size of 4096 bytes */
#define OUTPUT_MODE 0700                /* protection bits for output file */

int main(int argc, char *argv[])
{
    int in_fd, out_fd, rd_count, wt_count;
    char buffer[BUF_SIZE];

    if (argc != 3) exit(1);             /* syntax error if argc is not 3 */

    /* Open the input file and create the output file */
```

Figure 4-6. A simple program to copy a file.

Example Program Using File System Calls (2 of 3)

```
if (argc != 3) exit(1);                          /* syntax error if argc is not 3 */

/* Open the input file and create the output file */
in_fd = open(argv[1], O_RDONLY);                 /* open the source file */
if (in_fd < 0) exit(2);                          /* if it cannot be opened, exit */
out_fd = creat(argv[2], OUTPUT_MODE);            /* create the destination file */
if (out_fd < 0) exit(3);                         /* if it cannot be created, exit */

/* Copy loop */
while (TRUE) {
        rd_count = read(in_fd, buffer, BUF_SIZE); /* read a block of data */
if (rd_count <= 0) break;                        /* if end of file or error, exit loop */
        wt_count = write(out_fd, buffer, rd_count); /* write data */
```

Figure 4-6. A simple program to copy a file.

Example Program Using File System Calls (3 of 3)

```
        /* Copy loop */
        while (TRUE) {
                rd_count = read(in_fd, buffer, BUF_SIZE); /* read a block of data */
        if (rd_count <= 0) break;                         /* if end of file or error, exit loop */
                wt_count = write(out_fd, buffer, rd_count); /* write data */
                if (wt_count <= 0) exit(4);               /* wt_count <= 0 is an error */
        }

        /* Close the files */
        close(in_fd);
        close(out_fd);
        if (rd_count == 0)                                /* no error on last read */
                exit(0);
        else
                exit(5);                                  /* error on last read */
}
```

Figure 4-6. A simple program to copy a file.

Directories

- Data structures organizing and maintaining information about files
- Often stored as file entries with special attributes
- Directories are denoted by “/” (Unix) or “\” (Windows)
- Special directory entries:
  - `.` Current directory
  - `..` Parent directory

Single-Level Directory Systems

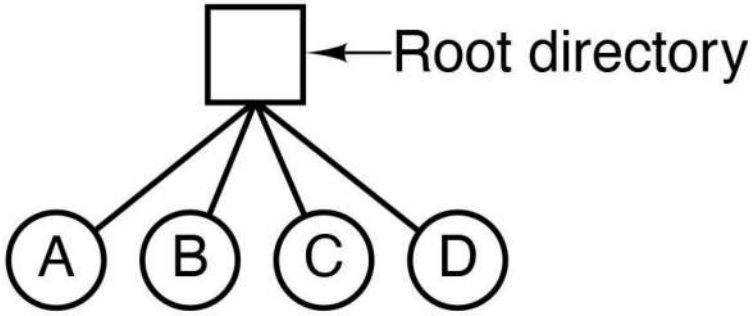


Figure 4-7. A single-level directory system containing four files.

Hierarchical Directory Systems

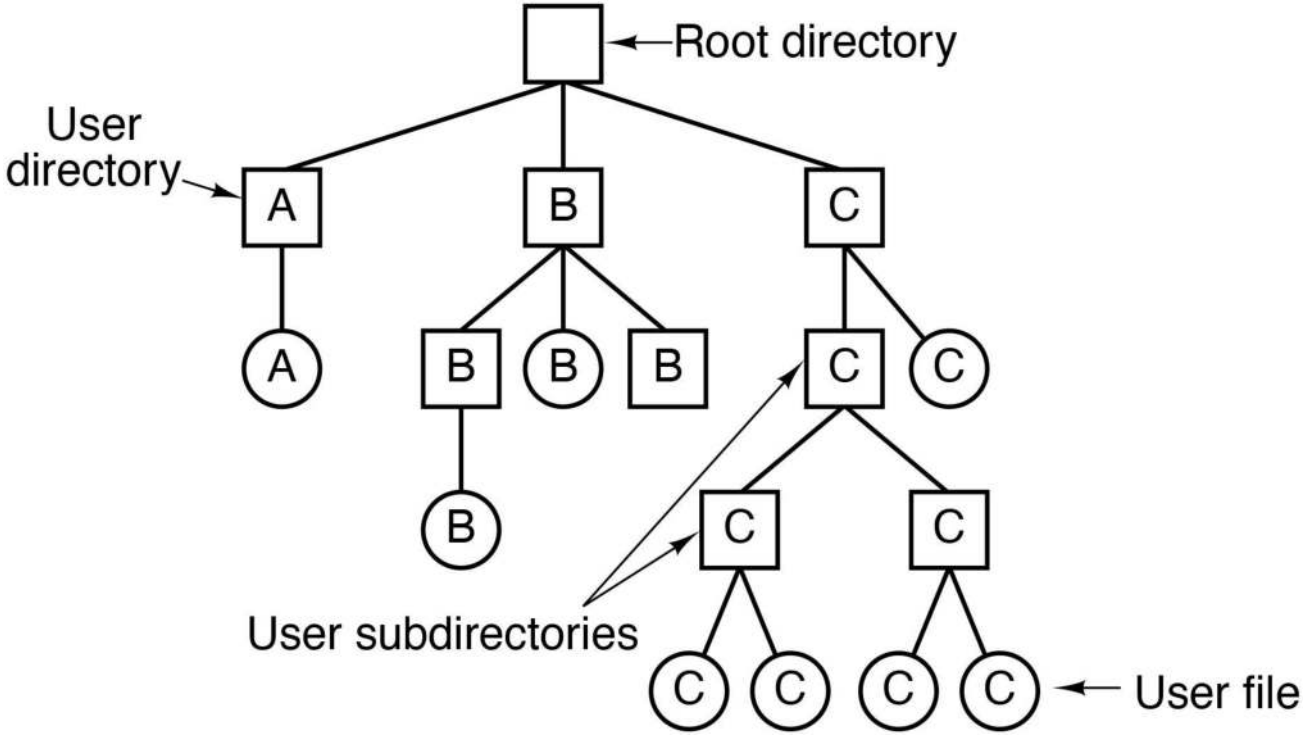


Figure 4-8. A hierarchical directory system.

Path Names

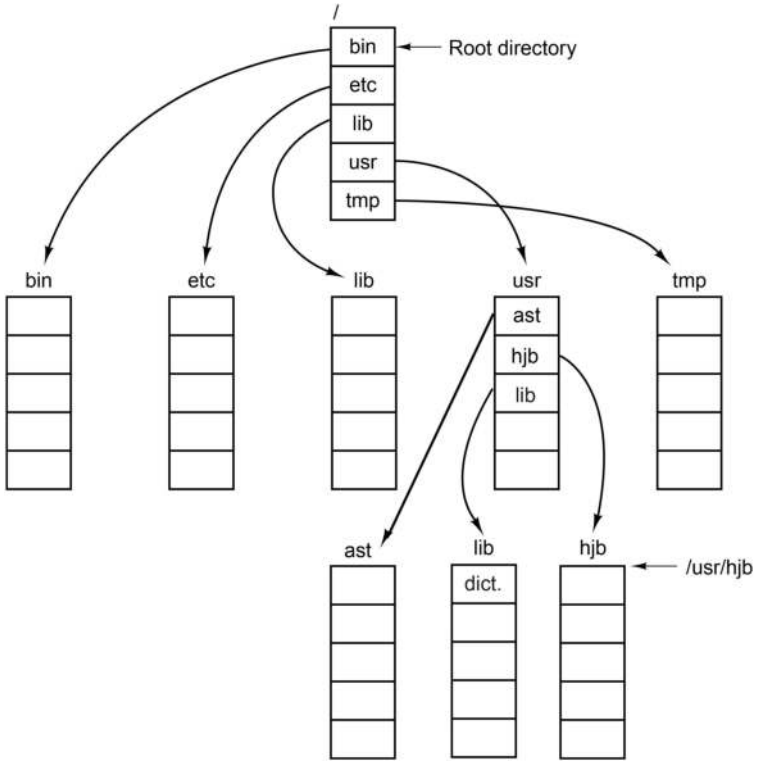


Figure 4-9. A UNIX directory tree.

Directory Operations

1. Create
2. Delete
3. Opendir
4. Closedir
5. Readdir
6. Rename
7. Link
8. Unlink

Directory Operations - UNIX

- Reading current directory contents:

```
DIR *dirp = opendir(“.”);

struct dirent *dirent;
while ((dirent = readdir(dirp)))
  printf(“%s\n”, dirent->d_name);

closedir(dirp);
```

Virtual File Systems (1 of 2)

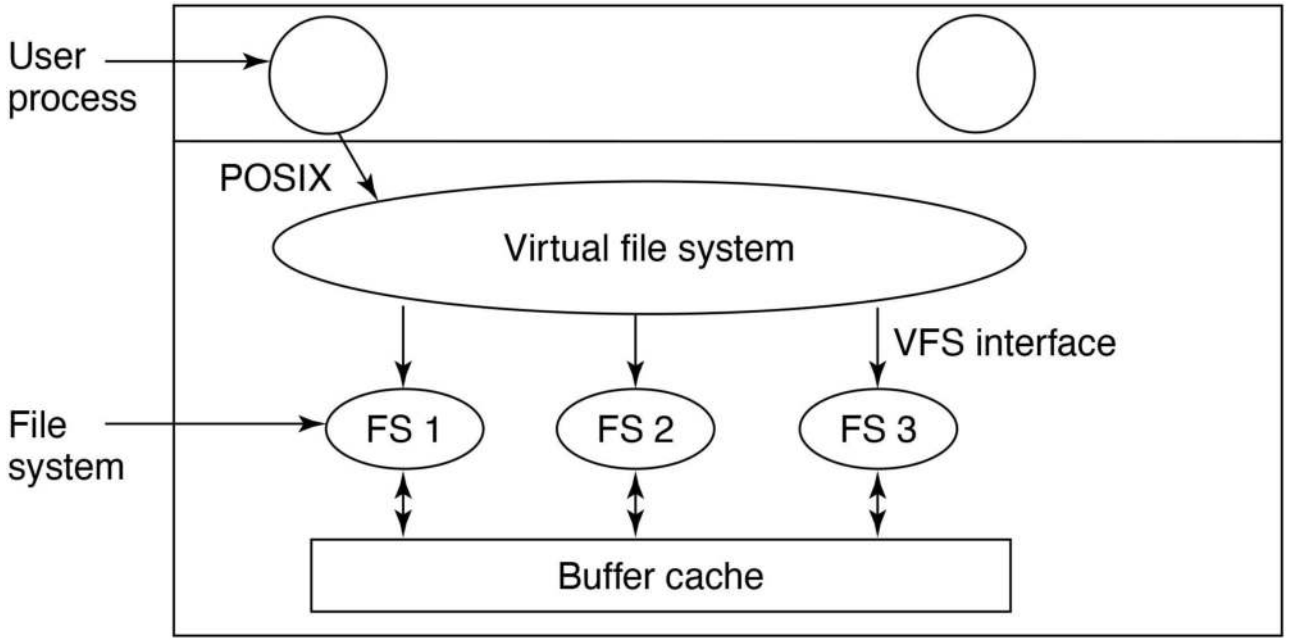


Figure 4-21. Position of the virtual file system.

Virtual File Systems (2 of 2)

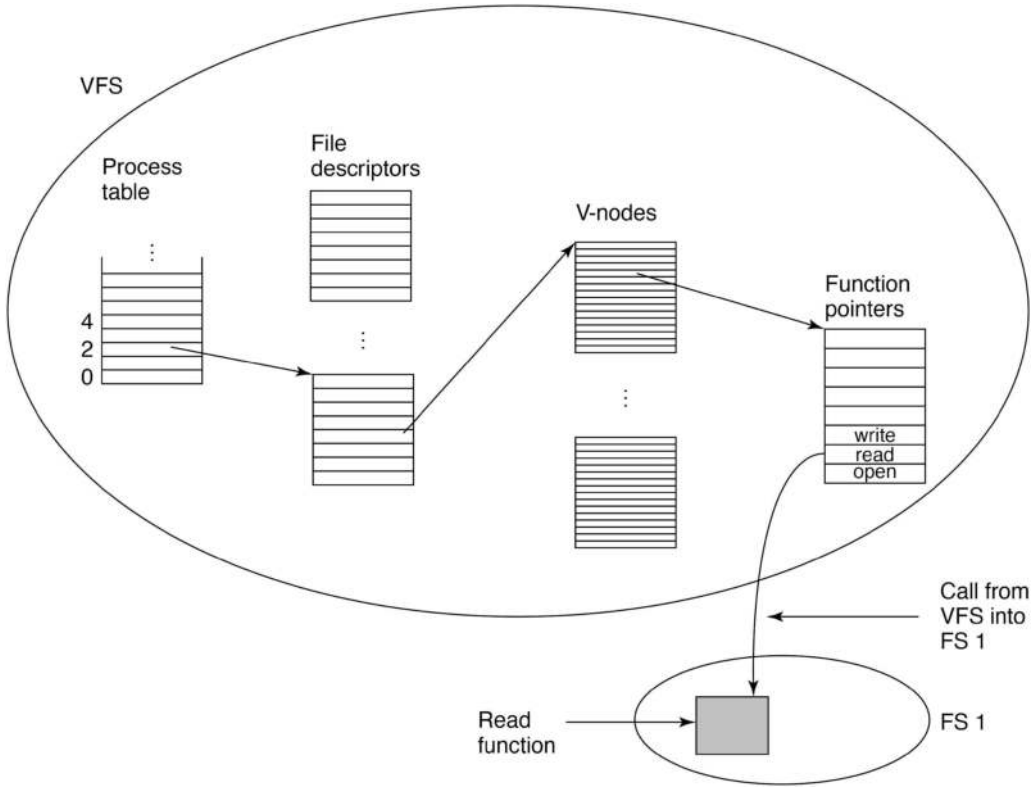


Figure 4-22. A simplified view of the data structures and code used by the VFS and concrete file system to do a **read**.

File System Implementation

- How to store files?
- How to implement directories?
- How to manage disk space?
- How to ensure file system performance?
- How to ensure file system dependability?

File System Implementation

- **How to store files?**
- How to implement directories?
- How to manage disk space?
- How to ensure file system performance?
- How to ensure file system dependability?

File System Layout

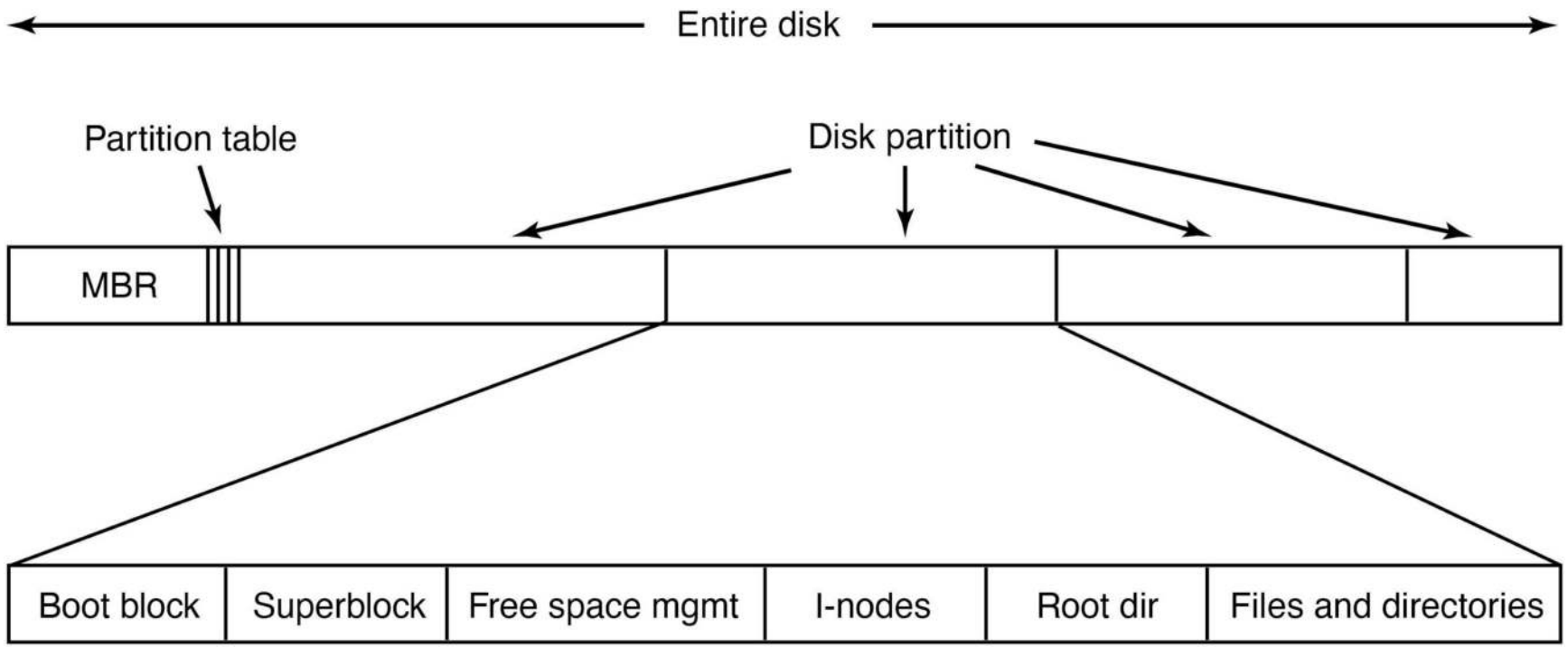


Figure 4-10. A possible file system layout.

Unified Extensible Firmware Interface (UEFI)

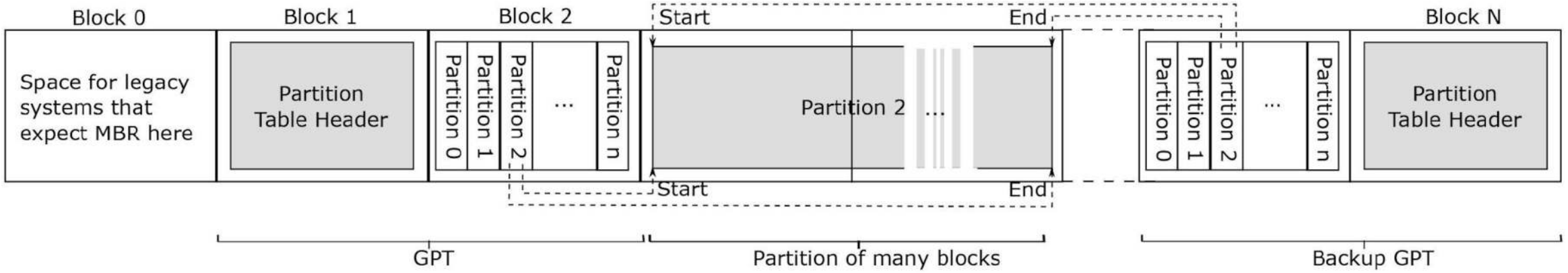


Figure 4-11. Layout for UEFI with partition table

How to Store Files on the Disk?

Contiguous allocation

- Store files as a contiguous stream of bytes
- Requires max file size, prone to fragmentation

Implementing Files Contiguous Layout

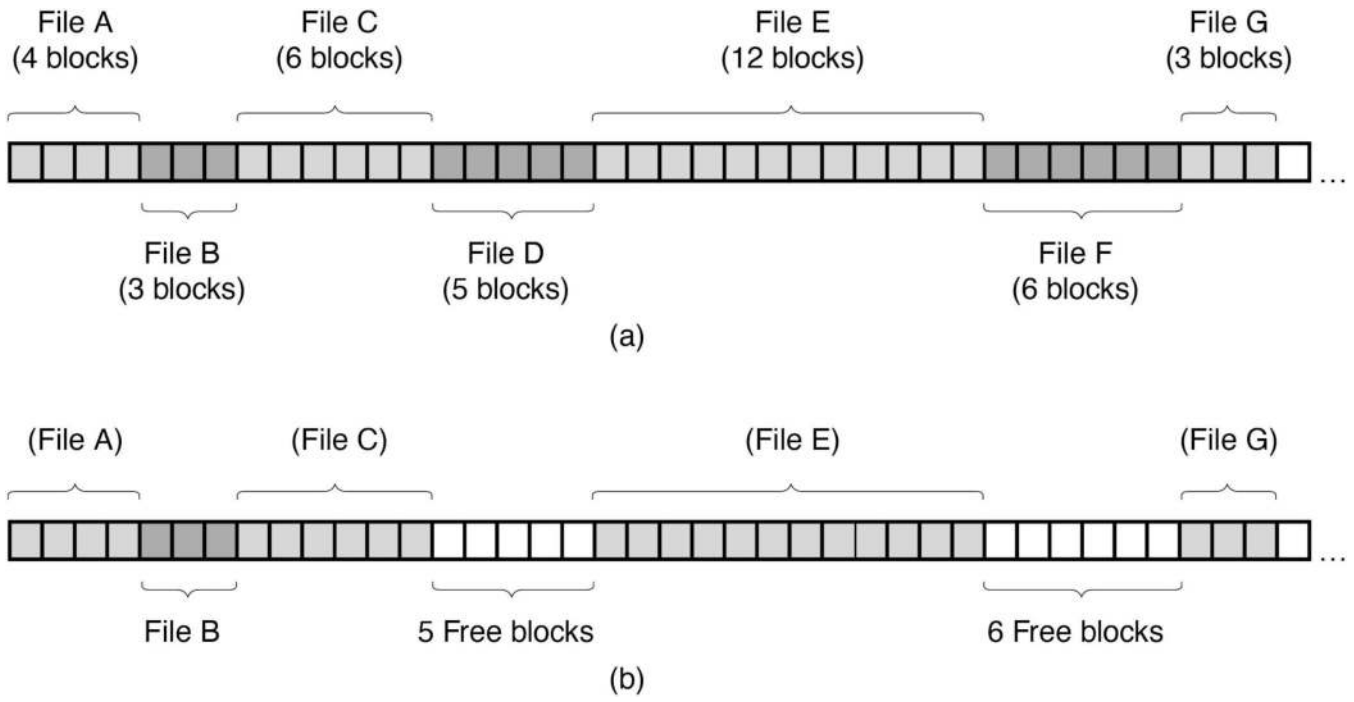


Figure 4-12. (a) Contiguous allocation of disk space for seven files. (b) The state of the disk after files **D** and **F** have been removed.

How to Store Files on the Disk?

Contiguous allocation

- Store files as a contiguous stream of bytes
- Requires max file size, prone to fragmentation

Block-based strategies

- Linked List
- File Allocation Table
- i-nodes

Implementing Files Linked List Allocation

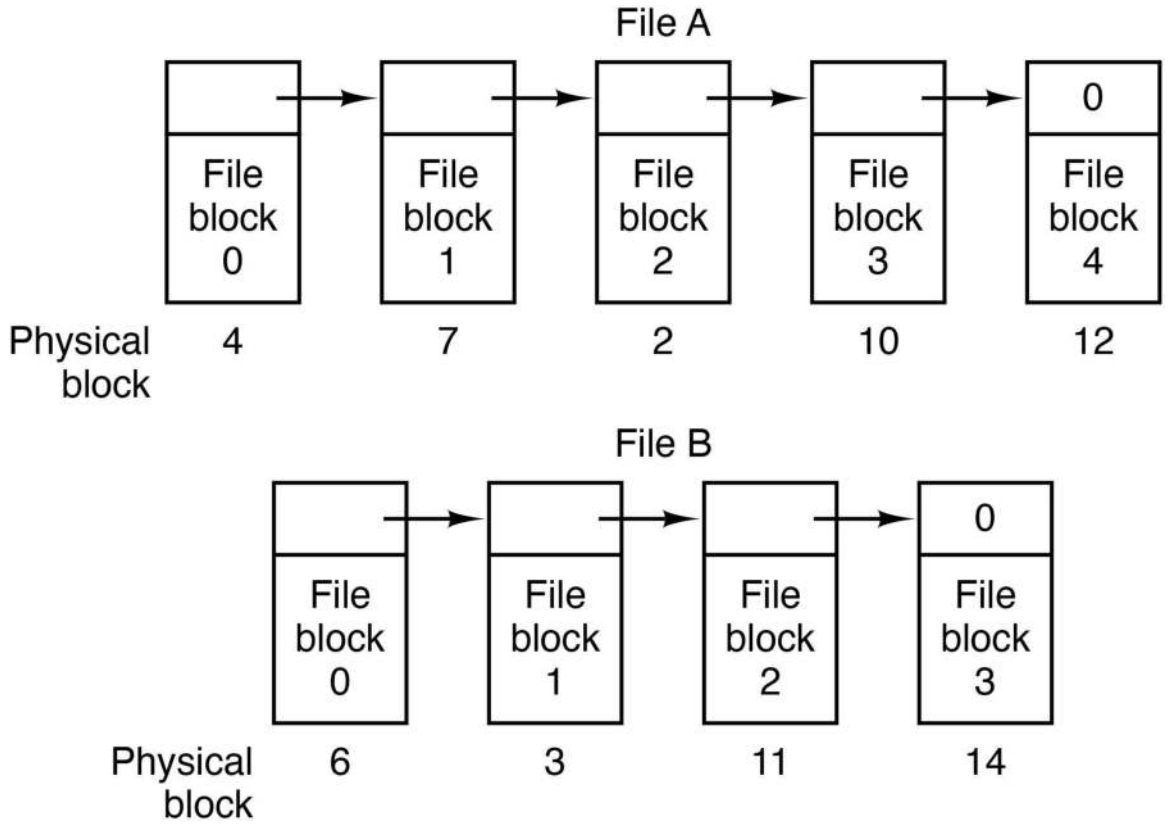


Figure 4-13. Storing a file as a linked list of disk blocks.

Implementing Files Linked List - Table in Memory

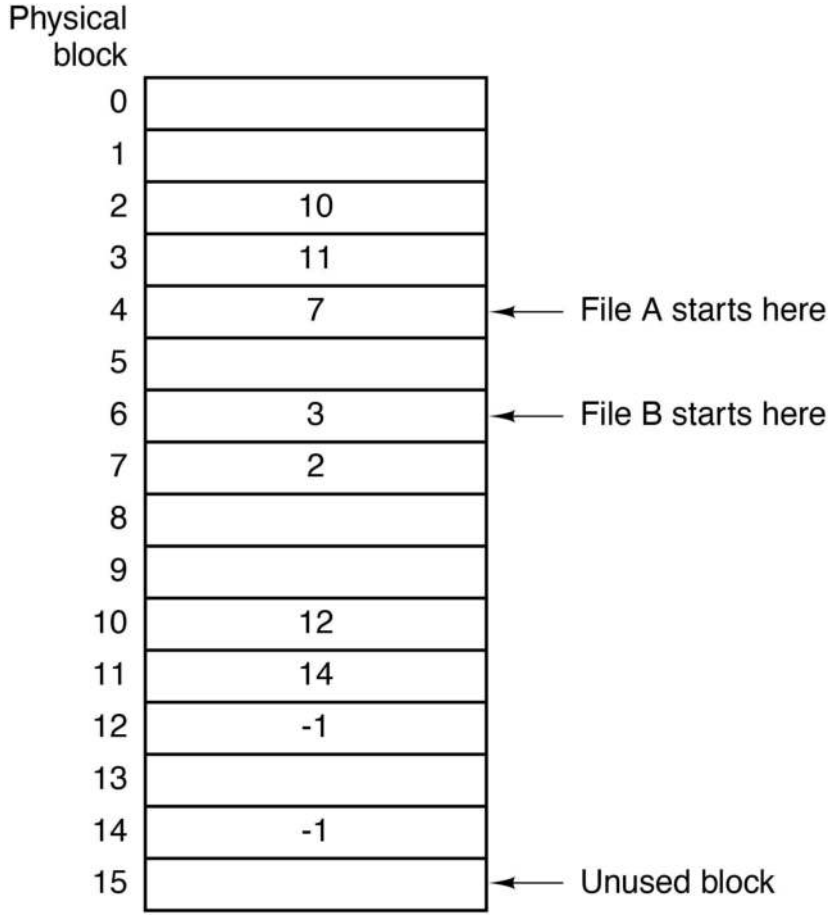


Figure 4-14. Linked list allocation using a file allocation table in main memory.

Implementing Files I-nodes

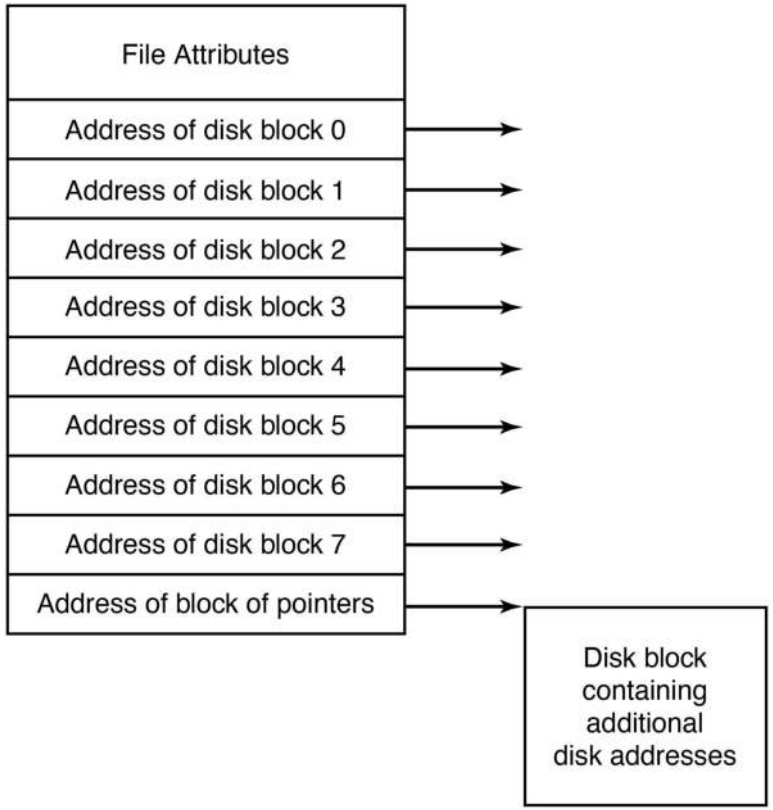


Figure 4-15. An example i-node.

Implementing Files I-nodes with Indirect Blocks

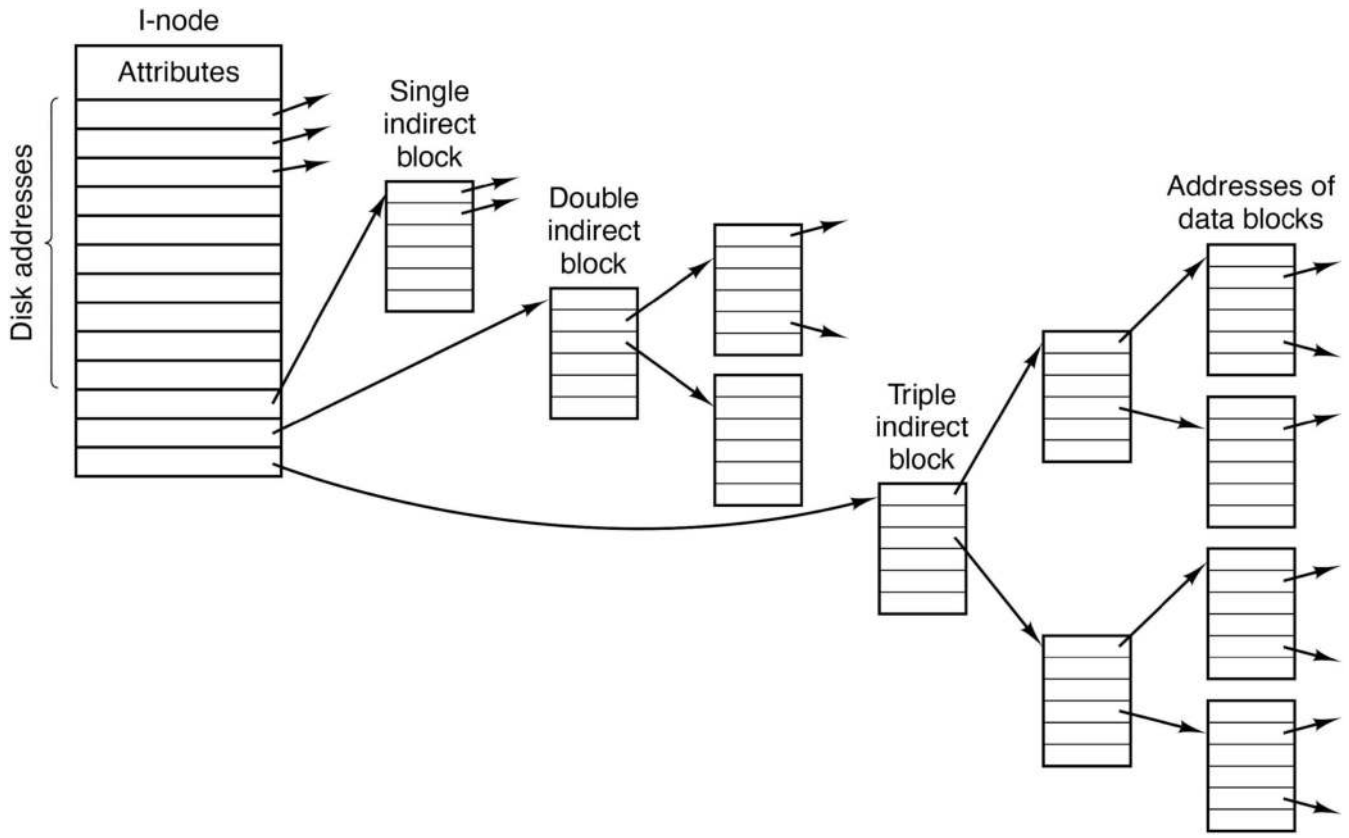


Figure 4-35. A UNIX i-node

File System Implementation

- How to store files?
- **How to implement directories?**
- How to manage disk space?
- How to ensure file system performance?
- How to ensure file system dependability?

Directory Implementation

- How to store directory entries and attributes?
- How to find the root directory?
- How to find any other directory?

Directory Entries and Attributes

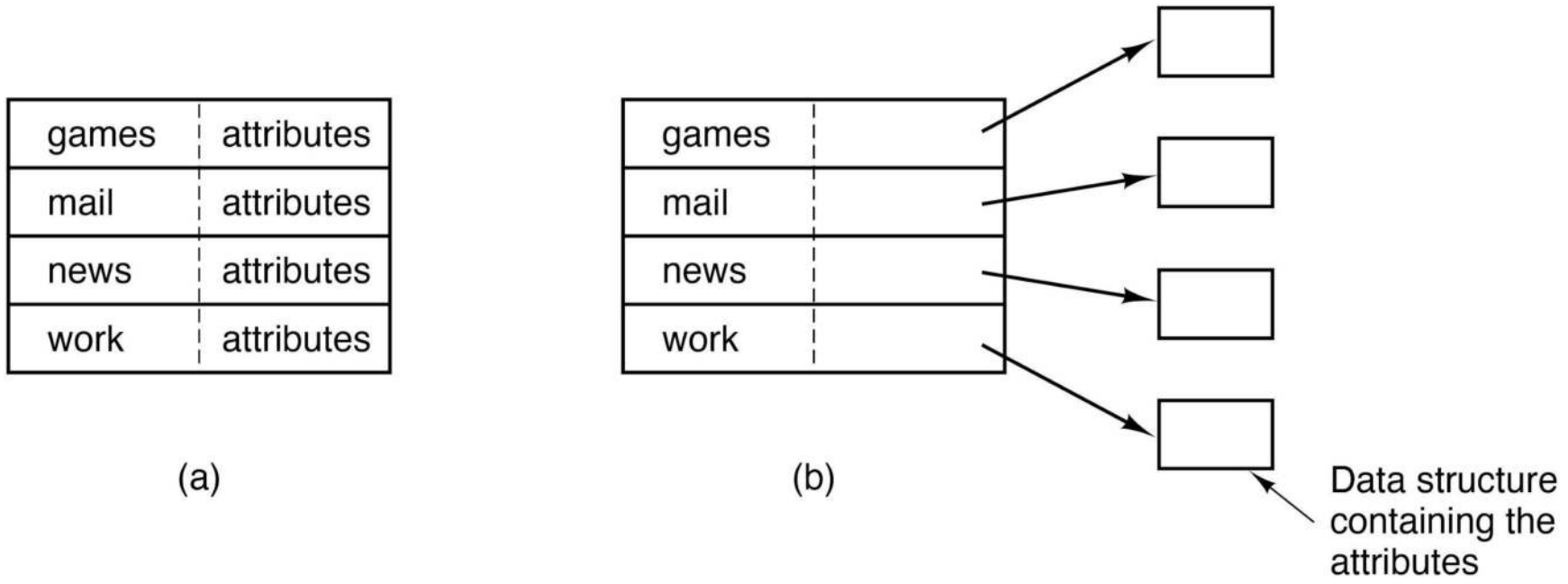
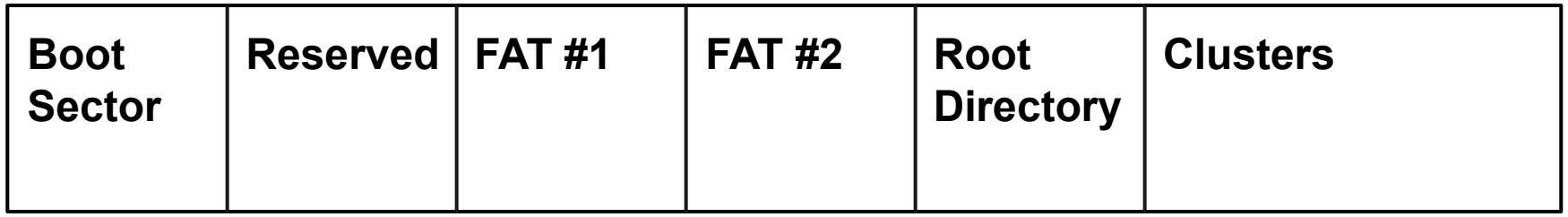


Figure 4-16. (a) A simple directory containing fixed-size entries with the disk addresses and attributes in the directory entry. (b) A directory in which each entry just refers to an i-node.

FAT12/FAT16 Layout

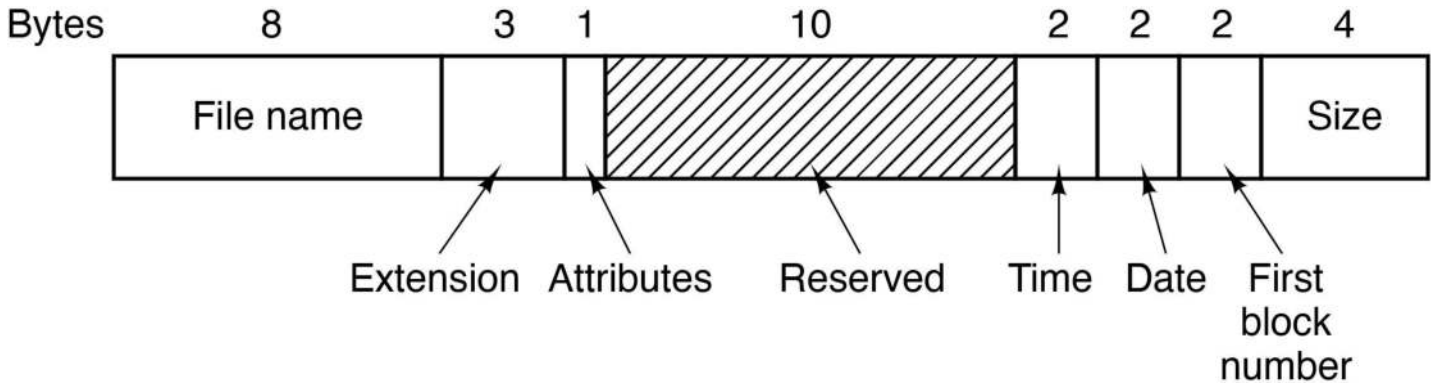


- Reserved later used for FS information
- FAT #1/#2: preallocated File Allocation Tables
- Preallocated root directory
- Clusters represent addressable blocks:
  - FAT contains chains indexed by starting cluster

Quiz

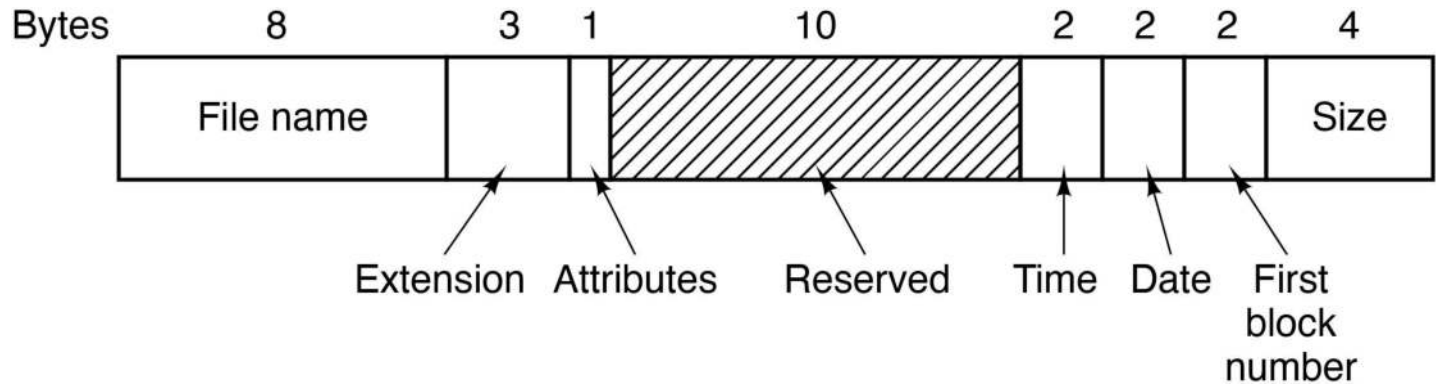
Why 2 FATs?

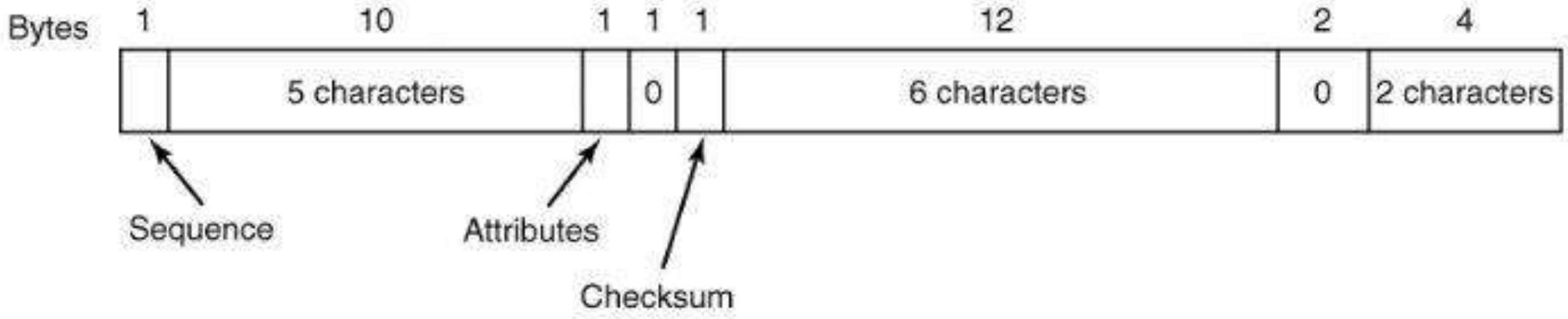
FAT12/16 Directory Entry



- Filename in 8.3 format
  - Later on, Long File Name (LFN) entries used multiple entries for a single file, setting several reserved bits for backwards compatibility

FAT12/16 Directory Entry





Other Ways to Handle Long File Names

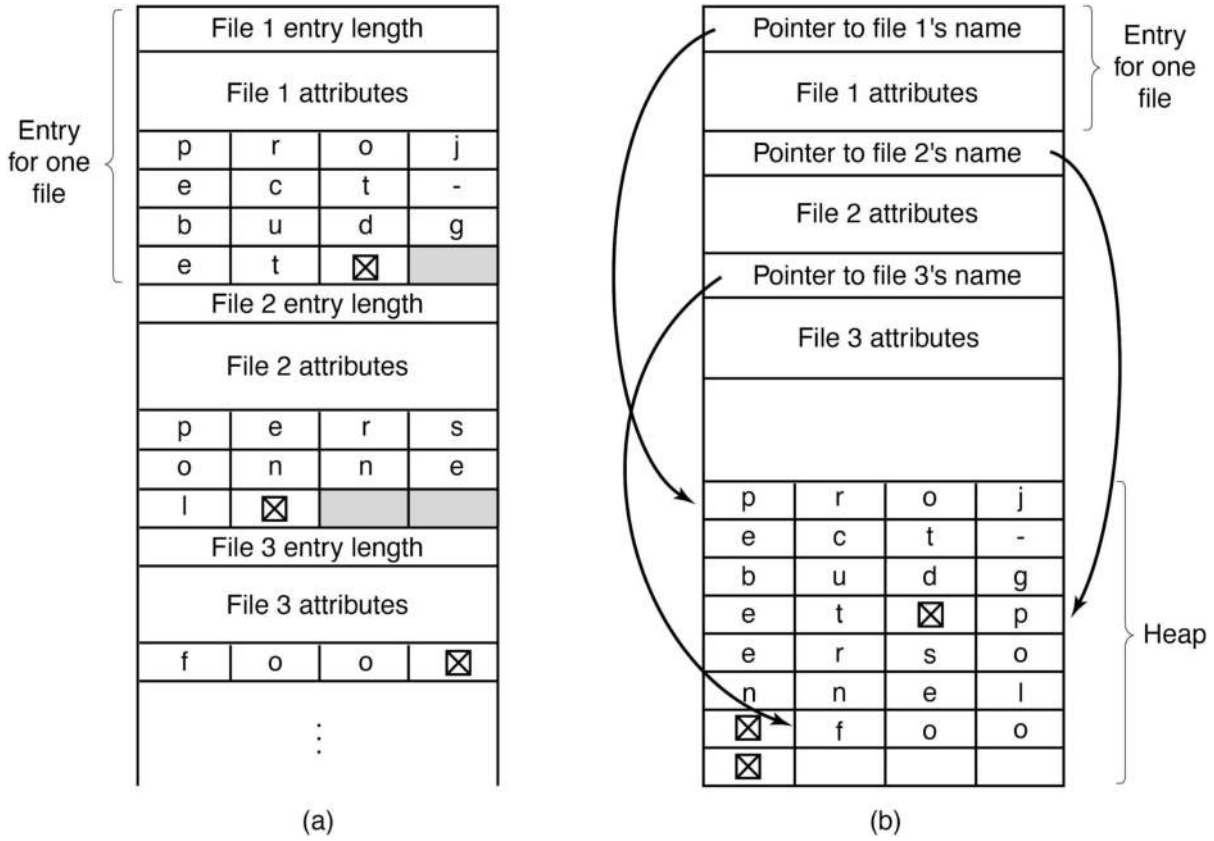


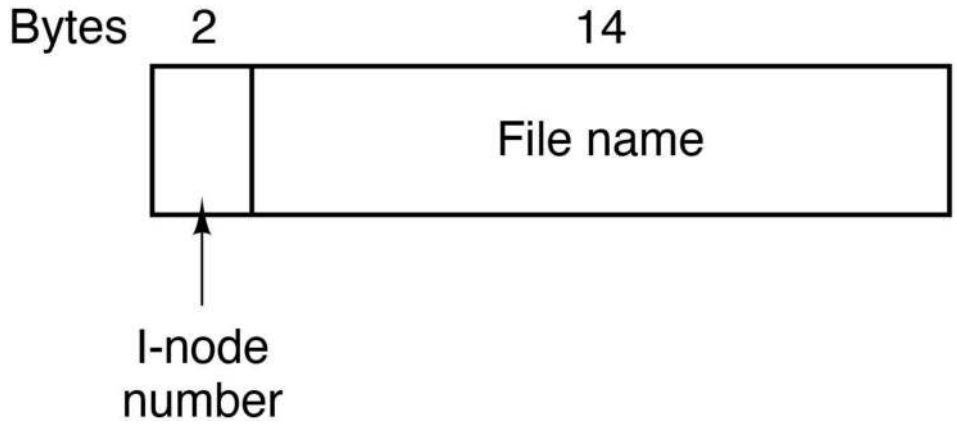
Figure 4-17. Two ways of handling long file names in a directory. (a) In-line. (b) In a heap.

FAT Partition Size

Figure 4-33. Maximum partition size for different block sizes. The empty boxes represent forbidden combinations.

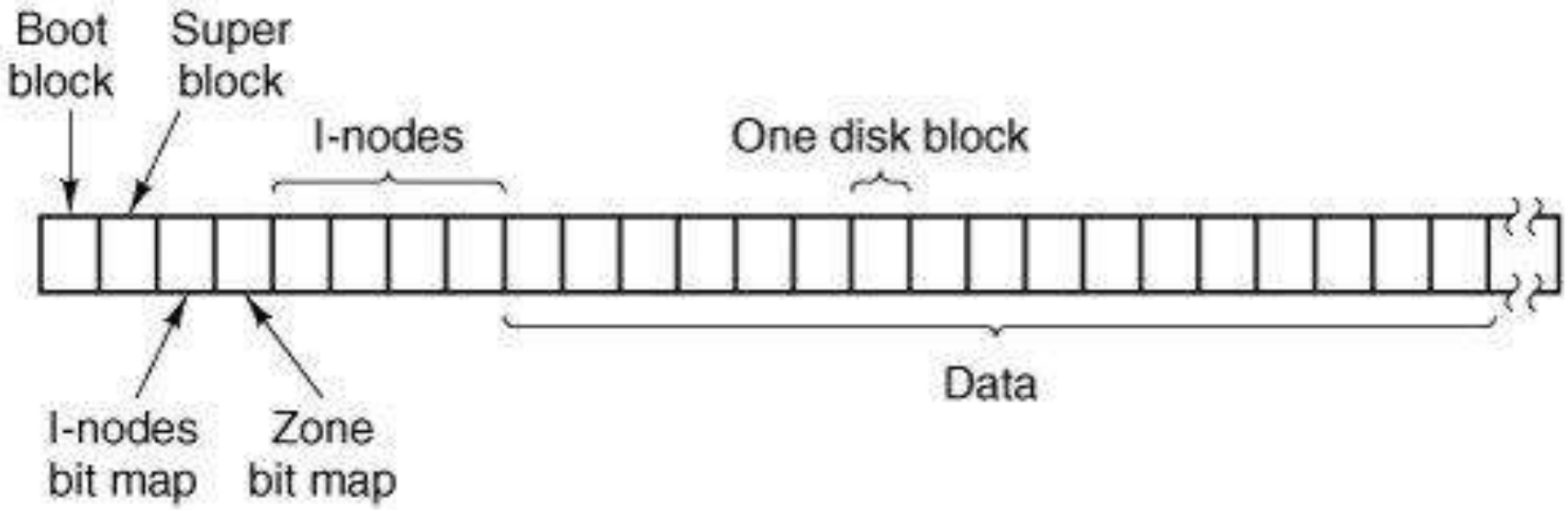
Block size	FAT-12	FAT-16	FAT-32
0.5 KB	2 MB		
1 KB	4 MB		
2 KB	8 MB	128 MB	
4 KB	16 MB	256 MB	1 TB
8 KB		512 MB	2 TB
16 KB		1024 MB	2 TB
32 KB		2048 MB	2 TB

UNIX Directory Entry



- Directory entry stores only name and #i-node
- Attributes are stored in the i-node
- One i-node per file, first i-node is the root
- Where are i-nodes stored?

UNIX FS Layout (MINIX Example)



Directory Lookup

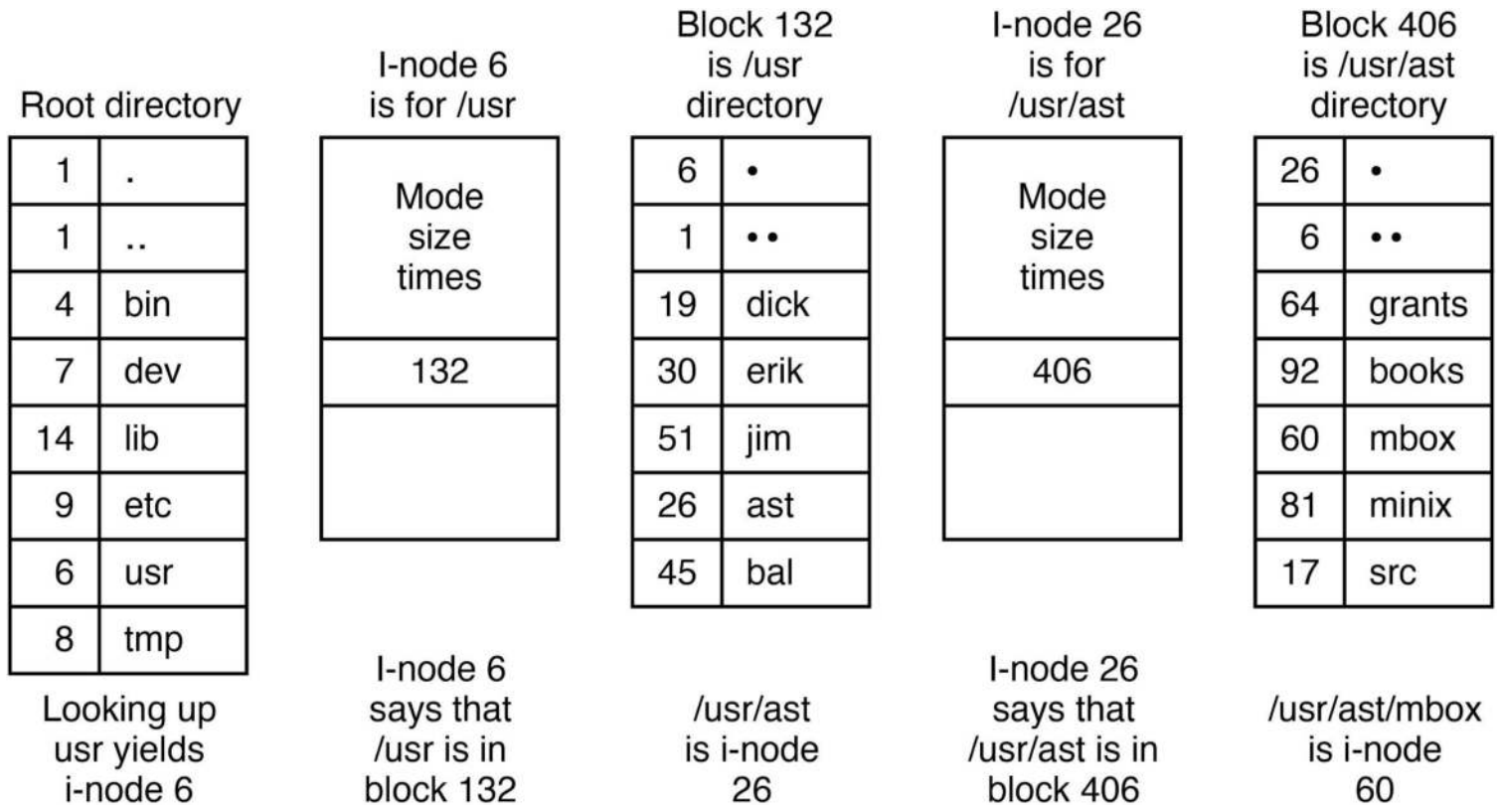


Figure 4-36. The steps in looking up **/usr/ast/mbox**.

Quiz

How many disk accesses for reading first byte from /usr/ast/mbox?

Quiz

How many disk accesses for reading first byte from /usr/ast/mbox?

- *7 (3 directories, 3 i-nodes, 1 file data)*

Quiz

How many disk accesses for reading first byte from /usr/ast/mbox?

- *7 (3 directories, 3 i-nodes, 1 file data)*

How many on FAT?

Quiz

How many disk accesses for reading first byte from /usr/ast/mbox?

- *7 (3 directories, 3 i-nodes, 1 file data)*

How many on FAT?

- *4 (3 directories, 1 file data)*

Shared Files (1 of 2)

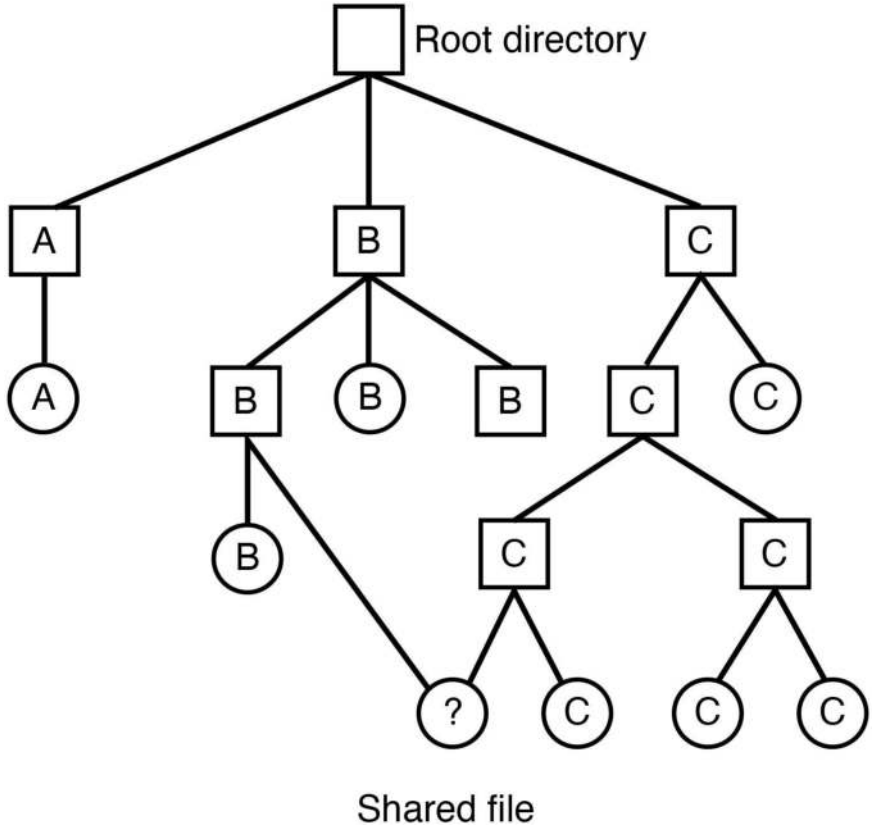


Figure 4-18. File system containing a shared file.

Shared Files (2 of 2)

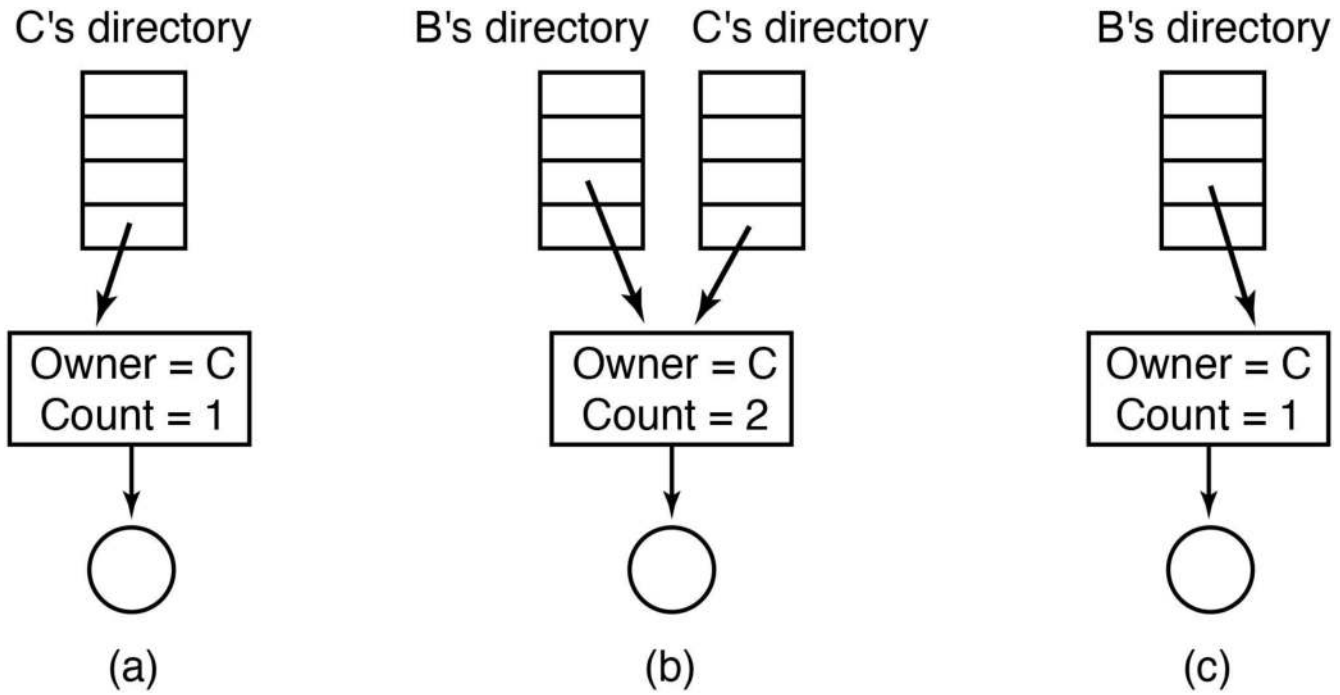


Figure 4-19. (a) Situation prior to linking. (b) After the link is created. (c) After the original owner removes the file.

Hard Links vs. Soft Links

- **Hard links** reference a shared file (i-node)
  - The file is removed only with no hard links left
  - Space-efficient (only 1 directory entry per hard link)
  - Good to manage shared files across owners

- **Soft** (or **symbolic**) **links** reference a file name
  - Removing the file renders its soft links invalid
  - Less space-efficient (need 1 i-node per soft link)
  - More flexible (can reference file names across FSes)

File System Implementation

- How to store files?
- How to implement directories?
- **How to manage disk space?**
- How to ensure file system performance?
- How to ensure file system dependability?

Block Size

- How do choose the block size?
- Important speed-space tradeoff to consider

- Larger block sizes allow transferring more data per block
  - Better overall data rates
- Smaller block sizes reduce space overhead for small files
  - Less fragmentation

Block Size Impact

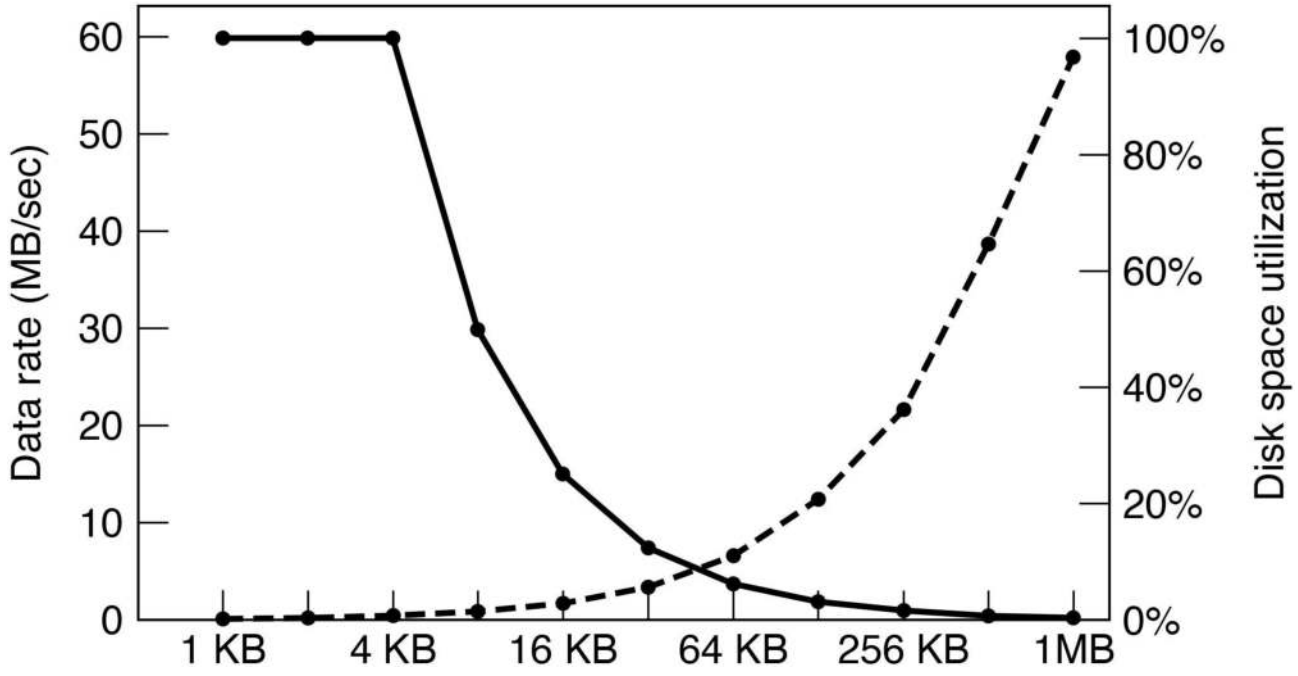


Figure 4-23. The dashed curve (left-hand scale) gives the data rate of a disk. The solid curve (right-hand scale) gives the disk space efficiency. All files are 4 KB.

Keeping Track of Free Blocks (1 of 2)

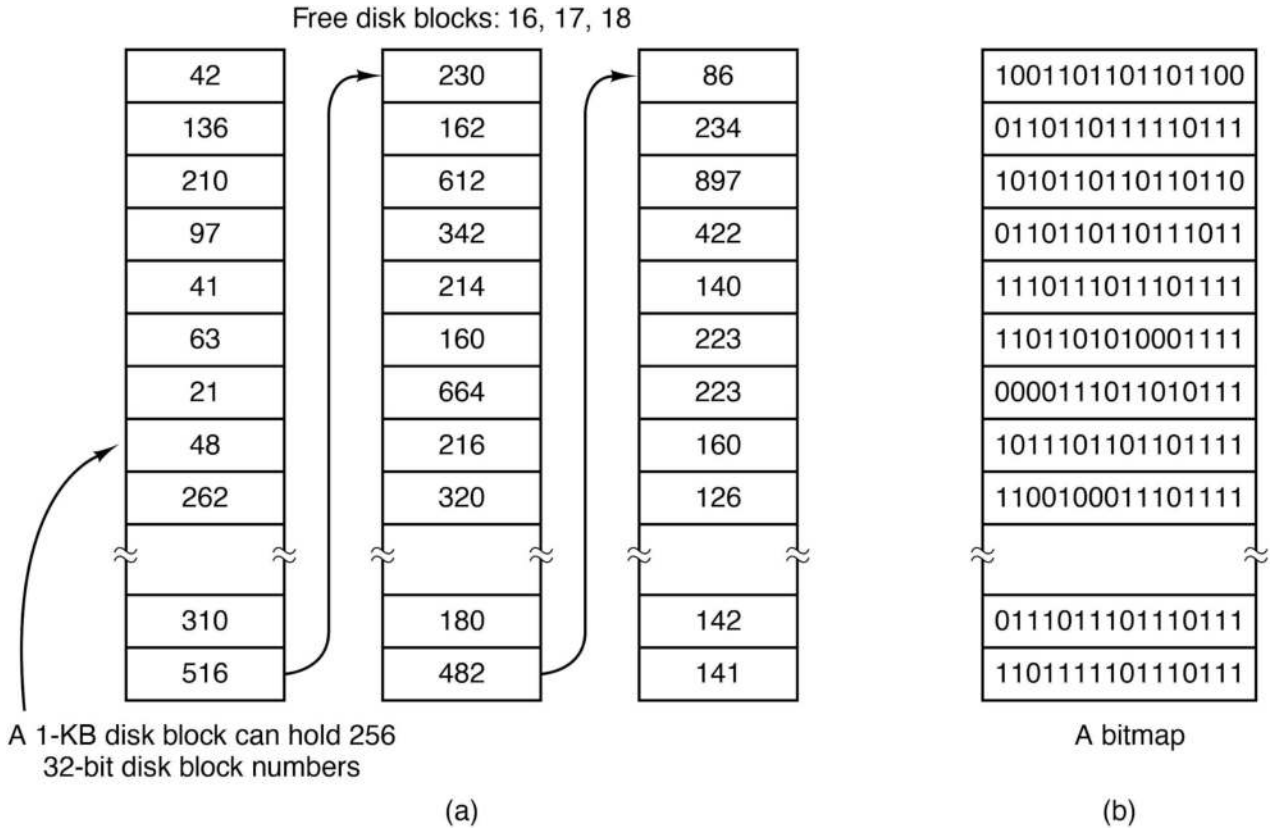


Figure 4-24. (a) Storing the free list on a linked list. (b) A bitmap.

Quiz

Which organization incurs more space overhead?

Hint: consider an almost empty disk scenario and an almost full disk scenario.

Keeping Track of Free Blocks (2 of 2)

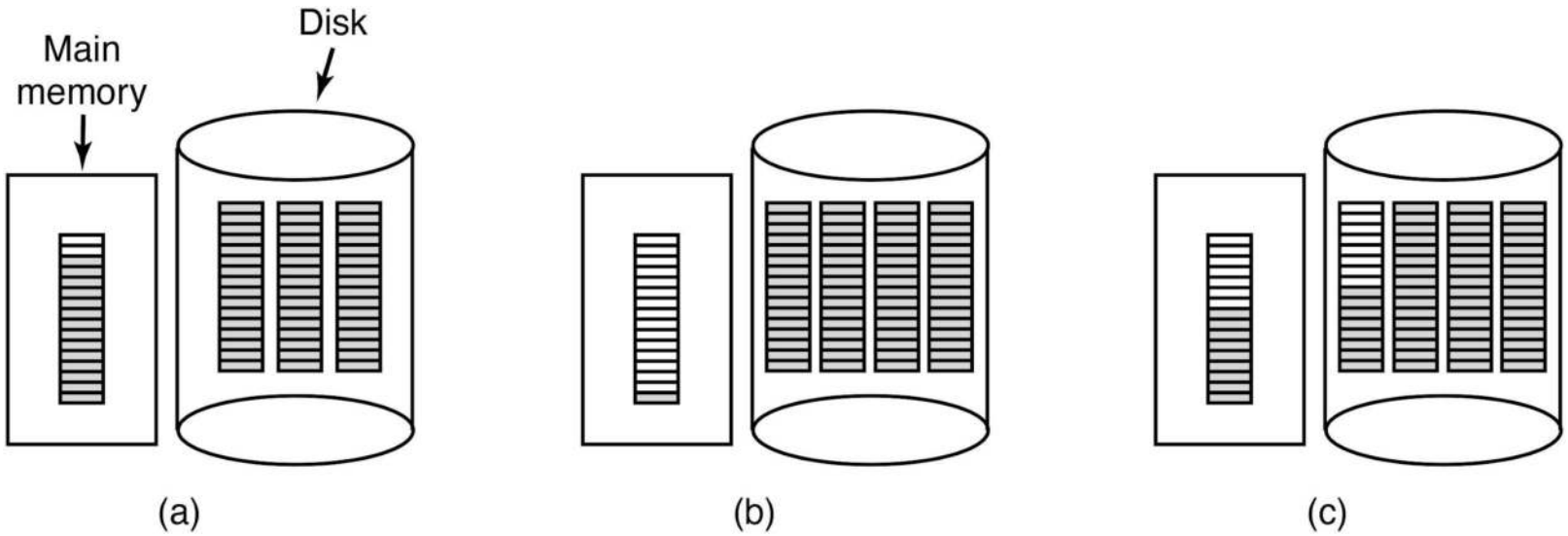


Figure 4-25. (a) An almost-full block of pointers to free disk blocks in memory and three blocks of pointers on disk. (b) Result of freeing a three-block file. (c) An alternative strategy for handling the three free blocks. The shaded entries represent pointers to free disk blocks.

Disk Quotas

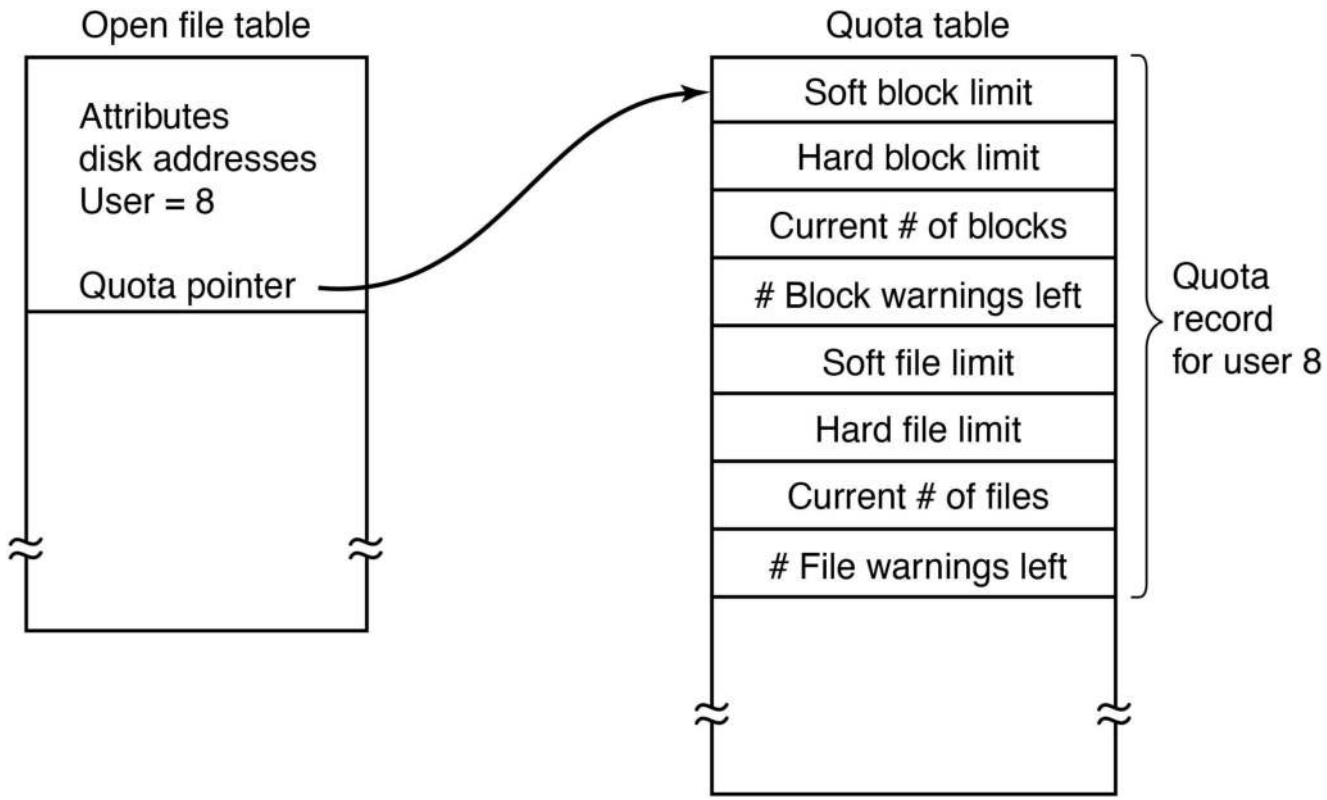


Figure 4-26. Quotas are kept track of on a per-user basis in a quota table.

File System Implementation

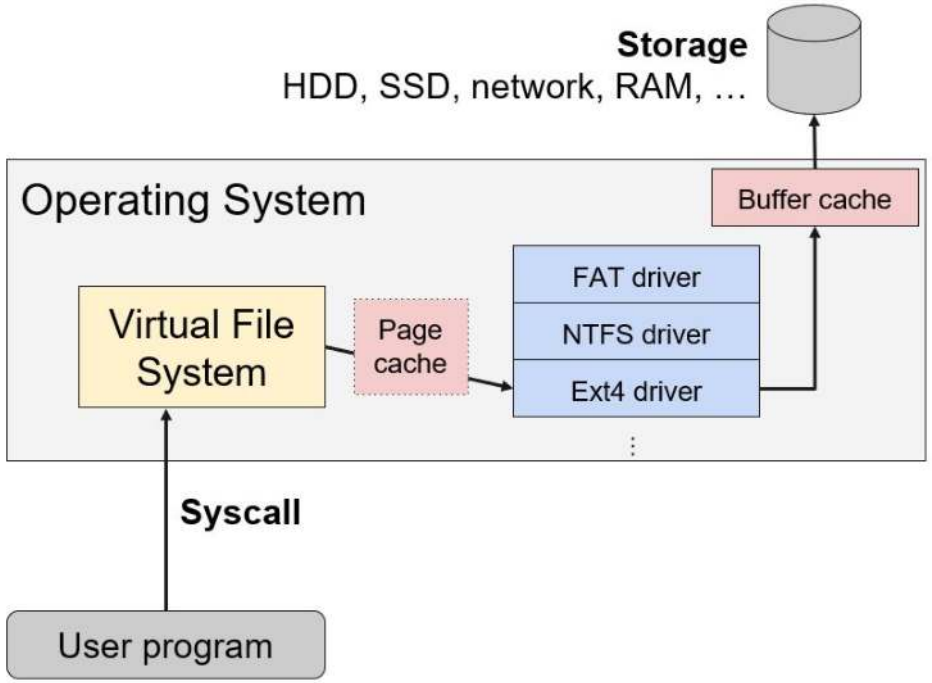
- How to store files?
- How to implement directories?
- How to manage disk space?
- **How to ensure file system performance?**
- How to ensure file system dependability?

File System Performance

How to optimize file system performance?

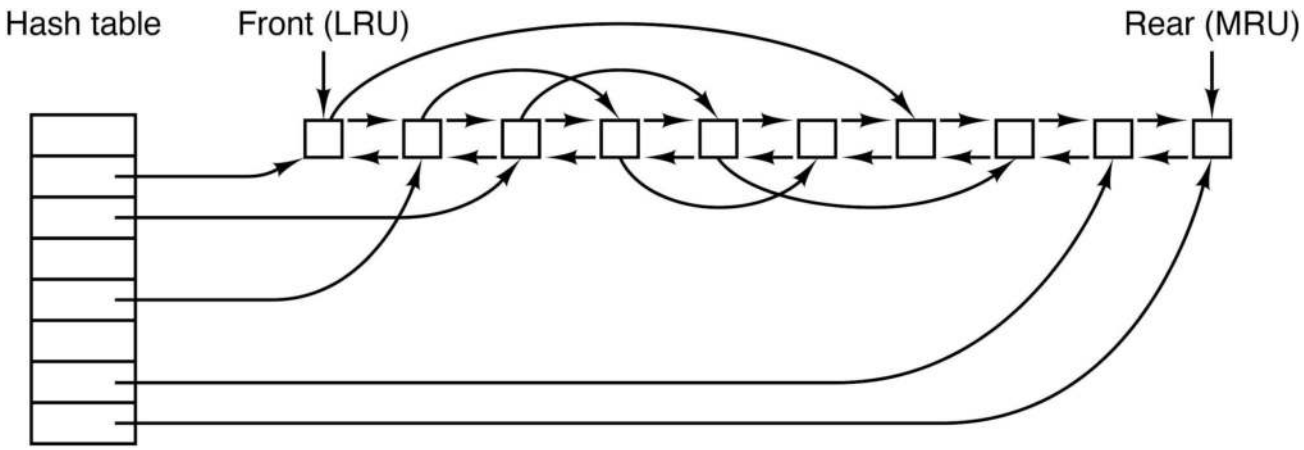
- Minimize disk access
- Minimize seek time
- Minimize space usage

Minimize Disk Access: Caching



- Buffer cache: Cache disk blocks in RAM
- Page cache: Cache VFS pages (before going to driver)
  - Often same data as buffer cache, so OS may merge them

Cache Management

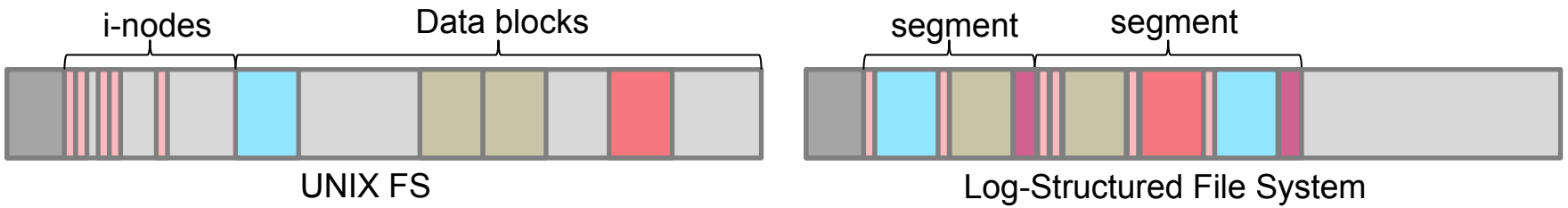


- Caches have LRU semantics adapted to:
  - Blocks with poor temporal locality
  - Critical blocks for file system consistency
- Write-through caching vs. periodic syncing
- Disk read-ahead: read blocks that may be used soon

Quiz

Why is yanking out a USB stick considered “unsafe”?

Minimize Disk Access: Log-structure File Systems



- Idea: Optimize for frequent small writes
- Collect pending writes in a log segment with i-nodes, entries, blocks
- Segments are often flushed to disk and can be large (e.g., 1MB)
- i-node map to find i-nodes in the log
- Garbage collection to reclaim old log entries

Similar Techniques Used in Flash-Based File Systems

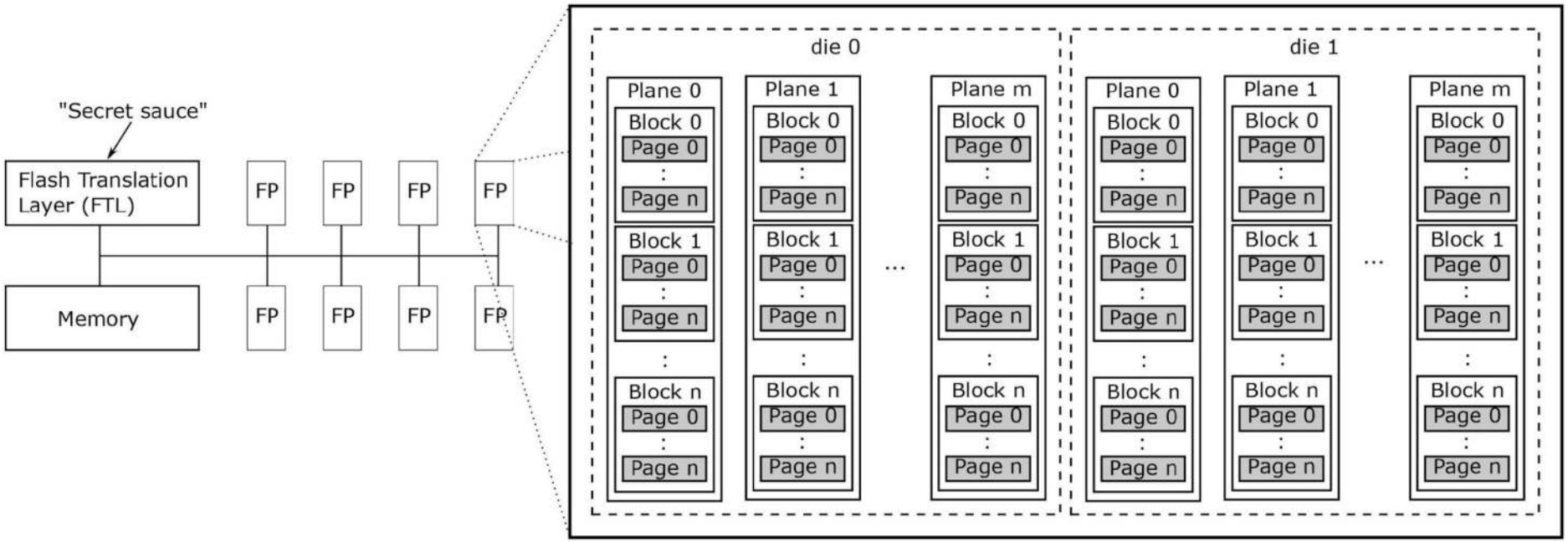


Figure 4-20. Components inside a typical flash SSD.

Minimize Seek Time: HDD Layout Management

- Not an issue for modern SSDs
  - Seek time is constant
- Important for traditional HDDs
- Different strategies to minimize HDD seek time:
  - Try to allocate files contiguously
  - Defragment disk
  - Store small file data “inline” in i-node
  - Spread i-nodes over the disk rather than at the start

Spread i-nodes Over the Disk

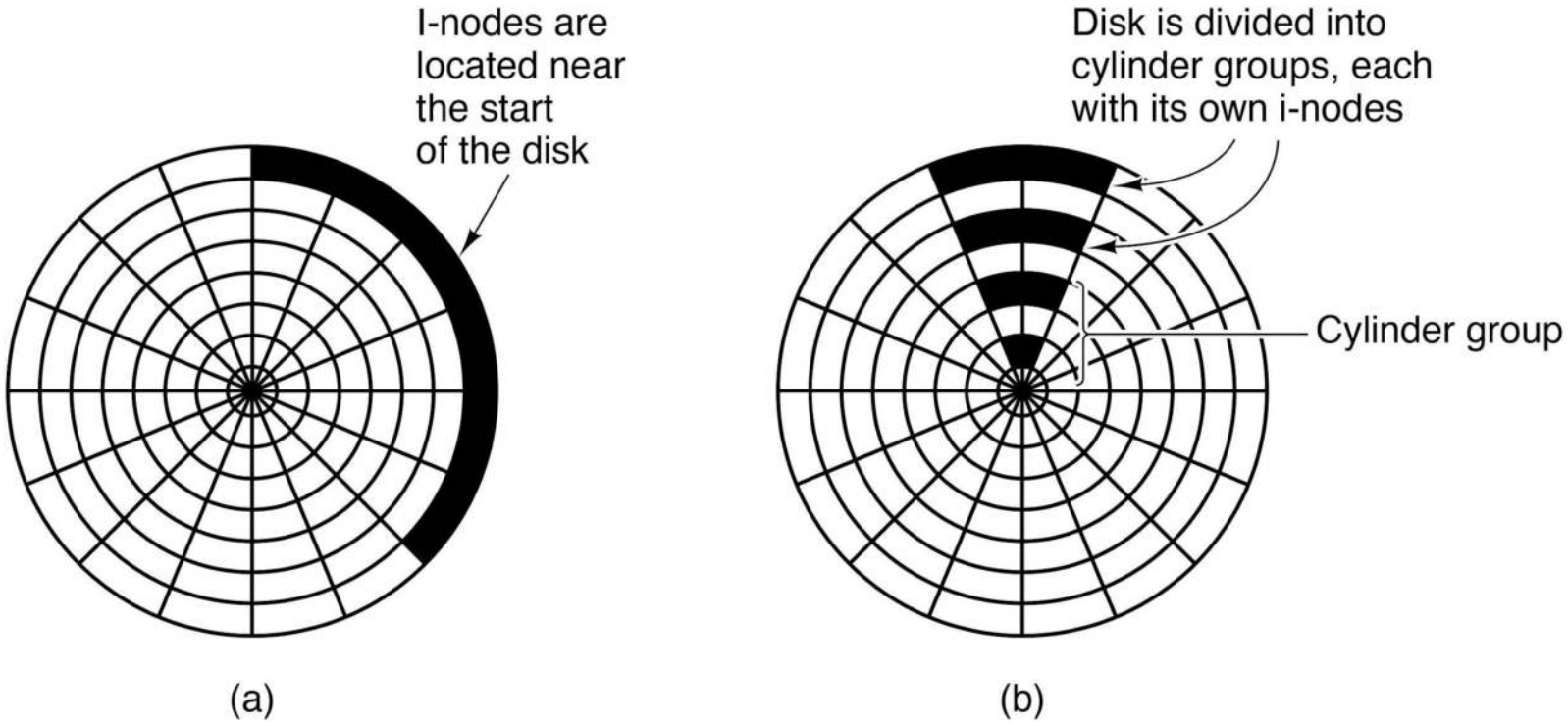


Figure 4-31. (a) I-nodes placed at the start of the disk. (b) Disk divided into cylinder groups, each with its own blocks and i-nodes.

Minimize Space Usage: Compression and Deduplication

- Compression: simplest method to use scarce storage more efficiently
- Can use a files system that compresses automatically
  - NTFS (Windows), Brtfs (Linux), ZFS (various)
- Can remove redundancy at file level or across files
- Deduplication: eliminate duplicate file copies
  - File level, portions of files, individual disk blocks
  - Inline using hashing
  - Or asynchronously in the background

File System Implementation

- How to store files?
- How to implement directories?
- How to manage disk space?
- How to ensure file system performance?
- **How to ensure file system dependability?**

File System Dependability Threats

- Disk failures:
  - Bad blocks
  - Whole-disk errors
- Power failures:
  - (Meta)-data inconsistently written to disk
- Software bugs:
  - Bad (meta)-data written to disk
- User errors:
  - `rm *.o  vs.  rm * .o`
- Lost or stolen computer

File System Backups

Backups to disk are generally made to handle one of two potential problems:

1. Recover from disaster.
2. Recover from stupidity.

Backups: Properties

- Incremental vs. full
- Online vs. offline
- Physical vs. logical
- Compressed vs. uncompressed
- Local vs. remote

Incremental Logical Backup (1 of 2)

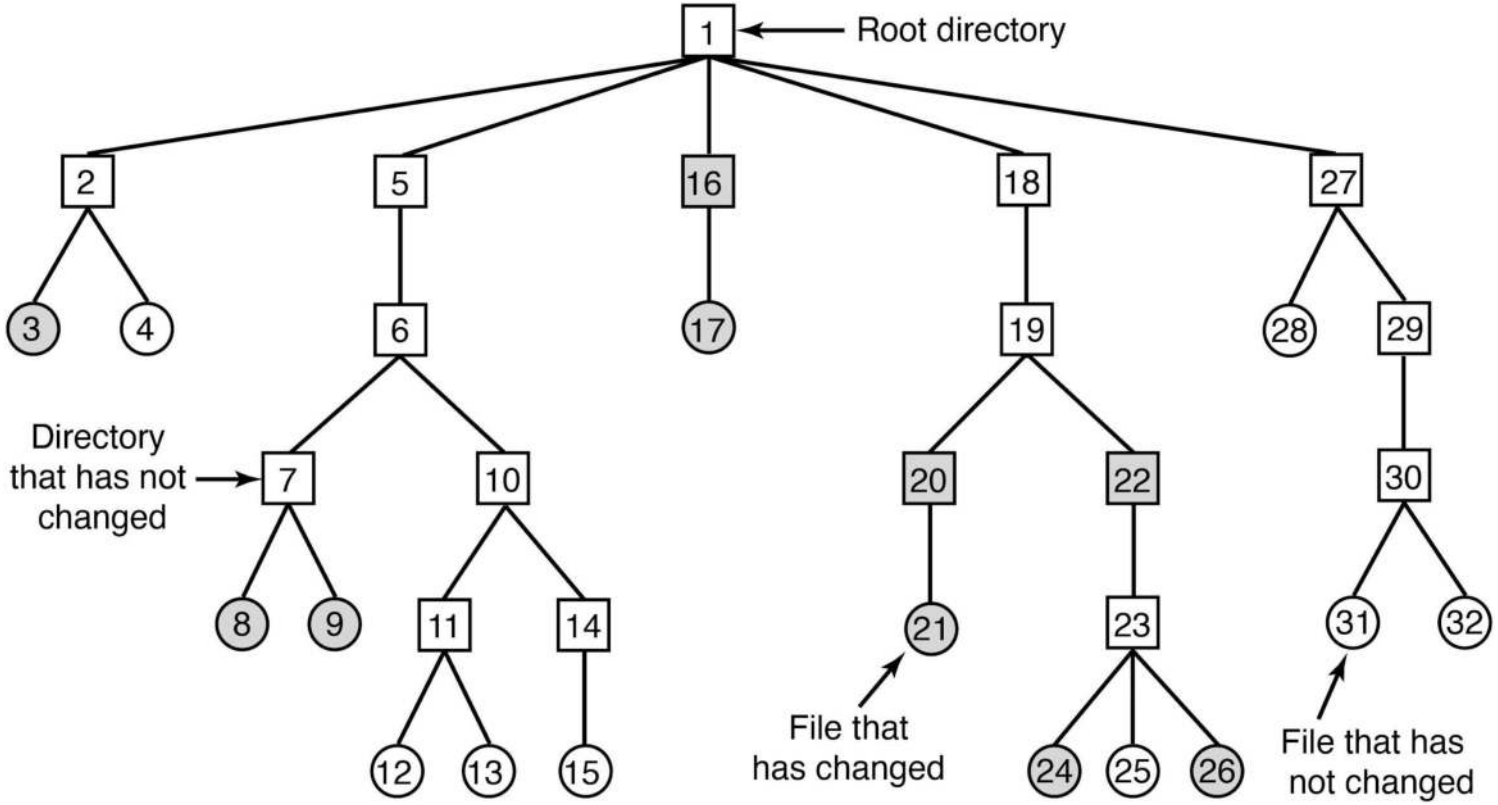


Figure 4-27. A file system to be dumped. The squares are directories and the circles are files. The shaded items have been modified since the last dump. Each directory and file is labeled by its i-node number.

Incremental Logical Backup (2 of 2)

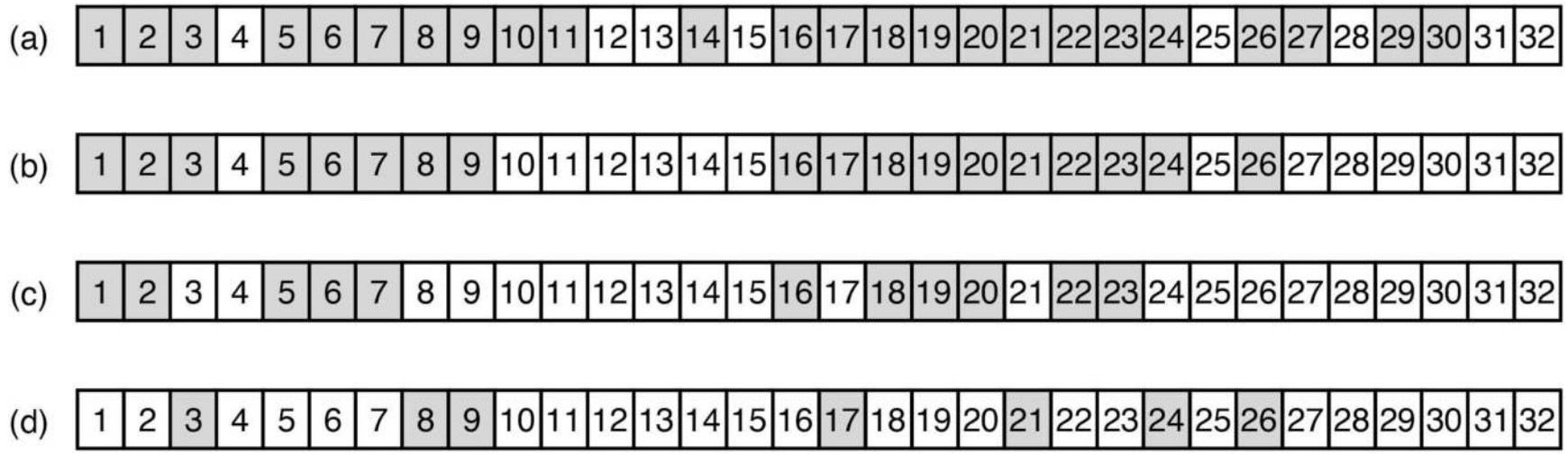


Figure 4-28. Bitmaps used by the logical dumping algorithm.

Stable Storage (1 of 2)

- Uses pair of identical disks
- Either can be read to get same results
- Operations defined to this end:
  1. Stable Writes
  2. Stable Reads
  3. Crash recovery

Stable Storage (2 of 2)

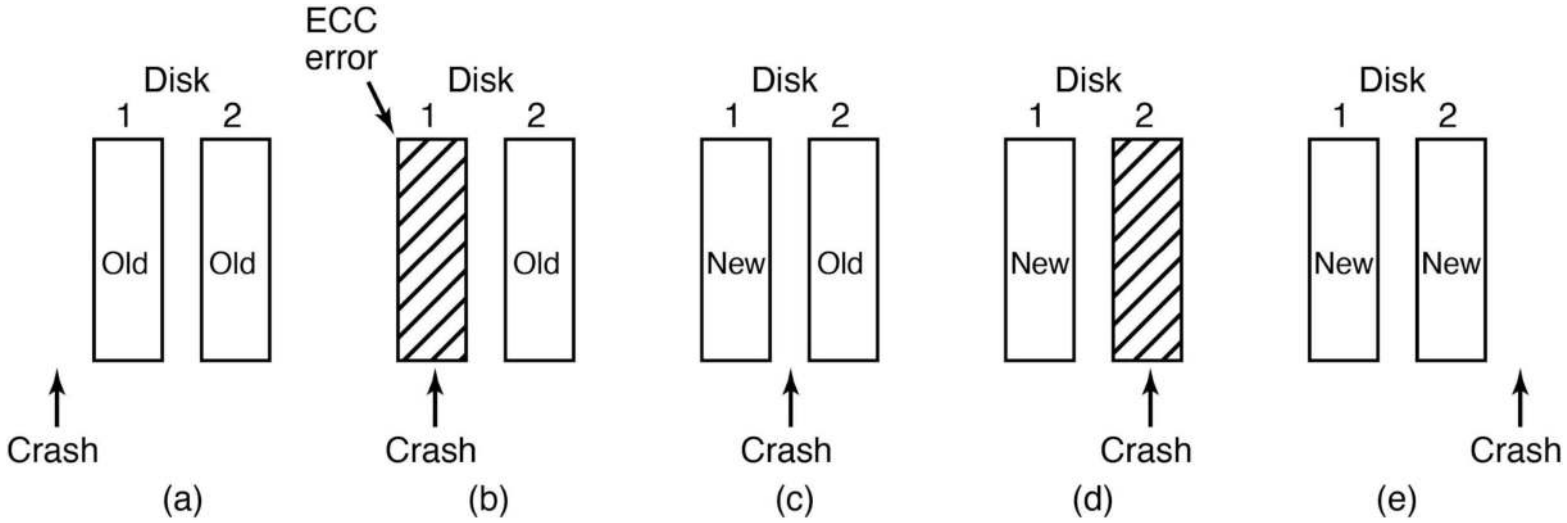
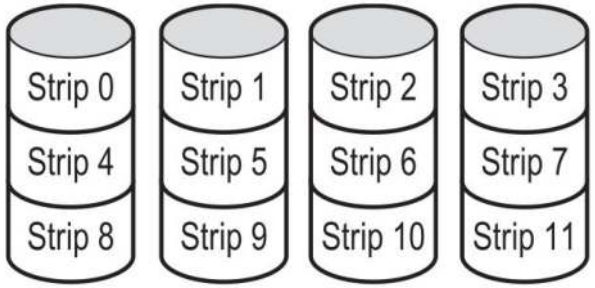
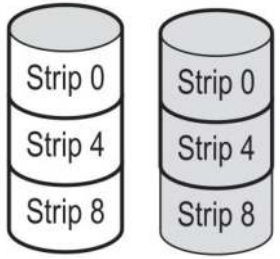


Figure 5-25. Analysis of the influence of crashes on stable writes.

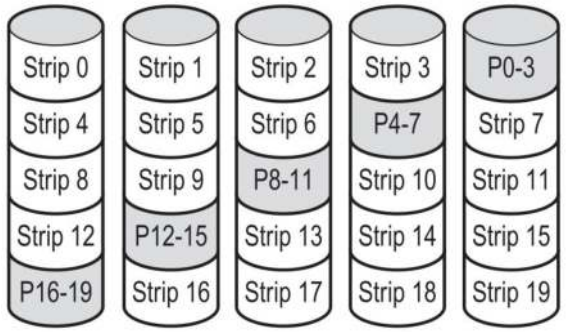
RAID: Redundant Array of Independent Disks



RAID 0



RAID 1



RAID 5

- **RAID 0 & 2**: Expand file system over multiple disks (+storage)
- **RAID 1**: Duplicate file system over disks (+redundancy)
- **RAID 3-6**: Store parity information (+storage, +redundancy)

RAID (1 of 2)

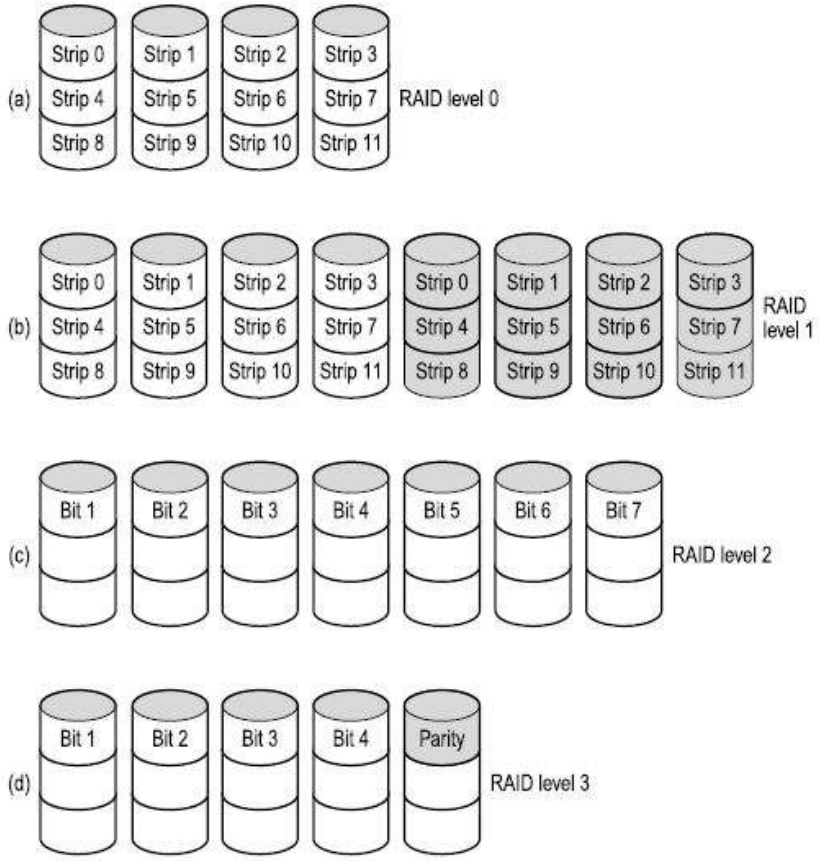


Figure 5-26. RAID levels 0 through 3. Backup and parity drives are shown shaded.

RAID (2 of 2)

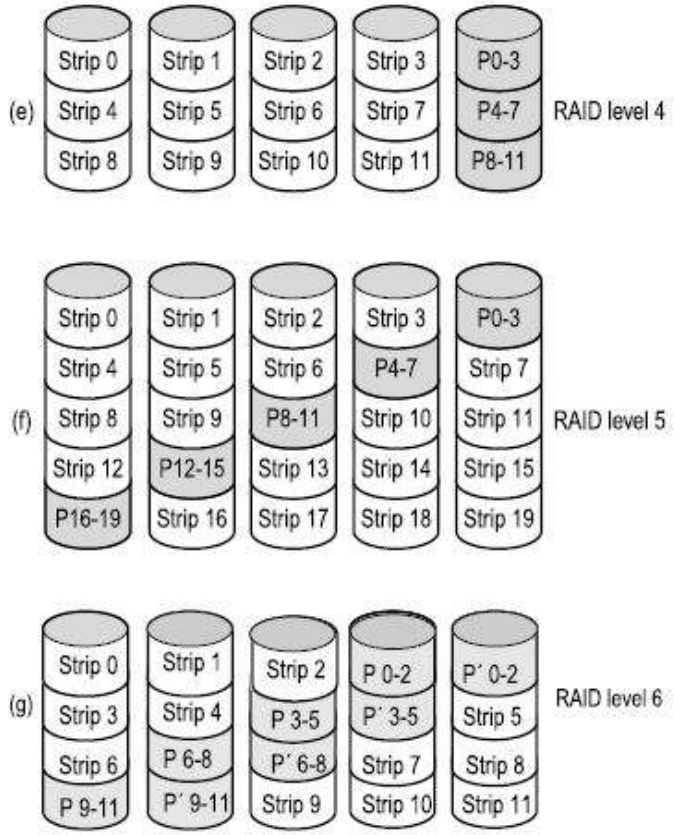


Figure 5-26. RAID levels 4 through 6. Backup and parity drives are shown shaded.

Quiz

Does RAID help with the following threats:

- *Physical failure (disk stops working)?*
- *Computer catching fire or getting stolen?*
- *Power failure (inconsistent write)?*
- *Bug in driver?*
- *User error?*

File System Consistency Check

Programs like `fsck` and `chkdsk` try to find (and fix) consistency errors in file system metadata

- Corrupt values
- Used blocks also marked as free
- Blocks not marked as free nor used
- Blocks being used multiple times
- ...

File System Consistency

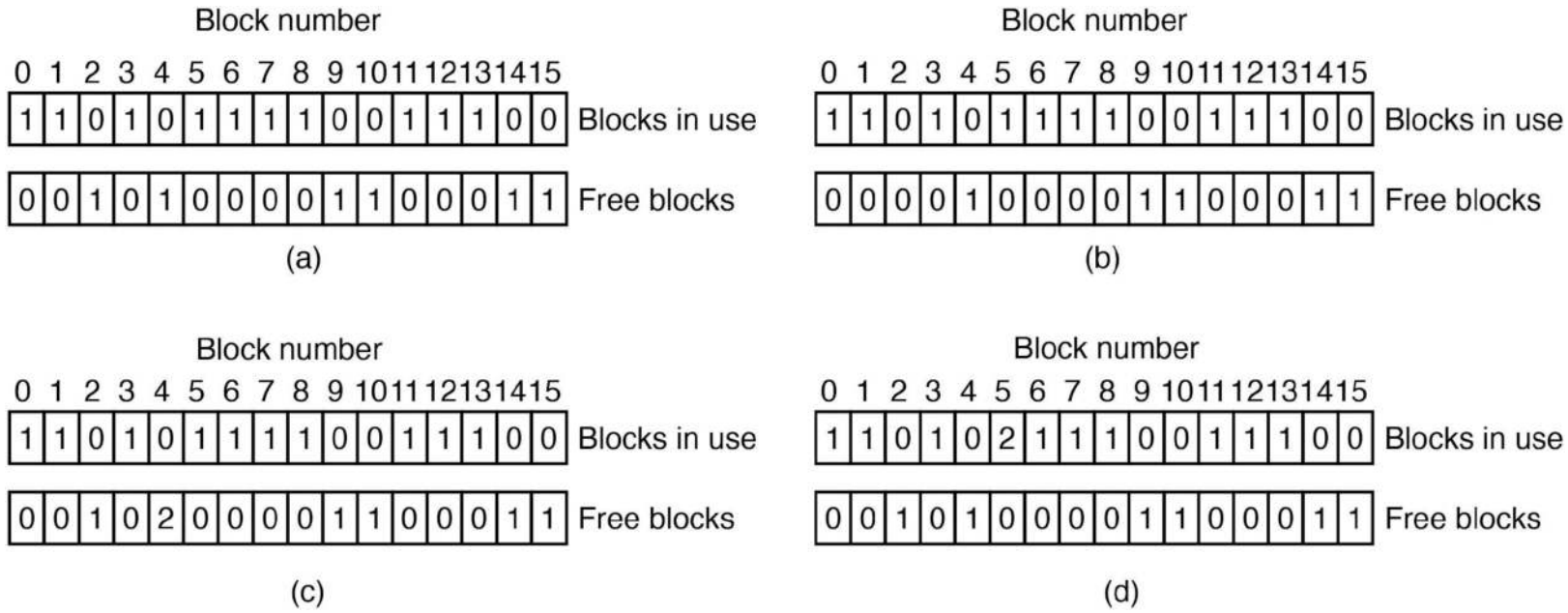


Figure 4-29. File system states. (a) Consistent. (b) Missing block. (c) Duplicate block in free list. (d) Duplicate data block.

Journaling File Systems

- Idea: Use “logs” for crash recovery
- First write transactional operations in log
  - E.g., removing a file:
    - Remove file from its directory
    - Release i-node to the pool of free i-nodes
    - Return all disk blocks to pool of free blocks
- After crash, replay operations from log
- Single operations need to be idempotent
- Should support multiple, arbitrary crashes
- Journaling widely used in modern FSes (e.g., Ext4, NTFS)

Secure File Detection and Disk Encryption

- Securely deleting data on a disk is not easy
  - Can physically destroy the disk
  - Overwriting with zeros might not be sufficient
  - Multiple overwrites (3-7 times)
- Best approach is to encrypt
  - Self-encrypting drives (SEDs)
    - Not foolproof!
  - Advanced Encryption Standard (AES)
    - Many users are unaware that their data are encrypted

Copyright