one, two, or three buttons, depending on the designers' estimate of the users' intellectual ability to keep track of more than one button.

The message to the computer contains three items: Δx, Δy, buttons. The first item is the change in x position since the last message. Then comes the change in y position since the last message. Finally, the status of the buttons is included. The format of the message depends on the system and the number of buttons the mouse has. Usually, it takes 3 bytes. Most mice report back a maximum of 40 times/sec, so the mouse may have moved multiple mickeys since the last report.

Note that the mouse only indicates changes in position, not absolute position itself. If the mouse is picked up and put down gently without causing the ball to rotate, no messages will be sent.

Some GUIs distinguish between single clicks and double clicks of a mouse button. If two clicks are close enough in space (mickeys) and also close enough in time (milliseconds), a double click is signaled. The maximum for "close enough" is up to the software, with both parameters usually being user settable.

That is enough about the input hardware; let us now turn to the display hardware. Display devices can be divided into two categories. **Vector graphics** devices can accept and carry out commands such as drawing points, lines, geometric figures, and text. In contrast, **raster graphics** devices represent the output area as a rectangular grid of points called **pixels**, each of which has some grayscale value or some color. In the early days of computing, vector graphics devices were common, but now plotters are the only vector graphics devices around. Everything else uses raster graphics, sometimes called **bitmap graphics**.

Raster graphics displays are implemented by a hardware device called a **graphics adapter**. A graphics adapter contains a special memory called a **video RAM**, which forms part of the computer's address space and is addressed by the CPU the same way as the rest of memory (see Fig. 5-38). The screen image is stored here in either character mode or bit mode. In character mode, each byte (or 2 bytes) of video RAM contains one character to be displayed. In bitmap mode, each pixel on the screen is represented separately in the video RAM, with 1 bit per pixel for the simplest black and white display to 24 or more bits per pixel for a high quality color display.

Also part of the graphics adapter is a chip called a **video controller**. This chip pulls characters or bits out of the video RAM and generates the video signal used to drive the monitor. A monitor generates a beam of electrons that scans horizontally across the screen, painting lines on it. Typically the screen has 480 to 1024 lines from top to bottom, with 640 to 1200 pixels per line. The video controller signal modulates the electron beam, determining whether a given pixel will be light or dark. Color monitors have three beams, for red, green, and blue, which are independently modulated. Flat panel displays also use pixels in three colors, but how these displays work in detail is beyond the scope of this book.

Video controllers have two modes: character mode (used for simple text) and bit mode (for everything else). In character mode, the controller might fit each

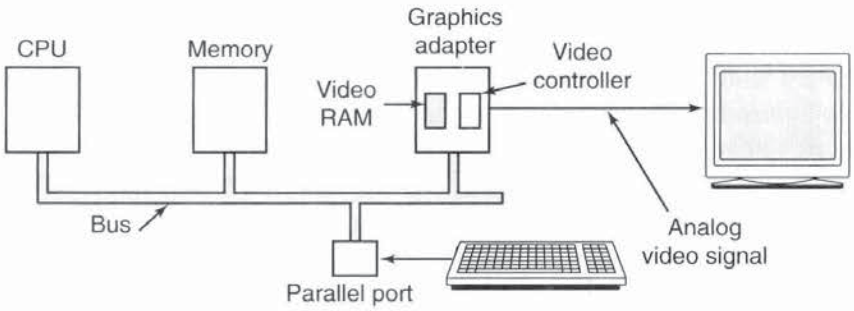


Figure 5-38. With memory-mapped displays, the driver writes directly into the display's video RAM.

character in a box 9 pixels wide by 14 pixels high (including the space between characters), and have 25 lines of 80 characters. The display would then have 350 scan lines of 720 pixels each. Each of these frames is redrawn 60 to 100 times a second to avoid flicker.

To display the text on the screen, the video controller might fetch the first 80 characters from the video RAM, generate 14 scan lines, fetch the next 80 characters from the video RAM, generate the following 14 scan lines, and so on. Alternatively, it could fetch each character once per scan line to eliminate the need for buffering in the controller. The 9-by-14 bit patterns for the characters are kept in a ROM used by the video controller. (RAM may also be used to support custom fonts.) The ROM is addressed by a 12-bit address, 8 bits from the character code and 4 bits to specify a scan line. The 8 bits in each byte of the ROM control 8 pixels; the 9th pixel between characters is always blank. Thus $14 \times 80 = 1120$ memory references to the video RAM are needed per line of text on the screen. The same number of references are made to the character generator ROM.

In Fig. 5-39(a) we see a portion of the video RAM for a display operating in character mode. Each character on the screen of Fig. 5-39(b) occupies two bytes in the RAM. The low-order character is the ASCII code for the character to be displayed. The high-order character is the attribute byte, which is used to specify the color, reverse video, blinking, and so on. A screen of 25 by 80 characters requires 4000 bytes of video RAM in this mode.

Operating in bitmap mode uses the same principle, except that each pixel on the screen is individually controlled and individually represented by 1 or more bits in the video RAM. In the simplest configuration for a monochrome display, each screen pixel has a corresponding bit in the video RAM. At the other extreme, each screen pixel is represented by a 24-bit number in the video RAM, with 8 bits each for the Red, Green, and Blue intensities. This RGB representation is used because red, green, and blue are the primary additive colors, from which all other colors can be constructed by summing various intensities of these colors.

Screen sizes vary, the most common being 640×480 (VGA), 800×600 (SVGA), 1024×768 (XGA), 1280×1024, and 1600×1200. All of these except

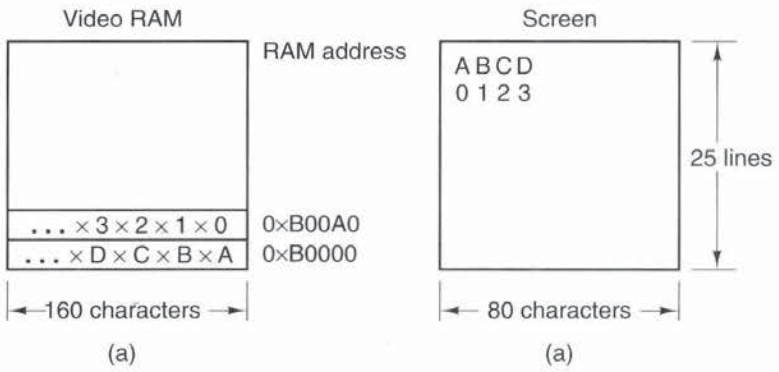


Figure 5-39. (a) A video RAM image for a simple monochrome display in character mode. (b) The corresponding screen. The ×s are attribute bytes.

1280 × 1024 are in the ratio of 4:3, which fits the aspect ratio of NTSC television sets and thus gives square pixels. 1280 × 1024 should have been 1280 × 960, but the lure of 1024 was apparently to great too resist, even though it distorts the pixels slightly and makes scaling to and from the other sizes harder. As an example, a 768 × 1024 color display with 24 bits per pixel requires 2.25 MB of RAM just to hold the image. If the full screen is refreshed 75 times/sec, the video RAM must be capable of delivering data continuously at 169 MB/sec.

To avoid having to manage such large screen images, some systems have the ability to trade off color resolution against image size. In the simplest scheme, each pixel is represented by an 8-bit number. Rather than indicating a color, this value is an index into a table of 256 entries, each holding a 24-bit (red, green, blue) value. This table, called a **color palette** and often stored in hardware, allows the screen to contain an arbitrary 256 colors at any instant. Changing, say, entry 7 in the color palette, changes the color of all the pixels in the image with a 7 as value. Using an 8-bit color palette reduces the amount of space needed to store the screen image from 3 bytes per pixel to 1 byte per pixel. The price paid is coarser color resolution. The GIF image compression scheme also works with a color palette like this.

It is also possible to use a color palette with 16 bits per pixel. In this case the color palette contains 65,536 entries so that up to 65,536 colors can be used at once. However, the saving in space is less since each pixel now requires 2 bytes in the video RAM. Also, if the color palette is kept in hardware (to avoid an expensive memory lookup on every pixel), more dedicated hardware is needed to store the color palette.

It is also possible to handle 16-bit color by storing the RGB values as three 5-bit numbers, with 1 bit left over (or to give green 6 bits, since the eye is more sensitive to green than to red or blue). In effect, this system is the same as 24-bit color, except with fewer shades of each color available.

5.7.2 Input Software

Once the keyboard driver has received the character, it must begin processing it. Since the keyboard delivers key numbers rather than the character codes used by application software, the driver must convert between the codes by using a table. Not all IBM "compatibles" use standard key numbering, so if the driver wants to support these machines, it must map different keyboards with different tables. A simple approach is to compile a table that maps between the codes provided by the keyboard and ASCII codes into the keyboard driver, but this is unsatisfactory for users of languages other than English. Keyboards are arranged differently in different countries, and the ASCII character set is not adequate even for the majority of people in the Western Hemisphere, where speakers of Spanish, Portuguese, and French need accented characters and punctuation marks not used in English.

To respond to the need for flexibility of keyboard layouts to provide for different languages, many operating systems provide for loadable **keymaps** or **code pages**, which make it possible to choose the mapping between keyboard codes and codes delivered to the application, either when the system is booted or later.

5.7.3 Output Software for Windows

Output software for GUIs is a massive topic. Many 1500-page books have been written about the Windows GUI alone (e.g., Petzold, 1999; Simon, 1997; and Rector and Newcomer, 1997). Clearly, in this section, we can only scratch the surface and present a few of the underlying concepts. To make the discussion concrete, we will describe the Win32 API, which is supported by all 32-bit versions of Windows. The output software for other GUIs is roughly comparable in a general sense, but the details are very different.

The basic item on the screen is a rectangular area called a **window**. A window's position and size are uniquely determined by giving the coordinates (in pixels) of two diagonally opposite corners. A window may contain a title bar, a menu bar, a tool bar, a vertical scroll bar, and a horizontal scroll bar. A typical window is shown in Fig. 5-40. Note that the Windows coordinate system puts the origin in the upper lefthand corner and has y increase downward, which is different from the Cartesian coordinates used in mathematics.

When a window is created, the parameters specify whether the window can be moved by the user, resized by the user, or scrolled (by dragging the thumb on the scroll bar) by the user. The main window produced by most programs can be moved, resized, and scrolled, which has enormous consequences for the way Windows programs are written. In particular, programs must be informed about changes to the size of their windows and must be prepared to redraw the contents of their windows at any time, even when they least expect it.

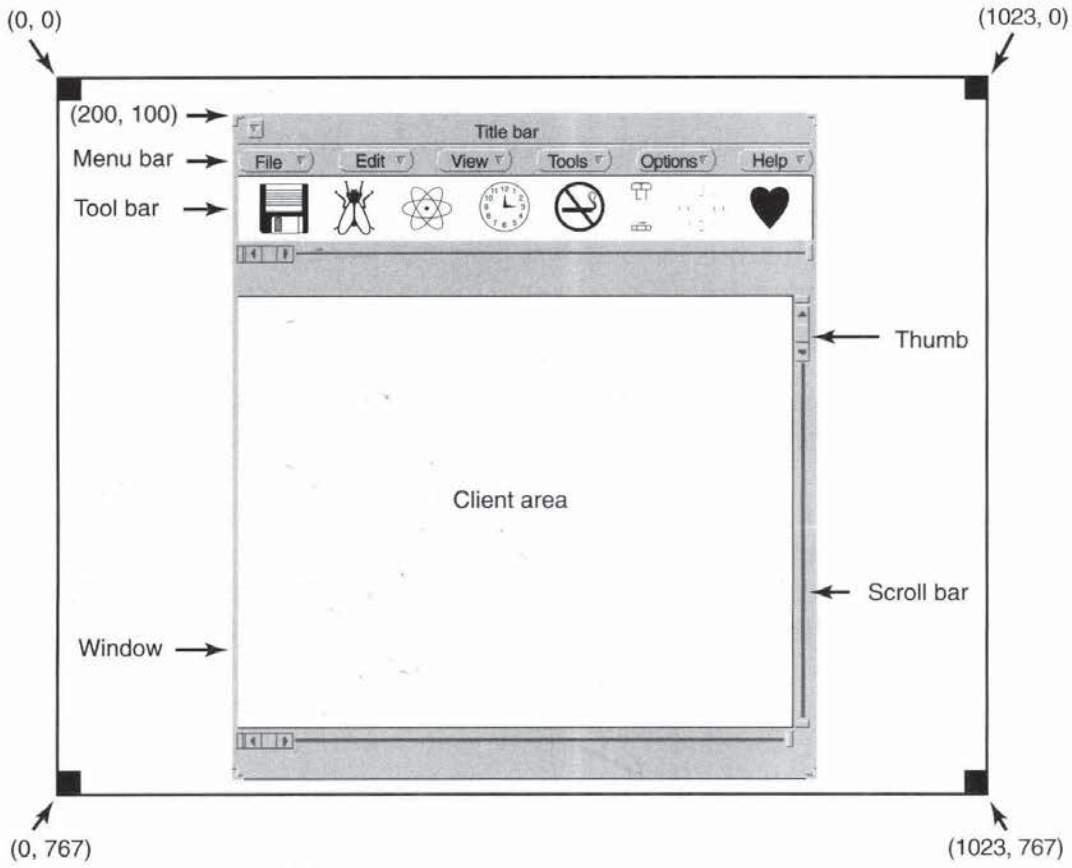


Figure 5-40. A sample window located at (200, 100) on an XGA display.

As a consequence, Windows programs are message oriented. User actions involving the keyboard or mouse are captured by Windows and converted into messages to the program owning the window being addressed. Each program has a message queue to which messages relating to all its windows are sent. The main loop of the program consists of fishing out the next message and processing it by calling an internal procedure for that message type. In some cases, Windows itself may call these procedures directly, bypassing the message queue. This model is quite different than the UNIX model of procedural code that makes system calls to interact with the operating system.

To make this programming model clearer, consider the example of Fig. 5-41. Here we see the skeleton of a main program for Windows. It is not complete and does no error checking, but it shows enough detail for our purposes. It starts by including a header file, *windows.h*, which contains many macros, data types, constants, function prototypes, and other information needed by Windows programs.

The main program starts with a declaration giving its name and parameters. The *WINAPI* macro is an instruction to the compiler to use a certain parameter

```
#include <windows.h>

int WINAPI WinMain(HINSTANCE h, HINSTANCE, hprev, char *szCmd, int iCmdShow)
{
    WNDCLASS wndclass;              /* class object for this window */
    MSG msg;                        /* incoming messages are stored here */
    HWND hwnd;                      /* handle (pointer) to the window object */

    /* Initialize wndclass */
    wndclass.lpfnWndProc = WndProc;          /* tells which procedure to call */
    wndclass.lpszClassName = "Program name"; /* Text for title bar */
    wndclass.hIcon = LoadIcon(NULL, IDI_APPLICATION);   /* load program icon */
    wndclass.hCursor = LoadCursor(NULL, IDC_ARROW);     /* load mouse cursor */

    RegisterClass(&wndclass);       /* tell Windows about wndclass */
    hwnd = CreateWindow ( ... )     /* allocate storage for the window */
    ShowWindow(hwnd, iCmdShow); /* display the window on the screen */
    UpdateWindow(hwnd);             /* tell the window to paint itself */

    while (GetMessage(&msg, NULL, 0, 0)) {    /* get message from queue */
        TranslateMessage(&msg);   /* translate the message */
        DispatchMessage(&msg);    /* send msg to the appropriate procedure */
    }
    return(msg.wParam);
}

long CALLBACK WndProc(HWND hwnd, UINT message, UINT wParam, long lParam)
{
    /* Declarations go here. */

    switch (message) {
        case WM_CREATE:    ... ;   return ... ;    /* create window */
        case WM_PAINT:     ... ;   return ... ;    /* repaint contents of window */
        case WM_DESTROY:   ... ;   return ... ;    /* destroy window */
    }
    return(DefWindowProc(hwnd, message, wParam, lParam));/* default */
}
```

Figure 5-41. A skeleton of a Windows main program.

passing convention and will not be of further concern to us. The first parameter, *h*, is an instance handle and is used to identify the program to the rest of the system. To some extent, Win32 is object oriented, which means that the system contains objects (e.g., programs, files, and windows) that have some state and associated code, called **methods**, that operate on that state. Objects are referred to using

handles, and in this case, *h* identifies the program. The second parameter is present only for reasons of backward compatibility. It is no longer used. The third parameter, *szCmd*, is a zero-terminated string containing the command line that started the program, even if it was not started from a command line. The fourth parameter, *iCmdShow*, tells whether the program's initial window should occupy the entire screen, part of the screen, or none of the screen (task bar only).

This declaration illustrates a widely used Microsoft convention called **Hungarian notation**. The name is a pun on Polish notation, the postfix system invented by the Polish logician J. Lukasiewicz for representing algebraic formulas without using precedence or parentheses. Hungarian notation was invented by a Hungarian programmer at Microsoft, Charles Simonyi, and uses the first few characters of an identifier to specify the type. The allowed letters and types include c (character), w (word, now meaning an unsigned 16-bit integer), i (32-bit signed integer), l (long, also a 32-bit signed integer), s (string), sz (string terminated by a zero byte), p (pointer), fn (function), and h (handle). Thus *szCmd* is a zero-terminated string and *iCmdShow* is an integer, for example. Many programmers believe that encoding the type in variable names this way has little value and makes Windows code exceptionally hard to read. Nothing analogous to this convention is present in UNIX.

Every window must have an associated class object that defines its properties. In Fig. 5-41, that class object is *wndclass*. An object of type *WNDCLASS* has 10 fields, four of which are initialized in Fig. 5-41. In an actual program, the other six would be initialized as well. The most important field is *lpfnWndProc*, which is a long (i.e., 32-bit) pointer to the function that handles the messages directed to this window. The other fields initialized here tell which name and icon to use in the title bar, and which symbol to use for the mouse cursor.

After *wndclass* has been initialized, *RegisterClass* is called to pass it to Windows. In particular, after this call Windows knows which procedure to call when various events occur that do not go through the message queue. The next call, *CreateWindow*, allocates memory for the window's data structure and returns a handle for referencing it later. The program then makes two more calls in a row, to put the window's outline on the screen, and finally fill it in completely.

At this point we come to the program's main loop, which consists of getting a message, having certain translations done to it, and then passing it back to Windows to have Windows invoke *WndProc* to process it. To answer the question of whether this whole mechanism could have been made simpler, the answer is yes, but it was done this way for historical reasons and we are now stuck with it.

Following the main program is the procedure **WndProc**, which handles the various messages that can be sent to the window. The use of *CALLBACK* here, like *WINAPI* above, specifies the calling sequence to use for parameters. The first parameter is the handle of the window to use. The second parameter is the message type. The third and fourth parameters can be used to provide additional information when needed.

Message types *WM_CREATE* and *WM_DESTROY* are sent at the start and end of the program, respectively. They give the program the opportunity, for example, to allocate memory for data structures and then return it.

The third message type, *WM_PAINT*, is an instruction to the program to fill in the window. It is not only called when the window is first drawn, but often during program execution as well. In contrast to text-based systems, in Windows a program cannot assume that whatever it draws on the screen will stay there until it removes it. Other windows can be dragged on top of this one, menus can be pulled down over it, dialog boxes and tool tips can cover part of it, and so on. When these items are removed, the window has to be redrawn. The way Windows tells a program to redraw a window is to send it a *WM_PAINT* message. As a friendly gesture, it also provides information about what part of the window has been overwritten, in case it is easier to regenerate that part of the window instead of redrawing the whole thing.

There are two ways Windows can get a program to do something. One way is to post a message to its message queue. This method is used for keyboard input, mouse input, and timers that have expired. The other way, sending a message to the window, involves having Windows directly call *WndProc* itself. This method is used for all other events. Since Windows is notified when a message is fully processed, it can refrain from making a new call until the previous one is finished. In this way race conditions are avoided.

There are many more message types. To avoid erratic behavior should an unexpected message arrive, it is best to call *DefWindowProc* at the end of *WndProc* to let the default handler take care of the other cases.

In summary, a Windows program normally creates one or more windows with a class object for each one. Associated with each program is a message queue and a set of handler procedures. Ultimately, the program's behavior is driven by the incoming events, which are processed by the handler procedures. This is a very different model of the world than the more procedural view that UNIX takes.

The actual drawing to the screen is handled by a package consisting of hundreds of procedures that are bundled together to form the **GDI (Graphics Device Interface)**. It can handle text and all kinds of graphics and is designed to be platform and device independent. Before a program can draw (i.e., paint) in a window, it needs to acquire a **device context** which is an internal data structure containing properties of the window, such as the current font, text color, background color, etc. Most GDI calls use the device context, either for drawing or for getting or setting the properties.

Various ways exist to acquire the device context exist. A simple example of its acquisition and use is

```
hdc = GetDC(hwnd);
TextOut(hdc, x, y, psText, iLength);
ReleaseDC(hwnd, hdc);
```

The first statement gets a handle to a device content, *hdc*. The second one uses the device context to write a line of text on the screen, specifying the (*x*, *y*) coordinates of where the string starts, a pointer to the string itself, and its length. The third call releases the device context to indicate that the program is through drawing for the moment. Note that *hdc* is used in a way analogous to a UNIX file descriptor. Also note that *ReleaseDC* contains redundant information (the use of *hdc* uniquely specifies a window). The use of redundant information that has no actual value is common in Windows.

Another interesting note is that when *hdc* is acquired in this way, the program can only write in the client area of the window, not in the title bar and other parts of it. Internally, in the device context's data structure, a clipping region is maintained. Any drawing outside the clipping region is ignored. However, there is another way to acquire a device context, *GetWindowDC*, which sets the clipping region to the entire window. Other calls restrict the clipping region in other ways. Having multiple calls that do almost the same thing is another characteristic of Windows.

A complete treatment of the GDI is out of the question here. For the interested reader, the references cited above provide additional information. Nevertheless, a few words about the GDI are probably worthwhile given how important it is. GDI has various procedure calls to get and release device contexts, obtain information about device contexts, get and set device context attributes (e.g., the background color), manipulate GDI objects such as pens, brushes, and fonts, each of which has its own attributes. Finally, of course, there are a large number of GDI calls to actually draw on the screen.

The drawing procedures fall into four categories: drawing lines and curves, drawing filled areas, managing bitmaps, and displaying text. We saw an example of drawing text above, so let us take a quick look at one of the others. The call

```
Rectangle(hdc, xleft, ytop, xright, ybottom);
```

draws a filled rectangle whose corners are (*xleft*, *ytop*) and (*xright*, *ybottom*). For example,

```
Rectangle(hdc, 2, 1, 6, 4);
```

will draw the rectangle shown in Fig. 5-42. The line width and color and fill color are taken from the device context. Other GDI calls are similar in flavor.

BitMaps

The GDI procedures are examples of vector graphics. They are used to place geometric figures and text on the screen. They can be scaled easily to larger or smaller screens (provided the number of pixels on the screen is the same). They are also relatively device independent. A collection of calls to GDI procedures can be assembled in a file that can describe a complex drawing. Such a file is

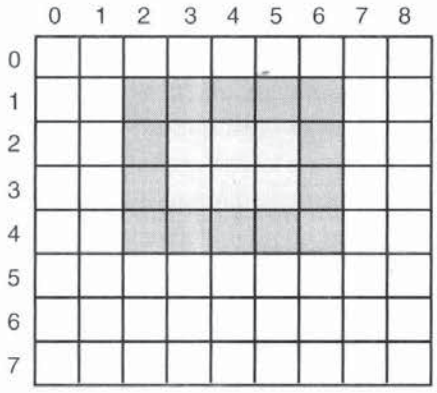


Figure 5-42. An example rectangle drawn using *Rectangle*. Each box represents one pixel.

called a Windows **metafile**, and is widely used to transmit drawings from one Windows program to another. Such files have extension *.wmf*.

Many Windows programs allow the user to copy (part of) a drawing and put in on the Windows clipboard. The user can then go to another program and paste the contents of the clipboard into another document. One way of doing this is for the first program to represent the drawing as a Windows metafile and put it on the clipboard in *.wmf* format. Other ways also exist.

Not all the images that computers manipulate can be generated using vector graphics. Photographs and videos, for example, do not use vector graphics. Instead, these items are scanned in by overlaying a grid on the image. The average red, green, and blue values of each grid square are then sampled and saved as the value of one pixel. Such a file is called a **bitmap**. There are extensive facilities in Windows for manipulating bitmaps.

Another use for bitmaps is for text. One way to represent a particular character in some font is as a small bitmap. Adding text to the screen then becomes a matter of moving bitmaps.

One general way to use bitmaps is through a procedure called *bitblt*. It is called as follows:

```
bitblt(dsthdc, dx, dy, wid, ht, srchdc, sx, sy, rasterop);
```

In its simplest form, it copies a bitmap from a rectangle in one window to a rectangle in another window (or the same one). The first three parameters specify the destination window and position. Then come the width and height. Next come the source window and position. Note that each window has its own coordinate system, with (0, 0) in the upper left-hand corner of the window. The last parameter will be described below. The effect of

```
BitBlt(hdc2, 1, 2, 5, 7, hdc1, 2, 2, SRCCOPY);
```

is shown in Fig. 5-43. Notice carefully that the entire 5×7 area of the letter A has been copied, including the background color.

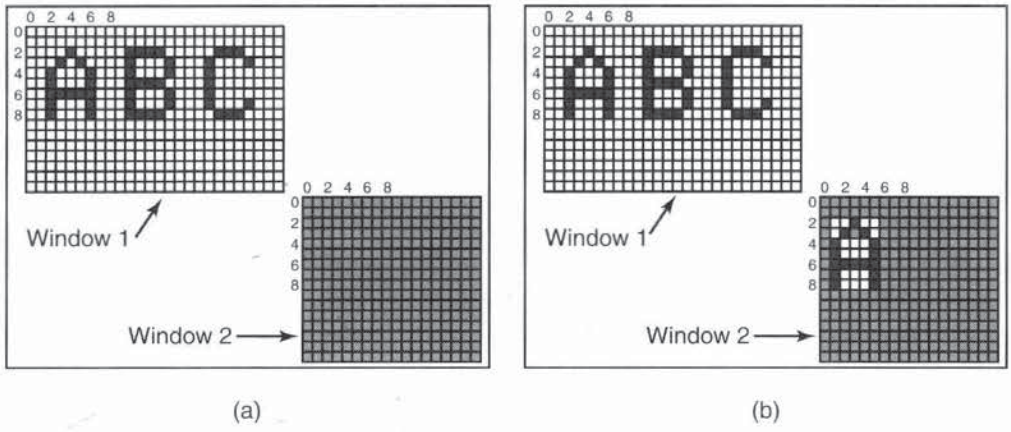


Figure 5-43. Copying bitmaps using *BitBlt*. (a) Before. (b) After.

BitBlt can do more than just copy bitmaps. The last parameter gives the possibility of performing Boolean operations to combine the source bitmap and the destination bitmap. For example, the source can be ORed into the destination to merge with it. It can also be EXCLUSIVE ORed into it, which maintains the characteristics of both source and destination.

A problem with bitmaps is that they do not scale. A character that is in a box of 8×12 on a display of 640×480 will look reasonable. However, if this bitmap is copied to a printed page at 1200 dots/inch, which is 10200 bits $\times$ 13200 bits, the character width (8 pixels) will be 8/1200 inch or 0.17 mm wide. In addition, copying between devices with different color properties or between monochrome and color does not work well.

For this reason, Windows also supports a data structure called a **DIB** (**Device Independent Bitmap**). Files using this format use the extension *.bmp*. These files have file and information headers and a color table before the pixels. This information makes it easier to move bitmaps between dissimilar devices.

Fonts

In versions of Windows before 3.1. characters were represented as bitmaps and copied onto the screen or printer using *BitBlt*. The problem with that, as we just saw, is that a bitmap that makes sense on the screen is too small for the printer. Also, a different bitmap is needed for each character in each size. In other words, given the bitmap for A in 10-point type, there is no way to compute it for 12-point type. Because every character of every font might be needed for sizes ranging from 4 point to 120 point, a vast number of bitmaps were needed. The whole system was just too cumbersome for text.

The solution was the introduction of TrueType fonts, which are not bitmaps but outlines of the characters. Each TrueType character is defined by a sequence

of points around its perimeter. All the points are relative to the (0, 0) origin. Using this system, it is easy to scale the characters up or down. All that has to be done is to multiply each coordinate by the same scale factor. In this way, a True-Type character can be scaled up or down to any point size, even fractional point sizes. Once at the proper size, the points can be connected using the well-known follow-the-dots algorithm taught in kindergarten (note that modern kindergartens use splines for smoother results). After the outline has been completed, the character can be filled in. An example of some characters scaled to three different point sizes is given in Fig. 5-44.



Figure 5-44. Some examples of character outlines at different point sizes.

Once the filled character is available in mathematical form, it can be rasterized, that is, converted to a bitmap at whatever resolution is desired. By first scaling and then rasterizing, we can be sure that the characters displayed on the screen and those that appear on the printer will be as close as possible, differing only in quantization error. To improve the quality still more, it is possible to embed hints in each character telling how to do the rasterization. For example, both serifs on the top of the letter T should be identical, something that might not otherwise be the case due to roundoff error.

5.8 NETWORK TERMINALS

Network terminals are used to connect a remote user to computer over a network, either a local area network or a wide area network. There are two different philosophies of how network terminals should work. In one view, the terminal

should have a large amount of computing power and memory in order to run complex protocols to compress the amount of data sent over the network. (A **protocol** is a set of requests and responses that a sender and receiver agree upon in order to communicate over a network or other interface.) In the other view, the terminal should be extremely simple, basically displaying pixels and not doing much thinking in order to make it very cheap. In the following two sections we will discuss an example of each philosophy. First we will examine the sophisticated X Window System. Then we will look at the minimal SLIM terminal.

5.8.1 The X Window System

The ultimate in intelligent terminals is a terminal that contains a CPU as powerful as the main computer, along with megabytes of memory, a keyboard, and a mouse. One terminal of this type is the **X terminal**, which runs the X Window System (often just called X), developed at M.I.T. as part of project Athena. An X terminal is a computer that runs the X software and which interacts with programs running on a remote computer.

The program inside the X terminal that collects input from the keyboard or mouse and accepts commands from a remote computer is called the **X server**. It has to keep track of which window is currently selected (where the mouse pointer is), so it knows which client to send any new keyboard input to. It communicates over the network with **X clients** running on some remote host. It sends them keyboard and mouse input and accepts display commands from them.

It may seem strange to have the X server inside the terminal and the clients on the remote host, but the X server's job is to display bits, so it makes sense to be near the user. From the program's point of view, it is a client telling the server to do things, like display text and geometric figures. The server (in the terminal) just does what it is told, as do all servers. The arrangement of client and server is shown in Fig. 5-45.

It is also possible to run the X Window System on top of UNIX or another operating system. In fact, many UNIX systems run X as their standard windowing system, even on standalone machines or for accessing remote computers over the Internet. What the X Window System really defines is the protocol between the X client and the X server, as shown in Fig. 5-45. It does not matter whether the client and server are on the same machine, separated by 100 meters over a local area network, or are thousands of kilometers apart and connected by the Internet. The protocol and operation of the system is identical in all cases.

X is just a windowing system. It is not a complete GUI. To get a complete GUI, others layer of software are run on top of it. One layer is **Xlib**, which is a set of library procedures for accessing the X functionality. These procedures form the basis of the X Window System and are what we will examine below, but they are too primitive for most user programs to access directly. For example, each mouse click is reported separately, so that determining that two clicks really form a double click has to be handled above Xlib.

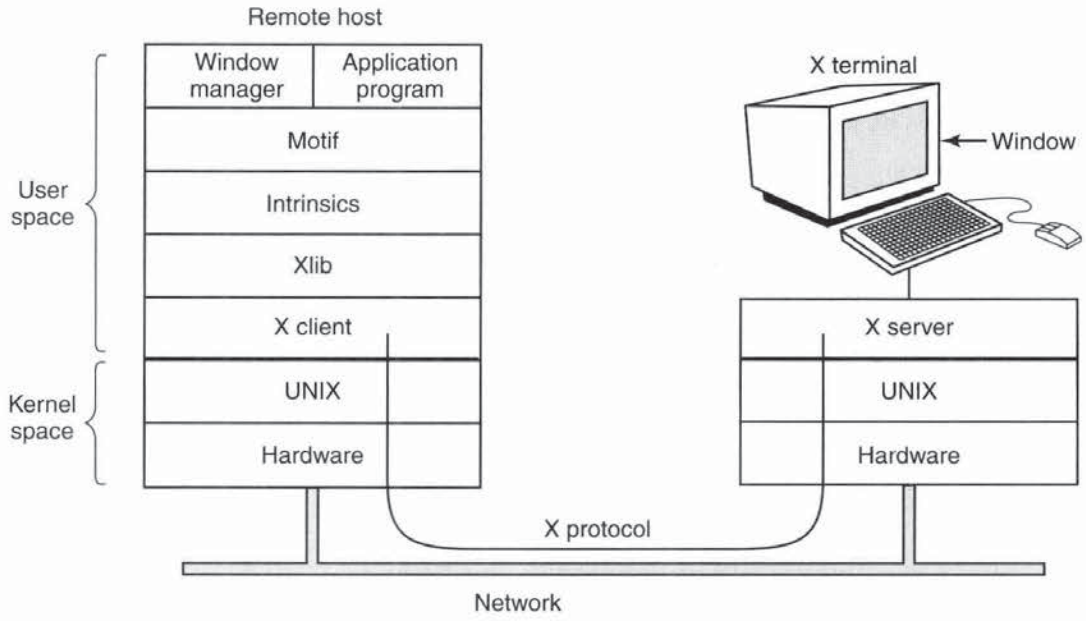


Figure 5-45. Clients and servers in the M.I.T. X Window System.

To make programming with X easier, a toolkit consisting of the **Intrinsics** is supplied as part of X. This layer manages buttons, scroll bars, and other GUI elements, called **widgets**. To make a true GUI interface, with a uniform look and feel, yet another layer is needed. The most popular one is called **Motif**. Most applications make use of calls to Motif rather than Xlib.

Also worth noting is that window management is not part of X itself. The decision to leave it out was fully intentional. Instead, a separate X client process, called a **window manager**, controls the creation, deletion, and movement of windows on the screen. To manage windows, it sends commands to the X server telling what to do. It often runs on the same machine as the X client, but in theory can run anywhere.

This modular design, consisting of several layers and multiple programs, makes X highly portable and flexible. It has been ported to most versions of UNIX, including Solaris, BSD, AIX, Linux, and so on, making it possible for application developers to have a standard user interface for multiple platforms. It has also been ported to other operating systems. In contrast, in Windows, the windowing and GUI systems are mixed together in the GDI and located in the kernel, which makes them harder to maintain. For example, the Windows 98 GUI is still fundamentally 16 bits, more than a decade after the Intel CPUs were 32 bits.

Now let us take a brief look at X as viewed from the Xlib level. When an X program starts, it opens a connection to one or more X servers— let us call them workstations even though they might be colocated on the same machine as the X

program itself. X considers this connection to be reliable in the sense that lost and duplicate messages are handled by the networking software and it does not have to worry about communication errors. Usually, TCP/IP is used between the client and server.

Four kinds of messages go over the connection:

1. Drawing commands from the program to the workstation.
2. Replies by the workstation to program queries.
3. Keyboard, mouse, and other event announcements.
4. Error messages.

Most drawing commands are sent from the program to the workstation as one-way messages. No reply is expected. The reason for this design is that when the client and server processes are on different machines, it may take a substantial period of time for the command to reach the server and be carried out. Blocking the application program during this time would slow it down unnecessarily. On the other hand, when the program needs information from the workstation, it simply has to wait until the reply comes back.

Like Windows, X is highly event driven. Events flow from the workstation to the program, usually in response to some human action such as keyboard strokes, mouse movements, or a window being uncovered. Each event message is 32 bytes, with the first byte giving the event type and the next 31 bytes providing additional information. Several dozen kinds of events exist, but a program is sent only those events that it has said it is willing to handle. For example, if a program does not want to hear about key releases, it is not sent any key release events. As in Windows, events are queued, and programs read events from the queue. However, unlike Windows, the operating system never calls procedures within the application program on its own. It does not even know which procedure handles which event.

A key concept in X is the **resource**. A resource is a data structure that holds certain information. Application programs create resources on workstations. Resources can be shared among multiple processes on the workstation. Resources tend to be short lived and do not survive workstation reboots. Typical resources include windows, fonts, colormaps (color palettes), pixmaps (bitmaps), cursors, and graphic contexts. The latter are used to associate properties with windows and are similar in concept to device contexts in Windows.

A rough, incomplete skeleton of an X program is shown in Fig. 5-46. It begins by including some required headers and then declaring some variables. It then connects to the X server specified as the parameter to *XOpenDisplay*. Then it allocates a window resource and stores a handle to it in *win*. In practice, some initialization would happen here. After that it tells the window manager that the new window exists so the window manager can manage it.

```
#include <X11/Xlib.h>
#include <X11/Xutil.h>

main(int argc, char *argv[])
{
    Display disp;                                   /* server identifier */
    Window win;                                     /* window identifier */
    GC gc;                                          /* graphic context identifier */
    XEvent event;                                   /* storage for one event */
    int running = 1;

    disp = XOpenDisplay("display_name");            /* connect to the X server */
    win = XCreateSimpleWindow(disp, ... );          /* allocate memory for new window */
    XSetStandardProperties(disp, ...);              /* announces window to window mgr */
    gc = XCreateGC(disp, win, 0, 0);                /* create graphic context */
    XSelectInput(disp, win, ButtonPressMask | KeyPressMask | ExposureMask);
    XMapRaised(disp, win);                  /* display window; send Expose event */

    while (running) {
        XNextEvent(disp, &event);           /* get next event */
        switch (event.type) {
            case Expose:        ...;  break;    /* repaint window */
            case ButtonPress:   ...;  break;    /* process mouse click */
            case Keypress:      ...;  break;    /* process keyboard input */
        }
    }

    XFreeGC(disp, gc);                      /* release graphic context */
    XDestroyWindow(disp, win);              /* deallocate window's memory space */
    XCloseDisplay(disp);                    /* tear down network connection */
}
```

Figure 5-46. A skeleton of an X Window application program.

The call to *XCreateGC* creates a graphic context in which properties of the window are stored. In a more complete program, they might be initialized here. The next statement, the call to *XSelectInput*, tells the X server which events the program is prepared to handle. In this case it is interested in mouse clicks, keystrokes, and windows being uncovered. In practice, a real program would be interested in other events as well. Finally, the call to *XMapRaised* maps the new window onto the screen as the uppermost window. At this point the window becomes visible on the screen.

The main loop consists of two statements and is logically much simpler than the corresponding loop in Windows. The first statement here gets an event and the second one dispatches on the event type for processing. When some event

indicates that the program has finished, *running* is set to 0 and the loop terminates. Before exiting, the program releases the graphic context, window, and connection.

It is worth mentioning that not everyone likes a GUI. Many programmers prefer a traditional command-line oriented interface of the type discussed in Sec. 5.6.2 above. X handles this via a client program called *xterm*. This program emulates an old VT102 intelligent terminal, complete with all the escape sequences. Thus editors such as *vi* and *emacs* and other software that uses termcap work in these windows without modification.

5.8.2 The SLIM Network Terminal

Over the years, the main computing paradigm has oscillated between centralized and decentralized computing. The first computers, such as the ENIAC, were, in fact, personal computers, albeit large ones, because only one person could use one at once. Then came timesharing systems, in which many remote users at simple terminals shared a big central computer. Next came the PC era, in which the users had their own personal computers again.

While the decentralized PC model has advantages, it also has some severe disadvantages that are only beginning to be taken seriously. Probably the biggest problem is that each PC has a large hard disk and complex software that must be maintained. For example, when a new release of the operating system comes out, a great deal of work has to be done to perform the upgrade on each machine separately. At most corporations, the labor costs of doing this kind of software maintenance dwarf the actual hardware and software costs. For home users, the labor is technically free, but few people are capable of doing it correctly and fewer still enjoy doing it. With a centralized system, only one or a few machines have to be updated and those machines have a staff of experts to do the work.

A related issue is that users should make regular backups of their gigabyte file systems, but few of them do. When disaster strikes, a great deal of moaning and wringing of hands tends to follow. With a centralized system, backups can be made every night by automated tape robots.

Another advantage is that resource sharing is easier with centralized systems. A system with 64 remote users, each with 64 MB of RAM will have most of that RAM idle most of the time. With a centralized system with 4 GB of RAM, it never happens that some user temporarily needs a lot of RAM but cannot get it because it is on someone else's PC. The same argument holds for disk space and other resources.

It is probably a fair conclusion to say that most users want high-performance interactive computing, but do not really want to administer a computer. This has led researchers to reexamine timesharing using dumb terminals (now politely called **thin clients**) that meet modern terminal expectations. X was a step in this direction, but an X server is still a complex system with megabytes of software

that must be upgraded from time to time. The holy grail would be a high performance interactive computing system in which the user machines had no software at all. Interestingly enough, this goal is achievable. Below we will describe one such system, developed by researchers at Sun Microsystems and Stanford University and now commercially available from Sun (Schmidt et al., 1999).

The system is called **SLIM**, which stands for **Stateless Low-level Interface Machine**. The idea is based on traditional centralized timesharing as shown in Fig. 5-47. The client machines are just dumb 1280 × 1024 bitmap displays with a keyboard and mouse but no user-installable software. They are very much in the spirit of the old intelligent character-oriented terminals that had a little firmware to interpret escape codes, but no other software. Like the old character-oriented terminals, the only user control is an on-off switch. Terminals of this type that have little processing capacity are called **thin clients**.

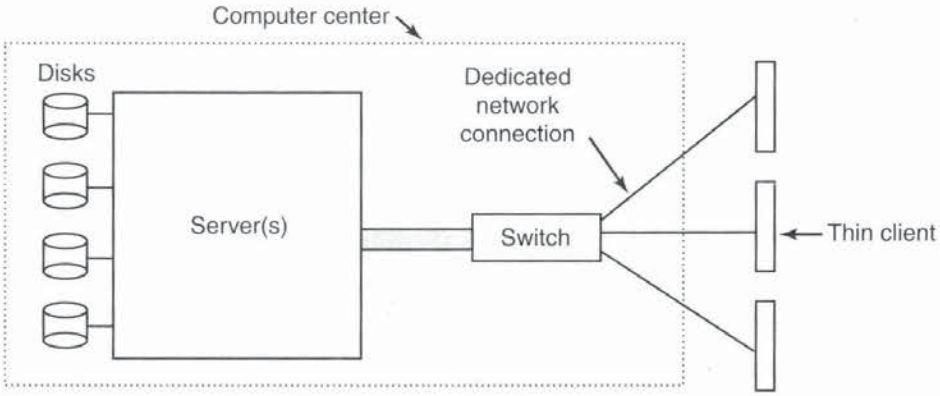


Figure 5-47. The architecture of the SLIM terminal system.

The simplest model—having the server ship bitmaps over the network to the SLIM clients 60 times a second—does not work. It requires about 2 Gbps of network bandwidth, which is far too much for current networks to handle. The next simplest model—storing the screen image in a frame buffer inside the terminal and having it refreshed 60 times a second locally— is much more promising. In particular, if the central server maintains a copy of each terminal's frame buffer and sends only updates (i.e., changes) to it as needed, the bandwidth requirements become quite modest. This is how the SLIM thin clients work.

Unlike the X protocol, which has hundreds of complex messages for managing windows, drawing geometric figures, and displaying text in many fonts, the SLIM protocol has only five display messages, listed in Fig. 5-48 (there are also a small number of control messages not listed). The first one, SET, just replaces a rectangle in the frame buffer with new pixels. Each pixel replaced requires 3 bytes in the message to specify its full (24-bit) color value. In theory, this message is sufficient to do the job; the other ones are just for optimization.

Message	Meaning
SET	Update a rectangle with new pixels
FILL	Fill a rectangle with one pixel value
BITMAP	Expand a bitmap to fill a rectangle
COPY	Copy a rectangle from one part of the frame buffer to another
CSCS	Convert a rectangle from television color (YUV) to RGB

Figure 5-48. Messages used in the SLIM protocol from the server to the terminals.

The FILL message fills an entire rectangle on the screen with a single pixel value. It is used for filling in uniform backgrounds. The BITMAP message fills an entire rectangle by repeating a pattern contained in a bitmap supplied in the message. This command is useful for filling in patterned backgrounds that have some texture and are not just a uniform color.

The COPY message instructs the terminal to copy a rectangle within the frame buffer to another part of the frame buffer. It is most useful for scrolling the screen and moving windows, for example.

Finally, the CSCS message converts the YUV color system used in U.S. (NTSC) television sets to the RGB system used by computer monitors. It is primarily used when a raw video frame has been shipped to a terminal in the YUV system and must be converted to RGB for display. Doing this conversion is algorithmically simple but time consuming, so it is better to offload the work to the terminals. If the terminals will not be used for watching videos, this message and its functionality are not needed.

The whole idea of dumb thin clients stands or falls with the performance, which Schmidt et al. extensively measured. In the prototype, 100-Mbps switched Fast Ethernet was used on both the server-to-switch segment and switch-to-terminal segments. Potentially a gigabit network could be used between the server and the switch because that segment is local to the central computer room.

The first measurement deals with character echoing to the screen. Each typed character is sent to the server, which computes which pixels have to be updated to place the character on the screen in the right position, font, and color. Measurements show that it takes 0.5 msec for the character to appear on the screen. In contrast, on a local workstation the echoing time is 30 msec due to kernel buffering.

The rest of the tests measured the performance with users running modern interactive application programs such as Adobe Photoshop (a program for retouching photographs), Adobe Framemaker (a desktop publishing program), and Netscape (a Web browser). It was observed that half the user commands required updating fewer than 10,000 pixels, which is 30,000 bytes uncompressed. At 100 Mbps, it takes 2.4 msec to pump 10,000 pixels out onto the wire. It takes another 2.7 msec to place them in the frame buffer upon arrival, for a total of 5.1 msec (but this varies a little depending on the circumstances). Since human reaction

time is about 100 msec, such updates appear to be instantaneous. Even the larger updates were almost instantaneous. Furthermore, when compression is used, more than 85% of the updates are under 30,000 bytes.

The experiments were repeated with a 10-Mbps network, a 1-Mbps network, and a 128-Kbps network. At 10 Mbps the system was virtually instantaneous and 1 Mbps it was still good. At 128 Kbps it was too slow to use. Since 1 Mbps connections to the home are rapidly becoming a reality using cable TV networks and ADSL (Asymmetric Digital Subscriber Loop), it appears that this technology may be applicable to home users as well as business users.

5.9 POWER MANAGEMENT

The first general-purpose electronic computer, the ENIAC, had 18,000 vacuum tubes and consumed 140,000 watts of power. As a result, it ran up a nontrivial electricity bill. After the invention of the transistor, power usage dropped dramatically and the computer industry lost interest in power requirements.
However, nowadays power management is back in the spotlight for several reasons, and the operating system is playing a role here.

Let us start with desktop PCs. A desktop PC often has a 200-watt power supply (which is typically 85% efficient, that is, loses 15% of the incoming energy to heat). If 100 million of these machines are turned on at once worldwide, together they use 20,000 megawatts of electricity. This is the total output of 20 average-sized nuclear power plants. If power requirements could be cut in half, we could get rid of 10 nuclear power plants. From an environmental point of view, getting rid of 10 nuclear power plants (or an equivalent number of fossil fuel plants) is a big win and well worth pursuing.

The other place where power is a big issue is on battery-powered computers, including notebooks, laptops, palmtops, and Webpads, among others. The heart of the problem is that the batteries cannot hold enough charge to last very long, a few hours at most. Furthermore, despite massive research efforts by battery companies, computer companies, and consumer electronics companies, progress is glacial. To an industry used to a doubling of the performance every 18 months (Moore's law), having no progress at all seems like a violation of the laws of physics, but that is the current situation. As a consequence, making computers use less energy so existing batteries last longer is high on everyone's agenda. The operating system plays a major role here, as we will see below.

There are two general approaches to reducing energy consumption. The first one is for the operating system to turn off parts of the computer (mostly I/O devices) when they are not in use because a device that is off uses little or no energy. The second one is for the application program to use less energy, possibly degrading the quality of the user experience, in order to stretch out battery time. We will look at each of these approaches in turn, but first we will say a little bit about hardware design with respect to power usage.

5.9.1 Hardware Issues

Batteries come in two general types: disposable and rechargeable. Disposable batteries (most commonly AAA, AA, and D cells) can be used to run handheld devices, but do not have enough energy to power laptop computers with large bright screens. A rechargeable battery, in contrast, can store enough energy to power a laptop for a few hours. Nickel cadmium batteries used to dominate here, but they gave way to nickel metal hydride batteries, which last longer and do not pollute the environment quite as badly when they are eventually discarded. Lithium ion batteries are even better, and may be recharged without first being fully drained, but their capacities are also severely limited.

The general approach most computer vendors take to battery conservation is to design the CPU, memory, and I/O devices to have multiple states: on, sleeping, hibernating, and off. To use the device, it must be on. When the device will not be needed for a short time, it can be put to sleep, which reduces energy consumption. When it is not expected to be needed for a longer interval, it can be made to hibernate, which reduces energy consumption even more. The trade-off here is that getting a device out of hibernation often takes more time and energy than getting it out of sleep state. Finally, when a device is off, it does nothing and consumes no power. Not all devices have all these states, but when they do, it is up to the operating system to manage the state transitions at the right moments.

Some computers have two or even three power buttons. One of these may put the whole computer in sleep state, from which it can be awakened quickly by typing a character or moving the mouse. Another may put the computer into hibernation, from which wakeup takes much longer. In both cases, these buttons typically do nothing except send a signal to the operating system, which does the rest in software. In some countries, electrical devices must, by law, have a mechanical power switch that breaks a circuit and removes power from the device, for safety reasons. To comply with this law, another switch may be needed.

Power management brings up a number of questions that the operating system must deal with. They include the following. Which devices can be controlled? Are they on/off, or do they have intermediate states? How much power is saved in the low-power states? Is energy expended to restart the device? Must some context be saved when going to a low-power state? How long does it take to go back to full power? Of course, the answers to these questions vary from device to device, so the operating system must be able to deal with a range of possibilities.

Various researchers have examined laptop computers to see where the power goes. Li et al. (1994) measured various workloads and came to the conclusions shown in Fig. 5-49. Lorch and Smith (1998) made measurements on other machines and came to the conclusions shown in Fig. 5-49. Weiser et al. (1994) also made measurements but did not publish the numerical values. They simply stated that the top three energy sinks were the display, hard disk, and CPU, in that order. While these numbers do not agree closely, possibly because the different brands

of computers measured indeed have different energy requirements, it seems clear that the display, hard disk, and CPU are obvious targets for saving energy.

Device	Li et al. (1994)	Lorch and Smith (1998)
Display	68%	39%
CPU	12%	18%
Hard disk	20%	12%
Modem		6%
Sound		2%
Memory	0.5%	1%
Other		22%

Figure 5-49. Power consumption of various parts of a laptop computer.

5.9.2 Operating System Issues

The operating system plays a key role in energy management. It controls all the devices, so it must decide what to shut down and when to shut it down. If it shuts down a device and that device is needed again quickly, there may be an annoying delay while it is restarted. On the other hand, if it waits too long to shut down a device, energy is wasted for nothing.

The trick is to find algorithms and heuristics that let the operating system make good decisions about what to shut down and when. The trouble is that "good" is highly subjective. One user may find it acceptable that after 30 seconds of not using the computer it takes 2 seconds for it to respond to a keystroke. Another user may swear a blue streak under the same conditions. In the absence of audio input, the computer cannot tell these users apart.

The Display

Let us now look at the big spenders of the energy budget to see what can be done about each one. The biggest item in everyone's energy budget is the display. To get a bright sharp image, the screen must be backlit and that takes substantial energy. Many operating systems attempt to save energy here by shutting down the display when there has been no activity for some number of minutes. Often the user can decide what the shutdown interval is, pushing the trade-off between frequent blanking of the screen and using the battery up quickly back to the user (who probably really does not want it). Turning off the display is a sleep state because it can be regenerated (from the video RAM) almost instantaneously when any key is struck or the pointing device is moved.

One possible improvement was proposed by Flinn and Satyanarayanan (1999). They suggested having the display consist of some number of zones that can be independently powered up or down. In Fig. 5-50, we depict 16 zones using dashed lines to separate them. When the cursor is in window 2, as shown in Fig. 5-50(a), only the four zones in the lower righthand corner have to be lit up. The other 12 can be dark, saving 3/4 of the screen power.

When the user moves the cursor to window 1, the zones for window 2 can be darkened and the zones behind window 1 can be turned on. However, because window 1 straddles 9 zones, more power is needed. If the window manager can sense of what is happening, it can automatically move window 1 to fit into four zones, with a kind of snap-to-zone action, as shown in Fig. 5-50(b). To achieve this reduction from 9/16 of full power to 4/16 of full power, the window manager has to understand power management or be capable of accepting instructions from some other piece of the system that does. Even more sophisticated would be the ability to partially illuminate a window that was not completely full (e.g., a window containing short lines of text could be kept dark on the right hand side).

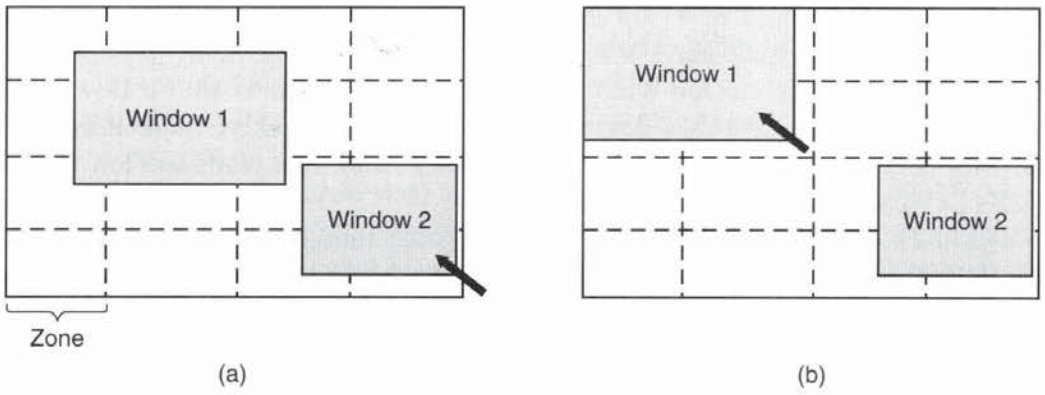


Figure 5-50. The use of zones for backlighting the display. (a) When window 2 is selected it is not moved. (b) When window 1 is selected, it moves to reduce the number of zones illuminated.

The Hard Disk

Another major villain is the hard disk. It takes substantial energy to keep it spinning at high speed, even if there are no accesses. Many computers, especially laptops, spin the disk down after a certain number of minutes of activity. When it is next needed, it is spun up again. Unfortunately, a stopped disk is hibernating rather than sleeping because it takes quite a few seconds to spin it up again, which causes noticeable delays for the user.

In addition, restarting the disk consumes considerable extra energy. As a consequence, every disk has a characteristic time, T_d, that is its break-even point,

often in the range 5 to 15 sec. Suppose that the next disk access is expected to some time t in the future. If $t < T_d$, it takes less energy to keep the disk spinning rather than spin it down and then spin it up so quickly. If $t > T_d$, the energy saved makes it worth spinning the disk down and up again much later. If a good prediction could be made (e.g., based on past access patterns), the operating system could make good shutdown predictions and save energy. In practice, most systems are conservative and only spin down the disk after a few minutes of inactivity.

Another way to save disk energy is to have a substantial disk cache in RAM. If a needed block is in the cache, an idle disk does not have to be restarted to satisfy the read. Similarly, if a write to the disk can be buffered in the cache, a stopped disk does not have to restarted just to handle the write. The disk can remain off until the cache fills up or a read miss happens.

Another way to avoid unnecessary disk starts is for the operating system to keep running programs informed about the disk state by sending it messages or signals. Some programs have discretionary writes that can be skipped or delayed. For example, a word processor may be set up to write the file being edited to disk every few minutes. If the word processor knows that the disk is off at the moment it would normally write the file out, it can delay this write until the disk is next turned on or until a certain additional time has elapsed.

The CPU

The CPU can also be managed to save energy. A laptop CPU can be put to sleep in software, reducing power usage to almost zero. The only thing it can do in this state is wake up when an interrupt occurs. Therefore, whenever the CPU goes idle, either waiting for I/O or because there is no work to do, it goes to sleep.

On many computers, there is a relationship between CPU voltage, clock cycle, and power usage. The CPU voltage can often be reduced in software, which saves energy but also reduces the clock cycle (approximately linearly). Since power consumed is proportional to the square of the voltage, cutting the voltage in half makes the CPU about half as fast but at 1/4 the power.

This property can be exploited for programs with well-defined deadlines, such as multimedia viewers that have to decompress and display a frame every 40 msec, but go idle if they do it faster. Suppose that a CPU uses x joules while running full blast for 40 msec and $x/4$ joules running at half speed. If a multimedia viewer can decompress and display a frame in 20 msec, the operating system can run at full power for 20 msec and then shut down for 20 msec for a total energy usage of $x/2$ joules. Alternatively, it can run at half power and just make the deadline, but use only $x/4$ joules instead. A comparison of running at full speed and full power for some time interval and at half speed and one quarter power for twice as long is shown in Fig. 5-51. In both cases the same work is done, but in Fig. 5-51(b) only half the energy is consumed doing it.

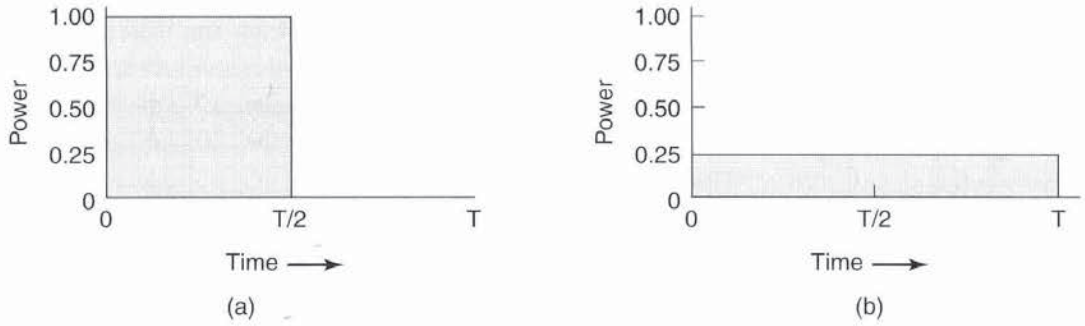


Figure 5-51. (a) Running at full clock speed. (b) Cutting voltage by two cuts clock speed by two and power consumption by four.

In a similar vein, if a user is typing at 1 char/sec, but the work needed to process the character takes 100 msec, it is better for the operating system to detect the long idle periods and slow the CPU down by a factor of 10. In short, running slowly is more energy efficient than running quickly.

The Memory

Two possible options exist for saving energy with the memory. First, the cache can be flushed and then switched off. It can always be reloaded from main memory with no loss of information. The reload can be done dynamically and quickly, so turning off the cache is entering a sleep state.

A more drastic option is to write the contents of main memory to the disk, then switch off the main memory itself. This approach is hibernation, since virtually all power can be cut to memory at the expense of a substantial reload time, especially if the disk is off too. When the memory is cut off, the CPU either has to be shut off as well or has to execute out of ROM. If the CPU is off, the interrupt that wakes it up has to cause it to jump to code in a ROM so the memory can be reloaded before being used. Despite all the overhead, switching off the memory for long periods of time (e.g., hours) may be worth it if restarting in a few seconds is considered much more desirable than rebooting the operating system from disk, which often takes a minute or more.

Wireless Communication

Increasingly many portable computers have a wireless connection to the outside world (e.g., the Internet). The radio transmitter and receiver required are often first-class power hogs. In particular, if the radio receiver is always on in order to listen for incoming email, the battery may drain fairly quickly. On the other hand, if the radio is switched off after, say, 1 minute of being idle, incoming messages may be missed, which is clearly undesirable.

One efficient solution to this problem has been proposed by Kravets and Krishnan (1998). The heart of their solution exploits the fact that mobile computers communicate with fixed base stations that have large memories and disks and no power constraints. What they propose is to have the mobile computer send a message to the base station when it is about to turn off the radio. From that time on, the base station buffers incoming messages on its disk. When the mobile computer switches on the radio again, it tells the base station. At that point any accumulated messages can be sent to it.

Outgoing messages that are generated while the radio is off are buffered on the mobile computer. If the buffer threatens to fill up, the radio is turned on and the queue transmitted to the base station.

When should the radio be switched off? One possibility is to let the user or the application program decide. Another is turn it off after some number of seconds of idle time. When should it be switched on again? Again, the user or program could decide, or it could be switched on periodically to check for inbound traffic and transmit any queued messages. Of course, it also should be switched on when the output buffer is close to full. Various other heuristics are possible.

Thermal Management

A somewhat different, but still energy-related issue, is thermal management. Modern CPUs get extremely hot due to their high speed. Desktop machines normally have an internal electric fan to blow the hot air out of the chassis. Since reducing power consumption is usually not a driving issue with desktop machines, the fan is usually on all the time.

With laptops, the situation is different. The operating system has to monitor the temperature continuously. When it gets close to the maximum allowable temperature, the operating system has a choice. It can switch on the fan, which makes noise and consumes power. Alternatively, it can reduce power consumption by reducing the backlighting of the screen, slowing down the CPU, being more aggressive about spinning down the disk, and so on.

Some input from the user may be valuable as a guide. For example, a user could specify in advance that the noise of the fan is objectionable, so the operating system would reduce power consumption instead.

Battery Management

In ye olde days, a battery just provided current until it was drained, at which time it stopped. Not any more. Laptops use smart batteries now, which can communicate with the operating system. Upon request they can report on things like maximum voltage, current voltage, maximum charge, current charge, maximum drain rate, current drain rate, and more. Most laptop computers have programs that can be run to query and display all these parameters. Smart batteries can also

be instructed to change various operational parameters under control of the operating system.

Some laptops have multiple batteries. When the operating system detects that one battery is about to go, it has to arrange for a graceful cutover to the next one, without causing any glitches during the transition. When the final battery is on its last legs, it is up to the operating system to warn the user and then cause an orderly shutdown, for example, making sure that the file system is not corrupted.

Driver Interface

The Windows system has an elaborate mechanism for doing power management called **ACPI** (**Advanced Configuration and Power Interface**). The operating system can send any conformant driver commands asking it to report on the capabilities of its devices and their current states. This feature is especially important when combined with plug and play because just after it is booted, the operating system does not even know what devices are present, let alone their properties with respect to energy consumption or power manageability.

It can also sends commands to drivers instructing them to cut their power levels (based on the capabilities that it learned earlier, of course). There is also some traffic the other way. In particular, when a device such as a keyboard or a mouse detects activity after a period of idleness, this is a signal to the system to go back to (near) normal operation.

5.9.3 Degraded Operation

So far we have looked at ways the operating system can reduce energy usage by various kinds of devices. But there is another approach as well: tell the programs to use less energy, even if this means providing a poorer user experience (better a poorer experience than no experience when the battery dies and the lights go out). Typically, this information is passed on when the battery charge is below some threshold. It is then up to the programs to decide between degrading performance to lengthen battery life or to maintain performance and risk running out of energy.

One of the questions that comes up here is how can a program degrade its performance to save energy? This question has been studied by Flinn and Satyanarayanan (1999). They provided four examples of how degraded performance can save energy. We will now look at these.

In this study, information is presented to the user in various forms. When no degradation is present, the best possible information is presented. When degradation is present, the fidelity (accuracy) of the information presented to the user is worse than what it could have been. We will see examples of this shortly.

In order to measure energy usage, Flinn and Satyanarayanan devised a software tool called PowerScope. What it does is provide a power usage profile of a

program. To use it, a computer must be hooked up to an external power supply through a software-controlled digital multimeter. Using the multimeter, software can read out the number of milliamperes coming in from the power supply and thus determine the instantaneous power being consumed by the computer. What PowerScope does is periodically sample the program counter and the power usage and write these data to a file. After the program has terminated the file is analyzed to give the energy usage of each procedure. These measurements formed the basis of their observations. Hardware energy saving measures were also used and formed the baseline against which the degraded performance was measured.

The first program measured was a video player. In undegraded mode, it plays 30 frames/sec in full resolution and in color. One form of degradation is to abandon the color information and display the video in black and white. Another form of degradation is to reduce the frame rate, which leads to flicker and gives the movie a jerky quality. Still another form of degradation is to reduce the number of pixels in both directions, either by lowering the spatial resolution or making the displayed image smaller. Measures of this type saved about 30% of the energy.

The second program was a speech recognizer. It sampled the microphone to construct a waveform. This waveform could either be analyzed on the laptop computer or sent over a radio link for analysis on a fixed computer. Doing this saves CPU energy but uses energy for the radio. Degradation was accomplished by using a smaller vocabulary and a simpler acoustic model. The win here was about 35%

The next example was a map viewer that fetched the map over the radio link. Degradation consisted of either cropping the map to smaller dimensions or telling the remote server to omit smaller roads, thus requiring fewer bits to be transmitted. Again here a gain of about 35% was achieved.

The fourth experiment was with transmission of JPEG images to a Web browser. The JPEG standard allows various algorithms, trading image quality against file size. Here the gain averaged only 9%. Still, all in all, the experiments showed that by accepting some quality degradation, the user can run longer on a given battery.

5.10 RESEARCH ON INPUT/OUTPUT

There is a fair amount of research on input/output, but most of it is focused on specific devices, rather than I/O in general. Often the goal is to improve performance in one way or another.

Disk systems are a case in point. Older disk arm scheduling algorithms use a disk model that is not really applicable any more, so Worthington et al. (1994) took a look at models that correspond to modern disks. RAID is a hot topic, with various researchers look at different aspects of it. Alvarez et al. (1997) looked at

enhanced fault tolerance, as did Blaum et al. (1994). Cao et al. (1994) examined the idea of having a parallel controller on a RAID. Wilkes et al. (1996) described an advanced RAID system they built at HP. Having multiple drives requires good parallel scheduling, so that is also a research topic (Chen and Towsley, 1996; and Kallahalla and Varman, 1999). Lumb et al. (2000) argue for utilizing the idle time after the seek but before the sector needed rotates by the head to preload data. Even better than using the rotational latency to due useful work is to eliminate the rotation in the first place using a solid-state microelectromechanical storage device (Griffin et al., 2000; and Carley et al., 2000) or holographic storage (Orlov, 2000). Another new technology worth watching is magneto-optical storage (McDaniel, 2000).

The SLIM terminal provides a modern version of the old timesharing system, with all the computing done centrally and providing users with terminals that just manage the display, mouse, and keyboard, and nothing else (Schmidt et al., 1999). The main difference with old-time timesharing is that instead of connecting the terminal to the computer with a 9600-bps modem, a 10-Mbps Ethernet is used, which provides enough bandwidth for a full graphical interface at the user's end.

GUIs are fairly standard now, but there is still work continuing to go on in that area, for example, speech input (Malkewitz, 1998; Manaris and Harkreader, 1998; Slaughter et al., 1998; and Van Buskirk and LaLomia, 1995). Internal structure of the GUI is also a research topic (Taylor et al., 1995).

Given the large number of computer scientists with laptop computers and given the microscopic battery lifetime on most of them, it should come as no surprise that there is a lot of interest in using software techniques to manage and conserve battery power (Ellis, 1999; Flinn and Satyanarayanan, 1999; Kravets and Krishnan, 1998; Lebeck et al., 2000; Lorch and Smith, 1996; and Lu et al., 1999).

5.11 SUMMARY

Input/output is an often neglected, but important, topic. A substantial fraction of any operating system is concerned with I/O. I/O can be accomplished in one of three ways. First, there is programmed I/O, in which the main CPU inputs or outputs each byte or word and sits in a tight loop waiting until it can get or send the next one. Second, there is interrupt-driven I/O, in which the CPU starts an I/O transfer for a character or word and goes off to do something else until an interrupt arrives signaling completion of the I/O. Third, there is DMA, in which a separate chip manages the complete transfer of a block of data, given an interrupt only when the entire block has been transferred.

I/O can be structured in four levels: the interrupt service procedures, the device drivers, the device-independent I/O software, and the I/O libraries and spoolers that run in user space. The device drivers handle the details of running the

devices and providing uniform interfaces to the rest of the operating system. The device-independent I/O software does things like buffering and error reporting.

Disks come in a variety of types, including magnetic disks, RAIDs, and various kinds of optical disks. Disk arm scheduling algorithms can often be used to improve disk performance, but the presence of virtual geometries complicates matters. By pairing two disks, a stable storage medium with certain useful properties can be constructed.

Clocks are used for keeping track of the real time, limiting how long processes can run, handling watchdog timers, and doing accounting.

Character-oriented terminals have a variety of issues concerning special characters that can be input and special escape sequences that can be output. Input can be in raw mode or cooked mode, depending on how much control the program wants over the input. Escape sequences on output control cursor movement and allow for inserting and deleting text on the screen.

Many personal computers use GUIs for their output. These are based on the WIMP paradigm: windows, icons, menus and a pointing device. GUI-based programs are generally event driven, with keyboard, mouse, and other events being sent to the program for processing as soon as they happen.

Network terminals come in several varieties. One of the most popular consists of those running X, a sophisticated system that can be used to build various GUIs. An alternative to X Windows is a low-level interface that simply ships raw pixels across the network. Experiments with the SLIM terminal show that this technique works surprisingly well.

Finally, power management is a major issue for laptop computers because battery lifetimes are limited. Various techniques can be employed by the operating system to reduce power consumption. Programs can also help out by sacrificing some quality for longer battery lifetimes.

PROBLEMS

1. Advances in chip technology have made it possible to put an entire controller, including all the bus access logic, on an inexpensive chip. How does that affect the model of Fig. 1-5?
2. Given the speeds listed in Fig. 5-1, is it possible to scan documents from a scanner onto an EIDE disk attached to an ISA bus at full speed? Defend your answer.
3. Figure 5-3(b) shows one way of having memory-mapped I/O even in the presence of separate buses for memory and I/O devices, namely, to first try the memory bus and if that fails try the I/O bus. A clever computer science student has thought of an improvement on this idea: try both in parallel, to speed up the process of accessing I/O devices. What do you think of this idea?

4. A DMA controller has four channels. The controller is capable of requesting a 32-bit word every 100 nsec. A response takes equally long. How fast does the bus have to be to avoid being a bottleneck?

5. Suppose that a computer can read or write a memory word in 10 nsec. Also suppose that when an interrupt occurs, all 32 CPU registers, plus the program counter and PSW are pushed onto the stack. What is the maximum number of interrupts per second this machine can process?

6. In Fig. 5-8(b), the interrupt is not acknowledged until after the next character has been output to the printer. Could it have equally well been acknowledged right at the start of the interrupt service procedure? If so, give one reason for doing it at the end, as in the text. If not, why not?

7. A computer has a three-stage pipeline as shown in Fig. 1-6(a). On each clock cycle, one new instruction is fetched from memory at the address pointed to by the PC and put into the pipeline and the PC advanced. Each instruction occupies exactly one memory word. The instructions already in the pipeline are each advanced one stage. When an interrupt occurs, the current PC is pushed onto the stack, and the PC is set to the address of the interrupt handler. Then the pipeline is shifted right one stage and the first instruction of the interrupt handler is fetched into the pipeline. Does this machine have precise interrupts? Defend your answer.

8. A typical printed page of text contains 50 lines of 80 characters each. Imagine that a certain printer can print 6 pages per minute and that the time to write a character to the printer's output register is so short it can be ignored. Does it make sense to run this printer using interrupt-driven I/O if each character printed requires an interrupt that takes 50 μsec all-in to service?

9. What is "device independence"?

10. In which of the four I/O software layers is each of the following done.

    (a) Computing the track, sector, and head for a disk read.
    (b) Writing commands to the device registers.
    (c) Checking to see if the user is permitted to use the device.
    (d) Converting binary integers to ASCII for printing.

11. Based on the data of Fig. 5-17, what is the transfer rate for transfers between the disk and the controller for a floppy disk and a hard disk? How does this compare with a 56-Kbps modem and 100-Mbps Fast Ethernet, respectively?

12. A local area network is used as follows. The user issues a system call to write data packets to the network. The operating system then copies the data to a kernel buffer. Then it copies the data to the network controller board. When all the bytes are safely inside the controller, they are sent over the network at a rate of 10 megabits/sec. The receiving network controller stores each bit a microsecond after it is sent. When the last bit arrives, the destination CPU is interrupted, and the kernel copies the newly arrived packet to a kernel buffer to inspect it. Once it has figured out which user the packet is for, the kernel copies the data to the user space. If we assume that each interrupt and its associated processing takes 1 msec, that packets are 1024 bytes (ignore the headers), and that copying a byte takes 1 μsec, what is the maximum rate

at which one process can pump data to another? Assume that the sender is blocked until the work is finished at the receiving side and an acknowledgement comes back. For simplicity, assume that the time to get the acknowledgement back is so small it can be ignored.

13. Why are output files for the printer normally spooled on disk before being printed?

14. How much cylinder skew is needed for a 7200-rpm disk with a track-to-track seek time of 1 msec? The disk has 200 sectors of 512 bytes each on each track.

15. Calculate the maximum data rate in MB/sec for the disk described in the previous problem.

16. RAID level 3 is able to correct single-bit errors using only one parity drive. What is the point of RAID level 2? After all, it also can only correct one error and takes more drives to do so.

17. A RAID can fail if two or more of its drives crash within a short time interval. Suppose that the probability of one drive crashing in a given hour is p. What is the probability of a k-drive RAID failing in a given hour?

18. Why are optical storage devices inherently capable of higher data density than magnetic storage devices? *Note*: This problem requires some knowledge of high-school physics and how magnetic fields are generated.

19. If a disk controller writes the bytes it receives from the disk to memory as fast as it receives them, with no internal buffering, is interleaving conceivably useful? Discuss.

20. A floppy disk is double interleaved, as in Fig. 5-26(c). It has eight sectors of 512 bytes per track, and a rotation rate of 300 rpm. How long does it take to read all the sectors of a track in order, assuming the arm is already correctly positioned, and 1/2 rotation is needed to get sector 0 under the head? What is the data rate? Now repeat the problem for a noninterleaved disk with the same characteristics. How much does the data rate degrade due to interleaving?

21. If a disk has double interleaving, does it also need cylinder skew in order to avoid missing data when making a track-to-track seek? Discuss your answer.

22. A disk manufacturer has two 5.25-inch disks that each have 10,000 cylinders. The newer one has double the linear recording density of the older one. Which disk properties are better on the newer drive and which are the same?

23. A computer manufacturer decides to redesign the partition table of a Pentium hard disk to provide more than four partitions. What are some consequences of this change?

24. Disk requests come in to the disk driver for cylinders 10, 22, 20, 2, 40, 6, and 38, in that order. A seek takes 6 msec per cylinder moved. How much seek time is needed for

(a) First-come, first served.
(b) Closest cylinder next.
(c) Elevator algorithm (initially moving upward).

In all cases, the arm is initially at cylinder 20.

25. A personal computer salesman visiting a university in South-West Amsterdam remarked during his sales pitch that his company had devoted substantial effort to making their version of UNIX very fast. As an example, he noted that their disk driver used the elevator algorithm and also queued multiple requests within a cylinder in sector order. A student, Harry Hacker, was impressed and bought one. He took it home and wrote a program to randomly read 10,000 blocks spread across the disk. To his amazement, the performance that he measured was identical to what would be expected from first-come, first-served. Was the salesman lying?

26. In the discussion of stable storage using nonvolatile RAM, the following point was glossed over. What happens if the stable write completes but a crash occurs before the operating system can write an invalid block number in the nonvolatile RAM? Does this race condition ruin the abstraction of stable storage? Explain your answer.

27. The clock interrupt handler on a certain computer requires 2 msec (including process switching overhead) per clock tick. The clock runs at 60 Hz. What fraction of the CPU is devoted to the clock?

28. Many versions of UNIX use an unsigned 32-bit integer to keep track of the time as the number of seconds since the origin of time. When will these systems wrap around (year and month)? Do you expect this to actually happen?

29. Some computers need to have large numbers of RS-232 lines, for example, servers or Internet providers. For this reason, plug-in cards with multiple RS-232 lines exist. Suppose that such a card contains a processor that must sample each incoming line at 8 times the baud rate to see if the incoming bit is a 0 or a 1. Also suppose that such a sample takes 1 μsec. For 28,800-bps lines operating at 3200 baud, how many lines can the processor support? Note: The baud rate of a line is the number of signal changes per second. A 3200-baud line can support 28,800 bps if each signaling interval encodes 9 bits using various amplitudes, frequencies, and phases. As an aside, 56K modems do not use RS-232, so are not a suitable example of RS-232 timing.

30. Why are RS232 terminals interrupt driven, but memory-mapped terminals not interrupt driven?

31. Consider the performance of a 56-Kbps modem. The driver outputs one character and then blocks. When the character has been printed, an interrupt occurs and a message is sent to the blocked driver, which outputs the next character and then blocks again. If the time to pass a message, output a character, and block is 100 μsec, what fraction of the CPU is eaten by the modem handling? Assume that each character has one start bit and one stop bit, for 10 bits in all.

32. A bitmap terminal contains 1280 by 960 pixels. To scroll a window, the CPU (or controller) must move all the lines of text upward by copying their bits from one part of the video RAM to another. If a particular window is 60 lines high by 80 characters wide (5280 characters, total), and a character's box is 8 pixels wide by 16 pixels high, how long does it take to scroll the whole window at a copying rate of 50 nsec per byte? If all lines are 80 characters long, what is the equivalent baud rate of the terminal? Putting a character on the screen takes 5 μsec. How many lines per second can be displayed?

33. After receiving a DEL (SIGINT) character, the display driver discards all output currently queued for that display. Why?

34. A user at an RS-232 terminal issues a command to an editor to delete the word on line 5 occupying character positions 7 through and including 12. Assuming the cursor is not on line 5 when the command is given, what ANSI escape sequence should the editor emit to delete the word?

35. Many RS232 terminals have escape sequences for deleting the current line and moving all the lines below it up one line. How do you think this feature is implemented inside the terminal?

36. On the original IBM PC's color display, writing to the video RAM at any time other than during the CRT beam's vertical retrace caused ugly spots to appear all over the screen. A screen image is 25 by 80 characters, each of which fits in a box 8 pixels by 8 pixels. Each row of 640 pixels is drawn on a single horizontal scan of the beam, which takes 63.6 μsec, including the horizontal retrace. The screen is redrawn 60 times a second, each of which requires a vertical retrace period to get the beam back to the top. What fraction of the time is the video RAM available for writing in?

37. The designers of a computer system expected that the mouse could be moved at a maximum rate of 20 cm/sec. If a mickey is 0.1 mm and each mouse message is 3 bytes, what is the maximum data rate of the mouse assuming that each mickey is reported separately?

38. The primary additive colors are red, green, and blue, which means that any color can be constructed from a linear superposition of these colors. Is it possible that someone could have a color photograph that cannot be represented using full 24-bit color?

39. One way to place a character on a bitmapped screen is to use bitblt from a font table. Assume that a particular font uses characters that are 16×24 pixels in true RGB color.

(a) How much font table space does each character take?
(b) If copying a byte takes 100 nsec, including overhead, what is the output rate to the screen in characters/sec?

40. Assuming that it takes 10 nsec to copy a byte, how much time does it take to completely rewrite the screen of an 80 character $\times$ 25 line text mode memory-mapped screen? What about a 1024×768 pixel graphics screen with 24-bit color?

41. In Fig. 5-41 there is a class to *RegisterClass*. In the corresponding X Window code, in Fig. 5-46, there is no such call or anything like it. Why not?

42. In the text we gave an example of how to draw a rectangle on the screen using the Windows GDI:

```
Rectangle(hdc, xleft, ytop, xright, ybottom);
```

Is there any real need for the first parameter (*hdc*), and if so, what? After all, the coordinates of the rectangle are explicitly specified as parameters.

43. A SLIM terminal is used to display a Web page containing an animated cartoon of size 400 pixels $\times$ 160 pixels running at 10 frames/sec. What fraction of a 100-Mbps Fast Ethernet is consumed by displaying the cartoon?

44. It has been observed that the SLIM system works well with a 1-Mbps network in a test. Are any problems likely in a multiuser situation? *Hint*: Consider a large number of users watching a scheduled TV show and the same number of users browsing the World Wide Web.

45. If a CPU's maximum voltage, V, is cut to V/n, its power consumption drops to $1/n^2$ of its original value and its clock speed drops to $1/n$ of its original value. Suppose that a user is typing at 1 char/sec, but the CPU time required to process each character is 100 msec. What is the optimal value of n and what is the corresponding energy saving in percent compared to not cutting the voltage? Assume that an idle CPU consumes no energy at all.

46. A laptop computer is set up to take maximum advantage of power saving features including shutting down the display and the hard disk after periods of inactivity. A user sometimes runs UNIX programs in text mode, and at other times uses the X Window System. She is surprised to find that battery life is significantly better when she uses text-only programs. Why?

47. Write a program that simulates stable storage. Use two large fixed-length files on your disk to simulate the two disks.

6

FILE SYSTEMS

All computer applications need to store and retrieve information. While a process is running, it can store a limited amount of information within its own address space. However, the storage capacity is restricted to the size of the virtual address space. For some applications this size is adequate, but for others, such as airline reservations, banking, or corporate record keeping, it is far too small.

A second problem with keeping information within a process' address space is that when the process terminates, the information is lost. For many applications, (e.g., for databases), the information must be retained for weeks, months, or even forever. Having it vanish when the process using it terminates is unacceptable. Furthermore, it must not go away when a computer crash kills the process.

A third problem is that it is frequently necessary for multiple processes to access (parts of) the information at the same time. If we have an online telephone directory stored inside the address space of a single process, only that process can access it. The way to solve this problem is to make the information itself independent of any one process.

Thus we have three essential requirements for long-term information storage:

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

The usual solution to all these problems is to store information on disks and other external media in units called **files**. Processes can then read them and write new

ones if need be. Information stored in files must be **persistent**, that is, not be affected by process creation and termination. A file should only disappear when its owner explicitly removes it.

Files are managed by the operating system. How they are structured, named, accessed, used, protected, and implemented are major topics in operating system design. As a whole, that part of the operating system dealing with files is known as the **file system** and is the subject of this chapter.

From the users' standpoint, the most important aspect of a file system is how it appears to them, that is, what constitutes a file, how files are named and protected, what operations are allowed on files, and so on. The details of whether linked lists or bitmaps are used to keep track of free storage and how many sectors there are in a logical block are of less interest, although they are of great importance to the designers of the file system. For this reason, we have structured the chapter as several sections. The first two are concerned with the user interface to files and directories, respectively. Then comes a detailed discussion of how the file system is implemented. Finally, we give some examples of real file systems.

6.1 FILES

In the following pages we will look at files from the user's point of view, that is, how they are used and what properties they have.

6.1.1 File Naming

Files are an abstraction mechanism. They provide a way to store information on the disk and read it back later. This must be done in such a way as to shield the user from the details of how and where the information is stored, and how the disks actually work.

Probably the most important characteristic of any abstraction mechanism is the way the objects being managed are named, so we will start our examination of file systems with the subject of file naming. When a process creates a file, it gives the file a name. When the process terminates, the file continues to exist and can be accessed by other processes using its name.

The exact rules for file naming vary somewhat from system to system, but all current operating systems allow strings of one to eight letters as legal file names. Thus *andrea*, *bruce*, and *cathy* are possible file names. Frequently digits and special characters are also permitted, so names like *2*, *urgent!*, and *Fig.2-14* are often valid as well. Many file systems support names as long as 255 characters.

Some file systems distinguish between upper and lower case letters, whereas others do not. UNIX falls in the first category; MS-DOS falls in the second. Thus a UNIX system can have all of the following as three distinct files: *maria*, *Maria*, and *MARIA*. In MS-DOS, all these names refer to the same file.

An aside on file systems is probably in order here. Windows 95 and Windows 98 both use the MS-DOS file system, and thus inherit many of its properties, such as how file names are constructed. In addition, Windows NT and Windows 2000 support the MS-DOS file system and thus also inherit its properties. However, the latter two systems also have a native file system (NTFS) that has different properties (such as file names in Unicode). In this chapter, when we refer to the Windows file system, we mean the MS-DOS file system, which is the only file system supported by all versions of Windows. We will discuss the Windows 2000 native file system in Chap. 11.

Many operating systems support two-part file names, with the two parts separated by a period, as in *prog.c*. The part following the period is called the **file extension** and usually indicates something about the file. In MS-DOS, for example, file names are 1 to 8 characters, plus an optional extension of 1 to 3 characters. In UNIX, the size of the extension, if any, is up to the user, and a file may even have two or more extensions, as in *prog.c.Z*, where *.Z* is commonly used to indicate that the file (*prog.c*) has been compressed using the Ziv-Lempel compression algorithm. Some of the more common file extensions and their meanings are shown in Fig. 6-1.

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

Figure 6-1. Some typical file extensions.

In some systems (e.g., UNIX), file extensions are just conventions and are not enforced by the operating system. A file named *file.txt* might be some kind of text file, but that name is more to remind the owner than to convey any actual information to the computer. On the other hand, a C compiler may actually insist that files it is to compile end in *.c*, and it may refuse to compile them if they do not.

Conventions like this are especially useful when the same program can handle several different kinds of files. The C compiler, for example, can be given a list of several files to compile and link together, some of them C files and some of them assembly language files. The extension then becomes essential for the compiler to tell which are C files, which are assembly files, and which are other files.

In contrast, Windows is aware of the extensions and assigns meaning to them. Users (or processes) can register extensions with the operating system and specify for each one which program "owns" that extension. When a user double clicks on a file name, the program assigned to its file extension is launched with the file as parameter. For example, double clicking on *file.doc* starts Microsoft *Word* with *file.doc* as the initial file to edit.

6.1.2 File Structure

Files can be structured in any of several ways. Three common possibilities are depicted in Fig. 6-2. The file in Fig. 6-2(a) is an unstructured sequence of bytes. In effect, the operating system does not know or care what is in the file. All it sees are bytes. Any meaning must be imposed by user-level programs. Both UNIX and Windows use this approach.

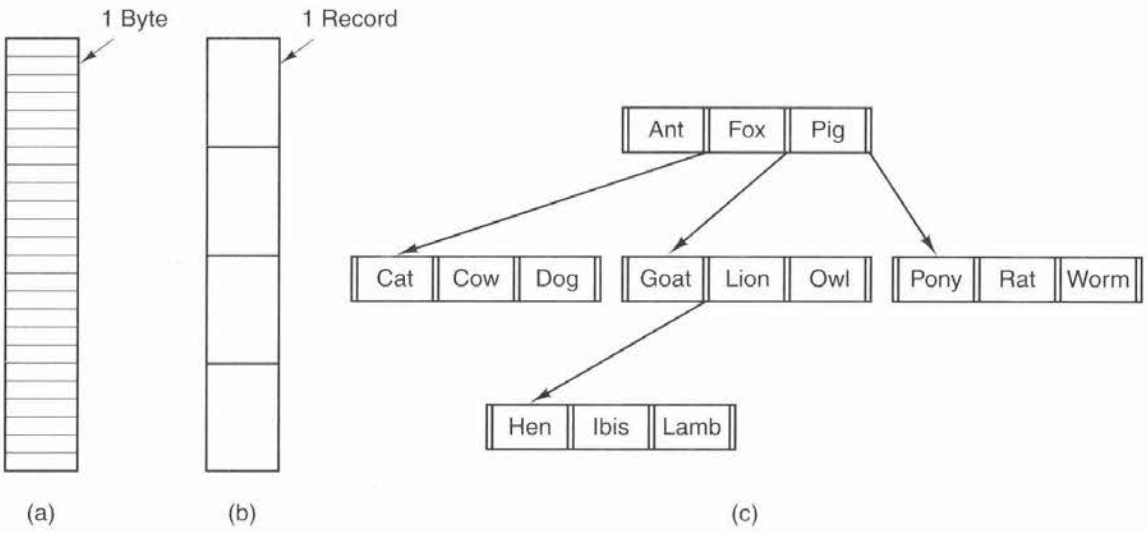


Figure 6-2. Three kinds of files. (a) Byte sequence. (b) Record sequence. (c) Tree.

Having the operating system regard files as nothing more than byte sequences provides the maximum flexibility. User programs can put anything they want in their files and name them any way that is convenient. The operating system does

not help, but it also does not get in the way. For users who want to do unusual things, the latter can be very important.

The first step up in structure is shown in Fig. 6-2(b). In this model, a file is a sequence of fixed-length records, each with some internal structure. Central to the idea of a file being a sequence of records is the idea that the read operation returns one record and the write operation overwrites or appends one record. As a historical note, in decades gone by, when the 80-column punched card was king, many (mainframe) operating systems based their file systems on files consisting of 80-character records, in effect, card images. These systems also supported files of 132-character records, which were intended for the line printer (which in those days were big chain printers having 132 columns). Programs read input in units of 80 characters and wrote it in units of 132 characters, although the final 52 could be spaces, of course. No current general-purpose system works this way.

The third kind of file structure is shown in Fig. 6-2(c). In this organization, a file consists of a tree of records, not necessarily all the same length, each containing a **key** field in a fixed position in the record. The tree is sorted on the key field, to allow rapid searching for a particular key.

The basic operation here is not to get the "next" record, although that is also possible, but to get the record with a specific key. For the zoo file of Fig. 6-2(c), one could ask the system to get the record whose key is *pony*, for example, without worrying about its exact position in the file. Furthermore, new records can be added to the file, with the operating system, and not the user, deciding where to place them. This type of file is clearly quite different from the unstructured byte streams used in UNIX and Windows but is widely used on the large mainframe computers still used in some commercial data processing.

6.1.3 File Types

Many operating systems support several types of files. UNIX and Windows, for example, have regular files and directories. UNIX also has character and block special files. **Regular files** are the ones that contain user information. All the files of Fig. 6-2 are regular files. **Directories** are system files for maintaining the structure of the file system. We will study directories below. **Character special files** are related to input/output and used to model serial I/O devices such as terminals, printers, and networks. **Block special files** are used to model disks. In this chapter we will be primarily interested in regular files.

Regular files are generally either ASCII files or binary files. ASCII files consist of lines of text. In some systems each line is terminated by a carriage return character. In others, the line feed character is used. Some systems (e.g., MS-DOS) use both. Lines need not all be of the same length.

The great advantage of ASCII files is that they can be displayed and printed as is, and they can be edited with any text editor. Furthermore, if large numbers

of programs use ASCII files for input and output, it is easy to connect the output of one program to the input of another, as in shell pipelines. (The interprocess plumbing is not any easier, but interpreting the information certainly is if a standard convention, such as ASCII, is used for expressing it.)

Other files are binary files, which just means that they are not ASCII files. Listing them on the printer gives an incomprehensible listing full of what is apparently random junk. Usually, they have some internal structure known to programs that use them.

For example, in Fig. 6-3(a) we see a simple executable binary file taken from a version of UNIX. Although technically the file is just a sequence of bytes, the operating system will only execute a file if it has the proper format. It has five sections: header, text, data, relocation bits, and symbol table. The header starts with a so-called **magic number**, identifying the file as an executable file (to prevent the accidental execution of a file not in this format). Then come the sizes of the various pieces of the file, the address at which execution starts, and some flag bits. Following the header are the text and data of the program itself. These are loaded into memory and relocated using the relocation bits. The symbol table is used for debugging.

Our second example of a binary file is an archive, also from UNIX. It consists of a collection of library procedures (modules) compiled but not linked. Each one is prefaced by a header telling its name, creation date, owner, protection code, and size. Just as with the executable file, the module headers are full of binary numbers. Copying them to the printer would produce complete gibberish.

Every operating system must recognize at least one file type: its own executable file, but some recognize more. The old TOPS-20 system (for the DECsystem 20) went so far as to examine the creation time of any file to be executed. Then it located the source file and saw if the source had been modified since the binary was made. If it had been, it automatically recompiled the source. In UNIX terms, the *make* program had been built into the shell. The file extensions were mandatory so the operating system could tell which binary program was derived from which source.

Having strongly typed files like this causes problems whenever the user does anything that the system designers did not expect. Consider, as an example, a system in which program output files have extension *.dat* (data files). If a user writes a program formatter that reads a *.c* file (C program), transforms it (e.g., by converting it to a standard indentation layout), and then writes the transformed file as output, the output file will be of type *.dat*. If the user tries to offer this to the C compiler to compile it, the system will refuse because it has the wrong extension. Attempts to copy *file.dat* to *file.c* will be rejected by the system as invalid (to protect the user against mistakes).

While this kind of "user friendliness" may help novices, it drives experienced users up the wall since they have to devote considerable effort to circumventing the operating system's idea of what is reasonable and what is not.

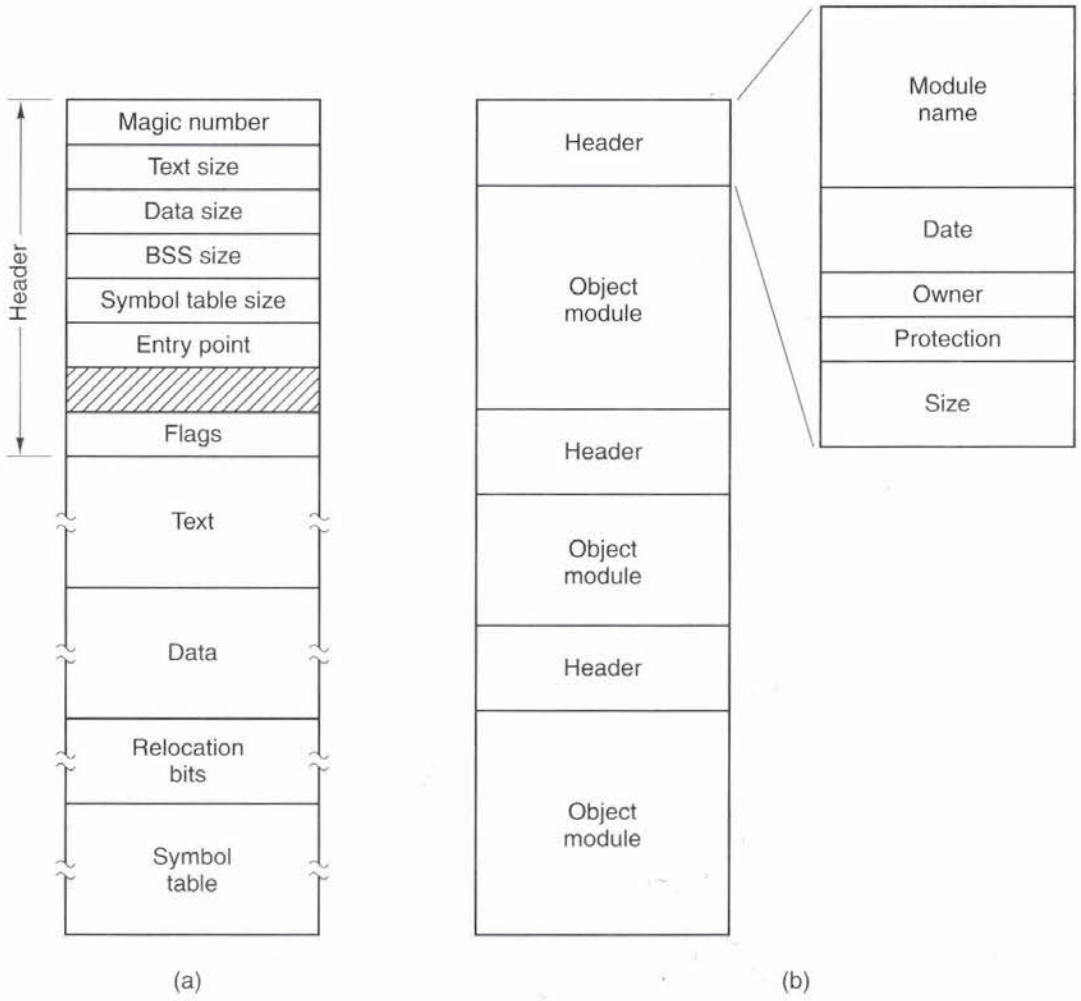


Figure 6-3. (a) An executable file. (b) An archive.

6.1.4 File Access

Early operating systems provided only one kind of file access: **sequential access**. In these systems, a process could read all the bytes or records in a file in order, starting at the beginning, but could not skip around and read them out of order. Sequential files could be rewound, however, so they could be read as often as needed. Sequential files were convenient when the storage medium was magnetic tape, rather than disk.

When disks came into use for storing files, it became possible to read the bytes or records of a file out of order, or to access records by key, rather than by position. Files whose bytes or records can be read in any order are called **random access files**. They are required by many applications.

Random access files are essential for many applications, for example, database systems. If an airline customer calls up and wants to reserve a seat on a particular flight, the reservation program must be able to access the record for that flight without having to read the records for thousands of other flights first.

Two methods are used for specifying where to start reading. In the first one, every read operation gives the position in the file to start reading at. In the second one, a special operation, seek, is provided to set the current position. After a seek, the file can be read sequentially from the now-current position.

In some older mainframe operating systems, files are classified as being either sequential or random access at the time they are created. This allows the system to use different storage techniques for the two classes. Modern operating systems do not make this distinction. All their files are automatically random access.

6.1.5 File Attributes

Every file has a name and its data. In addition, all operating systems associate other information with each file, for example, the date and time the file was created and the file's size. We will call these extra items the file's **attributes**. The list of attributes varies considerably from system to system. The table of Fig. 6-4 shows some of the possibilities, but other ones also exist. No existing system has all of these, but each one is present in some system.

The first four attributes relate to the file's protection and tell who may access it and who may not. All kinds of schemes are possible, some of which we will study later. In some systems the user must present a password to access a file, in which case the password must be one of the attributes.

The flags are bits or short fields that control or enable some specific property. Hidden files, for example, do not appear in listings of all the files. The archive flag is a bit that keeps track of whether the file has been backed up. The backup program clears it, and the operating system sets it whenever a file is changed. In this way, the backup program can tell which files need backing up. The temporary flag allows a file to be marked for automatic deletion when the process that created it terminates.

The record length, key position, and key length fields are only present in files whose records can be looked up using a key. They provide the information required to find the keys.

The various times keep track of when the file was created, most recently accessed and most recently modified. These are useful for a variety of purposes. For example, a source file that has been modified after the creation of the corresponding object file needs to be recompiled. These fields provide the necessary information.

The current size tells how big the file is at present. Some old mainframe operating systems require the maximum size to be specified when the file is created, in order to let the operating system reserve the maximum amount of storage in ad-

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
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 has last changed
Current size	Number of bytes in the file
Maximum size	Number of bytes the file may grow to

Figure 6-4. Some possible file attributes.

vance. Workstation and personal computer operating systems are clever enough to do without this feature.

6.1.6 File Operations

Files exist to store information and allow it to be retrieved later. Different systems provide different operations to allow storage and retrieval. Below is a discussion of the most common system calls relating to files.

1. Create. The file is created with no data. The purpose of the call is to announce that the file is coming and to set some of the attributes.
2. Delete. When the file is no longer needed, it has to be deleted to free up disk space. There is always a system call for this purpose.
3. Open. Before using a file, a process must open it. The purpose of the open call is to allow the system to fetch the attributes and list of disk addresses into main memory for rapid access on later calls.

4. Close. When all the accesses are finished, the attributes and disk addresses are no longer needed, so the file should be closed to free up internal table space. Many systems encourage this by imposing a maximum number of open files on processes. A disk is written in blocks, and closing a file forces writing of the file's last block, even though that block may not be entirely full yet.

5. Read. Data are read from file. Usually, the bytes come from the current position. The caller must specify how much data are needed and must also provide a buffer to put them in.

6. Write. Data are written to the file, again, usually at the current position. If the current position is the end of the file, the file's size increases. If the current position is in the middle of the file, existing data are overwritten and lost forever.

7. Append. This call is a restricted form of write. It can only add data to the end of the file. Systems that provide a minimal set of system calls do not generally have append, but many systems provide multiple ways of doing the same thing, and these systems sometimes have append.

8. Seek. For random access files, a method is needed to specify from where to take the data. One common approach is a system call, seek, that repositions the file pointer to a specific place in the file. After this call has completed, data can be read from, or written to, that position.

9. Get attributes. Processes often need to read file attributes to do their work. For example, the UNIX *make* program is commonly used to manage software development projects consisting of many source files. When *make* is called, it examines the modification times of all the source and object files and arranges for the minimum number of compilations required to bring everything up to date. To do its job, it must look at the attributes, namely, the modification times.

10. Set attributes. Some of the attributes are user settable and can be changed after the file has been created. This system call makes that possible. The protection mode information is an obvious example. Most of the flags also fall in this category.

11. Rename. It frequently happens that a user needs to change the name of an existing file. This system call makes that possible. It is not always strictly necessary, because the file can usually be copied to a new file with the new name, and the old file then deleted.

6.1.7 An Example Program Using File System Calls

In this section we will examine a simple UNIX program that copies one file from its source file to a destination file. It is listed in Fig. 6-5. The program has minimal functionality and even worse error reporting, but it gives a reasonable idea of how some of the system calls related to files work.

The program, *copyfile*, can be called, for example, by the command line

```
copyfile abc xyz
```

to copy the file *abc* to *xyz*. If *xyz* already exists, it will be overwritten. Otherwise, it will be created. The program must be called with exactly two arguments, both legal file names.

The four *#include* statements near the top of the program cause a large number of definitions and function prototypes to be included in the program. These are needed to make the program conformant to the relevant international standards, but will not concern us further. The next line is a function prototype for *main*, something required by ANSI C, but also not important for our purposes.

The first *#define* statement is a macro definition that defines the string *BUF_SIZE* as a macro that expands into the number 4096. The program will read and write in chunks of 4096 bytes. It is considered good programming practice to give names to constants like this and to use the names instead of the constants. Not only does this convention make programs easier to read, but it also makes them easier to maintain. The second *#define* statement determines who can access the output file.

The main program is called *main*, and it has two arguments, *argc*, and *argv*. These are supplied by the operating system when the program is called. The first one tells how many strings were present on the command line that invoked the program, including the program name. It should be 3. The second one is an array of pointers to the arguments. In the example call given above, the elements of this array would contain pointers to the following values:

```
argv[0] = "copyfile"
argv[1] = "abc"
argv[2] = "xyz"
```

It is via this array that the program accesses its arguments.

Five variables are declared. The first two, *in_fd* and *out_fd*, will hold the **file descriptors**, small integers returned when a file is opened. The next two, *rd_count* and *wt_count*, are the byte counts returned by the read and write system calls, respectively. The last one, *buffer*, is the buffer used to hold the data read and supply the data to be written.

The first actual statement checks *argc* to see if it is 3. If not, it exits with status code 1. Any status code other than 0 means that an error has occurred. The

```
/* 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 */
    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 */
        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 6-5. A simple program to copy a file.

status code is the only error reporting present in this program. A production version would normally print error messages as well.

Then we try to open the source file and create the destination file. If the source file is successfully opened, the system assigns a small integer to *in_fd*, to identify the file. Subsequent calls must include this integer so the system knows which file it wants. Similarly, if the destination is successfully created, *out_fd* is given a value to identify it. The second argument to *creat* sets the protection mode. If either the open or the create fails, the corresponding file descriptor is set to −1, and the program exits with an error code.

Now comes the copy loop. It starts by trying to read in 4 KB of data to *buffer*. It does this by calling the library procedure *read*, which actually invokes the read system call. The first parameter identifies the file, the second gives the buffer, and the third tells how many bytes to read. The value assigned to *rd_count* gives the number of bytes actually read. Normally, this will be 4096, except if fewer bytes are remaining in the file. When end of file is reached, it will be 0. If *rd_count* is ever zero or negative, the copying cannot continue, so the *break* statement is executed to terminate the (otherwise endless) loop.

The call to *write* outputs the buffer to the destination file. The first parameter identifies the file, the second gives the buffer, and the third tells how many bytes to write, analogous to *read*. Note that the byte count is the number of bytes actually read, not *BUF_SIZE*. This point is important because the last read will not return 4096, unless the file just happens to be a multiple of 4 KB.

When the entire file has been processed, the first call beyond the end of file will return 0 to *rd_count*, which will make it exit the loop. At this point the two files are closed and the program exits with a status indicating normal termination.

Although the Windows system calls are different from those of UNIX, the general structure of a command-line Windows program to copy a file is moderately similar to that of Fig. 6-5. We will examine the Windows 2000 calls in Chap. 11.

6.1.8 Memory-Mapped Files

Many programmers feel that accessing files as shown above is cumbersome and inconvenient, especially when compared to accessing ordinary memory. For this reason, some operating systems, starting with MULTICS, have provided a way to map files into the address space of a running process. Conceptually, we can imagine the existence of two new system calls, map and unmap. The former gives a file name and a virtual address, which causes the operating system to map the file into the address space at the virtual address.

For example, suppose that a file, *f*, of length 64 KB, is mapped into the virtual address starting at address 512K. Then any machine instruction that reads the contents of the byte at 512K gets byte 0 of the file, and so on. Similarly, a write to address 512K + 21000 modifies byte 21000 of the file. When the process

terminates, the modified file is left on the disk, just as though it had been changed by a combination of seek and write system calls.

What actually happens is that the system's internal tables are changed to make the file become the backing store for the memory region 512K to 576K. Thus a read from 512K causes a page fault, bringing in page 0 of the file. Similarly, a write to 512K + 1100 causes a page fault, bringing in the page containing that address, after which the write to memory can take place. If that page is ever evicted by the page replacement algorithm, it is written back to the appropriate place in the file. When the process finishes, all mapped, modified pages are written back to their files.

File mapping works best in a system that supports segmentation. In such a system, each file can be mapped onto its own segment so that byte *k* in the file is also byte *k* in the segment. In Fig. 6-6(a) we see a process that has two segments, text and data. Suppose that this process copies files, like the program of Fig. 6-5. First it maps the source file, say, *abc*, onto a segment. Then it creates an empty segment and maps it onto the destination file, *xyz* in our example. These operations give the situation shown in Fig. 6-6(b).

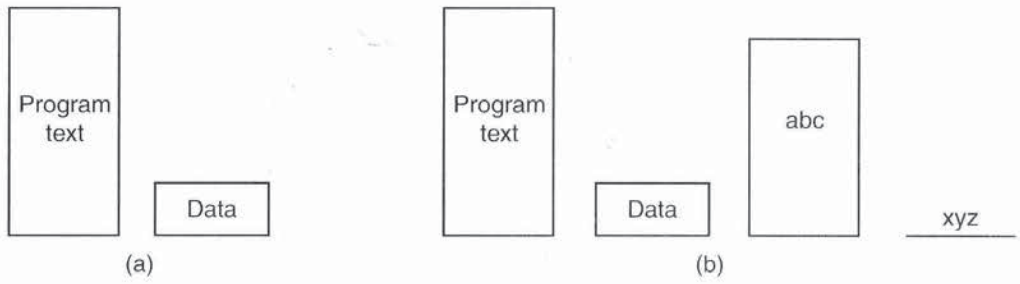


Figure 6-6. (a) A segmented process before mapping files into its address space. (b) The process after mapping an existing file *abc* into one segment and creating a new segment for file *xyz*.

At this point the process can copy the source segment into the destination segment using an ordinary copy loop. No read or write system calls are needed. When it is all done, it can execute the unmap system call to remove the files from the address space and then exit. The output file, *xyz*, will now exist, as though it had been created in the conventional way.

Although file mapping eliminates the need for I/O and thus makes programming easier, it introduces a few problems of its own. First, it is hard for the system to know the exact length of the output file, *xyz*, in our example. It can easily tell the number of the highest page written, but it has no way of knowing how many bytes in that page were written. Suppose that the program only uses page 0, and after execution all the bytes are still 0 (their initial value). Maybe *xyz* is a file consisting of 10 zeros. Maybe it is a file consisting of 100 zeros. Maybe it is a file consisting of 1000 zeros. Who knows? The operating system cannot tell. All it can do is create a file whose length is equal to the page size.

A second problem can (potentially) occur if a file is mapped in by one process and opened for conventional reading by another. If the first process modifies a page, that change will not be reflected in the file on disk until the page is evicted. The system has to take great care to make sure the two processes do not see inconsistent versions of the file.

A third problem with mapping is that a file may be larger than a segment, or even larger than the entire virtual address space. The only way out is to arrange the map system call to be able to map a portion of a file, rather than the entire file. Although this works, it is clearly less satisfactory than mapping the entire file.

6.2 DIRECTORIES

To keep track of files, file systems normally have **directories** or **folders**, which, in many systems, are themselves files. In this section we will discuss directories, their organization, their properties, and the operations that can be performed on them.

6.2.1 Single-Level Directory Systems

The simplest form of directory system is having one directory containing all the files. Sometimes it is called the **root directory**, but since it is the only one, the name does not matter much. On early personal computers, this system was common, in part because there was only one user. Interestingly enough, the world's first supercomputer, the CDC 6600, also had only a single directory for all files, even though it was used by many users at once. This decision was no doubt made to keep the software design simple.

An example of a system with one directory is given in Fig. 6-7. Here the directory contains four files. The file *owners* are shown in the figure, not the file *names* (because the owners are important to the point we are about to make). The advantages of this scheme are its simplicity and the ability to locate files quickly—there is only one place to look, after all.

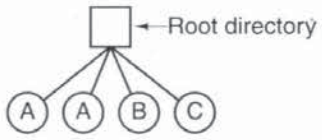


Figure 6-7. A single-level directory system containing four files, owned by three different people, *A*, *B*, and *C*.

The problem with having only one directory in a system with multiple users is that different users may accidentally use the same names for their files. For example, if user *A* creates a file called *mailbox*, and then later user *B* also creates a file called *mailbox*, *B*'s file will overwrite *A*'s file. Consequently, this scheme is

not used on multiuser systems any more, but could be used on a small embedded system, for example, a system in a car that was designed to store user profiles for a small number of drivers.

6.2.2 Two-level Directory Systems

To avoid conflicts caused by different users choosing the same file name for their own files, the next step up is giving each user a private directory. In that way, names chosen by one user do not interfere with names chosen by a different user and there is no problem caused by the same name occurring in two or more directories. This design leads to the system of Fig. 6-8. This design could be used, for example, on a multiuser computer or on a simple network of personal computers that shared a common file server over a local area network.

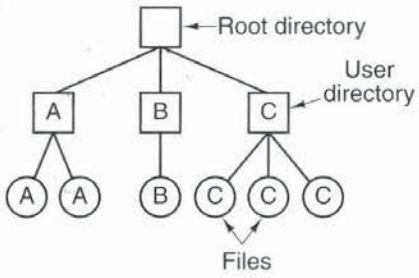


Figure 6-8. A two-level directory system. The letters indicate the owners of the directories and files.

Implicit in this design is that when a user tries to open a file, the system knows which user it is in order to know which directory to search. As a consequence, some kind of login procedure is needed, in which the user specifies a login name or identification, something not required with a single-level directory system.

When this system is implemented in its most basic form, users can only access files in their own directories. However, a slight extension is to allow users to access other users' files by providing some indication of whose file is to be opened. Thus, for example,

```
open("x")
```

might be the call to open a file called *x* in the user's directory, and

```
open("nancy/x")
```

might be the call to open a file *x* in the directory of another user, Nancy.

One situation in which users need to access files other than their own is to execute system binary programs. Having copies of all the utility programs present in each directory clearly is inefficient. At the very least, there is a need for a system directory with the executable binary programs.

6.2.3 Hierarchical Directory Systems

The two-level hierarchy eliminates name conflicts among users but is not satisfactory for users with a large number of files. Even on a single-user personal computer, it is inconvenient. It is quite common for users to want to group their files together in logical ways. A professor, for example, might have a collection of files that together form a book that he is writing for one course, a second collection of files containing student programs submitted for another course, a third group of files containing the code of an advanced compiler-writing system he is building, a fourth group of files containing grant proposals, as well as other files for electronic mail, minutes of meetings, papers he is writing, games, and so on. Some way is needed to group these files together in flexible ways chosen by the user.

What is needed is a general hierarchy (i.e., a tree of directories). With this approach, each user can have as many directories as are needed so that files can be grouped together in natural ways. This approach is shown in Fig. 6-9. Here, the directories *A*, *B*, and *C* contained in the root directory each belong to a different user, two of whom have created subdirectories for projects they are working on.

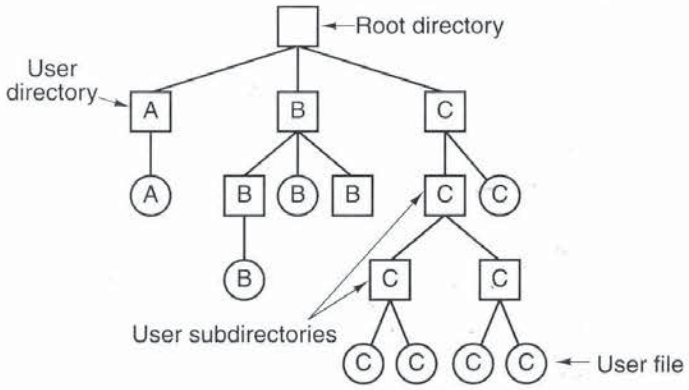


Figure 6-9. A hierarchical directory system.

The ability for users to create an arbitrary number of subdirectories provides a powerful structuring tool for users to organize their work. For this reason, nearly all modern file systems are organized in this manner.

6.2.4 Path Names

When the file system is organized as a directory tree, some way is needed for specifying file names. Two different methods are commonly used. In the first method, each file is given an **absolute path name** consisting of the path from the root directory to the file. As an example, the path */usr/ast/mailbox* means that the

root directory contains a subdirectory *usr*, which in turn contains a subdirectory *ast*, which contains the file *mailbox*. Absolute path names always start at the root directory and are unique. In UNIX the components of the path are separated by /. In Windows the separator is \. In MULTICS it was >. Thus the same path name would be written as follows in these three systems:

Windows	\usr\ast\mailbox
UNIX	/usr/ast/mailbox
MULTICS	>usr>ast>mailbox

No matter which character is used, if the first character of the path name is the separator, then the path is absolute.

The other kind of name is the **relative path name**. This is used in conjunction with the concept of the **working directory** (also called the **current directory**). A user can designate one directory as the current working directory, in which case all path names not beginning at the root directory are taken relative to the working directory. For example, if the current working directory is */usr/ast*, then the file whose absolute path is */usr/ast/mailbox* can be referenced simply as *mailbox*. In other words, the UNIX command

```
cp /usr/ast/mailbox /usr/ast/mailbox.bak
```

and the command

```
cp mailbox mailbox.bak
```

do exactly the same thing if the working directory is */usr/ast*. The relative form is often more convenient, but it does the same thing as the absolute form.

Some programs need to access a specific file without regard to what the working directory is. In that case, they should always use absolute path names. For example, a spelling checker might need to read */usr/lib/dictionary* to do its work. It should use the full, absolute path name in this case because it does not know what the working directory will be when it is called. The absolute path name will always work, no matter what the working directory is.

Of course, if the spelling checker needs a large number of files from */usr/lib*, an alternative approach is for it to issue a system call to change its working directory to */usr/lib*, and then use just *dictionary* as the first parameter to open. By explicitly changing the working directory, it knows for sure where it is in the directory tree, so it can then use relative paths.

Each process has its own working directory, so when a process changes its working directory and later exits, no other processes are affected and no traces of the change are left behind in the file system. In this way it is always perfectly safe for a process to change its working directory whenever that is convenient. On the other hand, if a *library procedure* changes the working directory and does not change back to where it was when it is finished, the rest of the program may not work since its assumption about where it is may now suddenly be invalid. For

this reason, library procedures rarely change the working directory, and when they must, they always change it back again before returning.

Most operating systems that support a hierarchical directory system have two special entries in every directory, "." and "..", generally pronounced "dot" and "dotdot." Dot refers to the current directory; dotdot refers to its parent. To see how these are used, consider the UNIX file tree of Fig. 6-10. A certain process has */usr/ast* as its working directory. It can use .. to go up the tree. For example, it can copy the file */usr/lib/dictionary* to its own directory using the command

```
cp ../lib/dictionary .
```

The first path instructs the system to go upward (to the *usr* directory), then to go down to the directory *lib* to find the file *dictionary*.

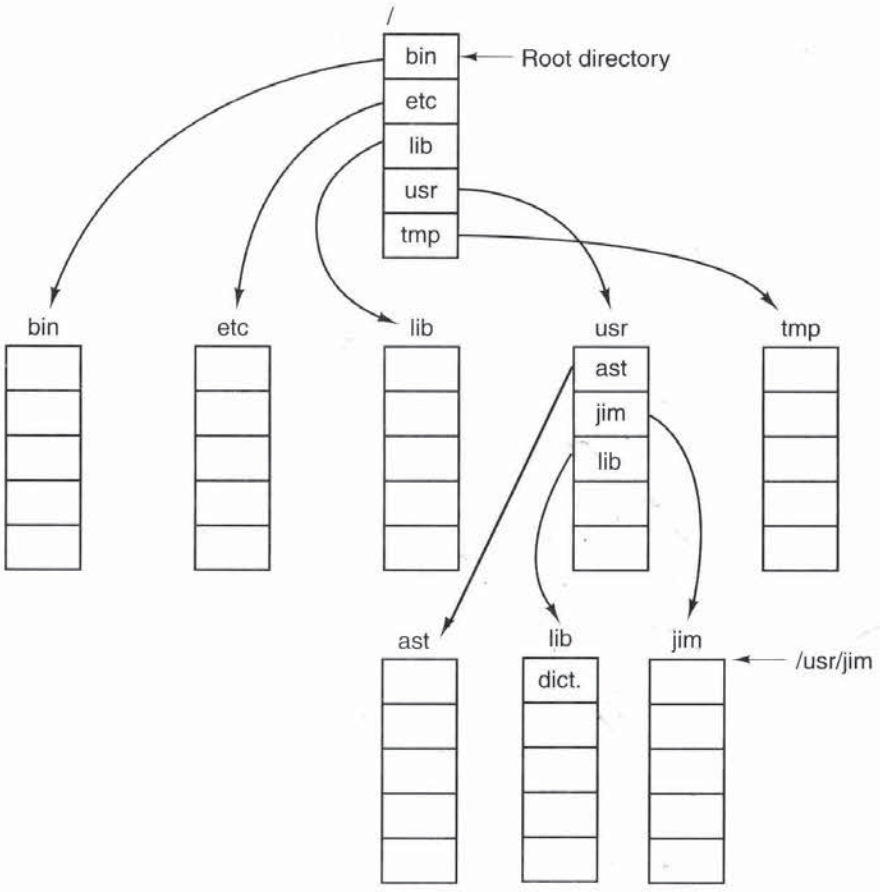


Figure 6-10. A UNIX directory tree.

The second argument (dot) names the current directory. When the *cp* command gets a directory name (including dot) as its last argument, it copies all the files there. Of course, a more normal way to do the copy would be to type

```
cp /usr/lib/dictionary .
```

Here the use of dot saves the user the trouble of typing *dictionary* a second time. Nevertheless, typing

```
cp /usr/lib/dictionary dictionary
```

also works fine, as does

```
cp /usr/lib/dictionary /usr/ast/dictionary
```

All of these do exactly the same thing.

6.2.5 Directory Operations

The allowed system calls for managing directories exhibit more variation from system to system than system calls for files. To give an impression of what they are and how they work, we will give a sample (taken from UNIX).

1. Create. A directory is created. It is empty except for dot and dotdot, which are put there automatically by the system (or in a few cases, by the *mkdir* program).
2. Delete. A directory is deleted. Only an empty directory can be deleted. A directory containing only dot and dotdot is considered empty as these cannot usually be deleted.
3. Opendir. Directories can be read. For example, to list all the files in a directory, a listing program opens the directory to read out the names of all the files it contains. Before a directory can be read, it must be opened, analogous to opening and reading a file.
4. Closedir. When a directory has been read, it should be closed to free up internal table space.
5. Readdir. This call returns the next entry in an open directory. Formerly, it was possible to read directories using the usual read system call, but that approach has the disadvantage of forcing the programmer to know and deal with the internal structure of directories. In contrast, readdir always returns one entry in a standard format, no matter which of the possible directory structures is being used.
6. Rename. In many respects, directories are just like files and can be renamed the same way files can be.
7. Link. Linking is a technique that allows a file to appear in more than one directory. This system call specifies an existing file and a path name, and creates a link from the existing file to the name specified by the path. In this way, the same file may appear in multiple directories. A link of this kind, which increments the counter in the file's i-node (to keep track of the number of directory entries containing the file), is sometimes called a **hard link**.

8. Unlink. A directory entry is removed. If the file being unlinked is only present in one directory (the normal case), it is removed from the file system. If it is present in multiple directories, only the path name specified is removed. The others remain. In UNIX, the system call for deleting files (discussed earlier) is, in fact, unlink.

The above list gives the most important calls, but there are a few others as well, for example, for managing the protection information associated with a directory.

6.3 FILE SYSTEM IMPLEMENTATION

Now it is time to turn from the user's view of the file system to the implementor's view. Users are concerned with how files are named, what operations are allowed on them, what the directory tree looks like, and similar interface issues. Implementors are interested in how files and directories are stored, how disk space is managed, and how to make everything work efficiently and reliably. In the following sections we will examine a number of these areas to see what the issues and trade-offs are.

6.3.1 File System Layout

File systems are stored on disks. Most disks can be divided up into one or more partitions, with independent file systems on each partition. Sector 0 of the disk is called the **MBR** (**Master Boot Record**) and is used to boot the computer. The end of the MBR contains the partition table. This table gives the starting and ending addresses of each partition. One of the partitions in the table is marked as active. When the computer is booted, the BIOS reads in and executes the MBR. The first thing the MBR program does is locate the active partition, read in its first block, called the **boot block**, and execute it. The program in the boot block loads the operating system contained in that partition. For uniformity, every partition starts with a boot block, even if it does not contain a bootable operating system. Besides, it might contain one in the future, so reserving a boot block is a good idea anyway.

Other than starting with a boot block, the layout of a disk partition varies strongly from file system to file system. Often the file system will contain some of the items shown in Fig. 6-11. The first one is the **superblock**. It contains all the key parameters about the file system and is read into memory when the computer is booted or the file system is first touched. Typical information in the superblock includes a magic number to identify the file system type, the number of blocks in the file system, and other key administrative information.

Next might come information about free blocks in the file system, for example in the form of a bitmap or a list of pointers. This might be followed by the i-

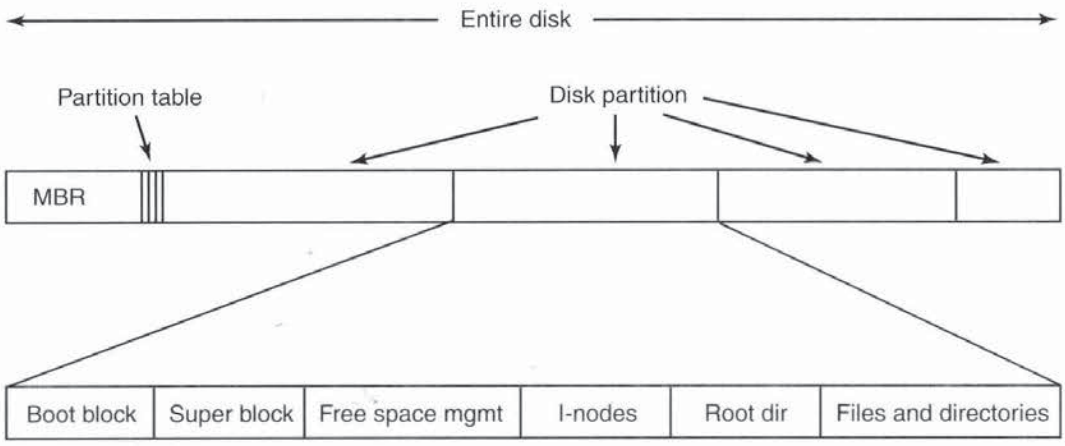


Figure 6-11. A possible file system layout.

nodes, an array of data structures, one per file, telling all about the file. After that might come the root directory, which contains the top of the file system tree. Finally, the remainder of the disk typically contains all the other directories and files.

6.3.2 Implementing Files

Probably the most important issue in implementing file storage is keeping track of which disk blocks go with which file. Various methods are used in different operating systems. In this section, we will examine a few of them.

Contiguous Allocation

The simplest allocation scheme is to store each file as a contiguous run of disk blocks. Thus on a disk with 1-KB blocks, a 50-KB file would be allocated 50 consecutive blocks. With 2-KB blocks, it would be allocated 25 consecutive blocks.

We see an example of contiguous storage allocation in Fig. 6-12(a). Here the first 40 disk blocks are shown, starting with block 0 on the left. Initially, the disk was empty. Then a file *A*, of length four blocks was written to disk starting at the beginning (block 0). After that a six-block file, *B*, was written starting right after the end of file *A*. Note that each file begins at the start of a new block, so that if file *A* was really 3½ blocks, some space is wasted at the end of the last block. In the figure, a total of seven files are shown, each one starting at the block following the end of the previous one. Shading is used just to make it easier to tell the files apart.

Contiguous disk space allocation has two significant advantages. First, it is simple to implement because keeping track of where a file's blocks are is reduced

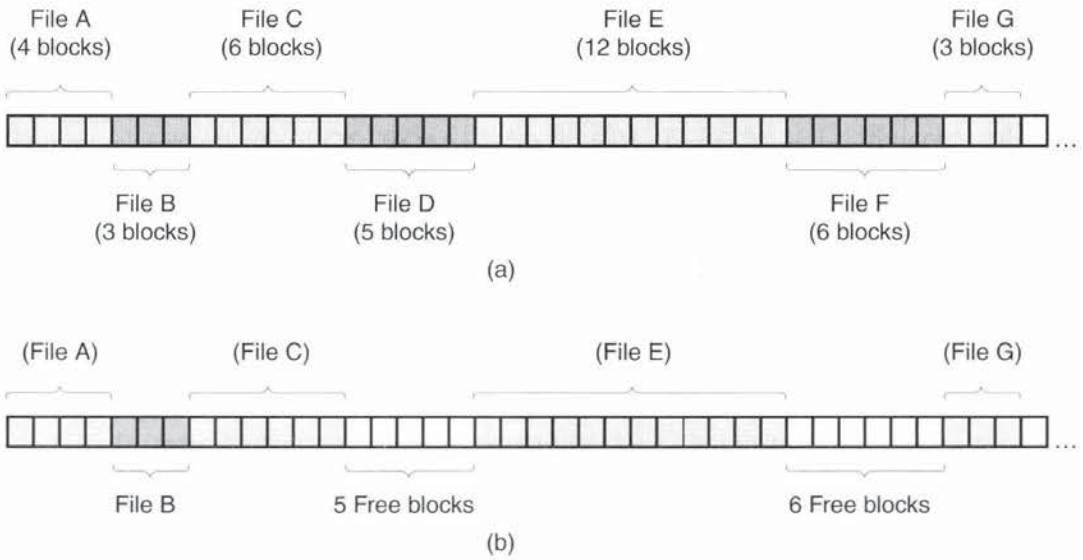


Figure 6-12. (a) Contiguous allocation of disk space for seven files. (b) The state of the disk after files *D* and *F* have been removed.

to remembering two numbers: the disk address of the first block and the number of blocks in the file. Given the number of the first block, the number of any other block can be found by a simple addition.

Second, the read performance is excellent because the entire file can be read from the disk in a single operation. Only one seek is needed (to the first block). After that, no more seeks or rotational delays are needed so data come in at the full bandwidth of the disk. Thus contiguous allocation is simple to implement and has high performance.

Unfortunately, contiguous allocation also has a significant drawback: in time, the disk becomes fragmented. To see how this comes about, examine Fig. 6-12(b). Here two files, *D* and F have been removed. When a file is removed, its blocks are freed, leaving a run of free blocks on the disk. The disk is not compacted on the spot to squeeze out the hole since that would involve copying all the blocks following the hole, potentially millions of blocks. As a result, the disk ultimately consists of files and holes, as illustrated in the figure.

Initially, this fragmentation is not a problem since each new file can be written at the end of disk, following the previous one. However, eventually the disk will fill up and it will become necessary to either compact the disk, which is prohibitively expensive, or to reuse the free space in the holes. Reusing the space requires maintaining a list of holes, which is doable. However, when a new file is to be created, it is necessary to know its final size in order to choose a hole of the correct size to place it in.

Imagine the consequences of such a design. The user starts a text editor or word processor in order to type a document. The first thing the program asks is

how many bytes the final document will be. The question must be answered or the program will not continue. If the number given ultimately proves too small, the program has to terminate prematurely because the disk hole is full and there is no place to put the rest of the file. If the user tries to avoid this problem by giving an unrealistically large number as the final size, say, 100 MB, the editor may be unable to find such a large hole and announce that the file cannot be created. Of course, the user would be free to start the program again and say 50 MB this time, and so on until a suitable hole was located. Still, this scheme is not likely to lead to happy users.

However, there is one situation in which contiguous allocation is feasible and, in fact, widely used: on CD-ROMs. Here all the file sizes are known in advance and will never change during subsequent use of the CD-ROM file system. We will study the most common CD-ROM file system later in this chapter.

As we mentioned in Chap. 1, history often repeats itself in computer science as new generations of technology occur. Contiguous allocation was actually used on magnetic disk file systems years ago due to its simplicity and high performance (user friendliness did not count for much then). Then the idea was dropped due to the nuisance of having to specify final file size at file creation time. But with the advent of CD-ROMs, DVDs, and other write-once optical media, suddenly contiguous files are a good idea again. It is thus important to study old systems and ideas that were conceptually clean and simple because they may be applicable to future systems in surprising ways.

Linked List Allocation

The second method for storing files is to keep each one as a linked list of disk blocks, as shown in Fig. 6-13. The first word of each block is used as a pointer to the next one. The rest of the block is for data.

Unlike contiguous allocation, every disk block can be used in this method. No space is lost to disk fragmentation (except for internal fragmentation in the last block). Also, it is sufficient for the directory entry to merely store the disk address of the first block. The rest can be found starting there.

On the other hand, although reading a file sequentially is straightforward, random access is extremely slow. To get to block n, the operating system has to start at the beginning and read the $n - 1$ blocks prior to it, one at a time. Clearly, doing so many reads will be painfully slow.

Also, the amount of data storage in a block is no longer a power of two because the pointer takes up a few bytes. While not fatal, having a peculiar size is less efficient because many programs read and write in blocks whose size is a power of two. With the first few bytes of each block occupied to a pointer to the next block, reads of the full block size require acquiring and concatenating information from two disk blocks, which generates extra overhead due to the copying.

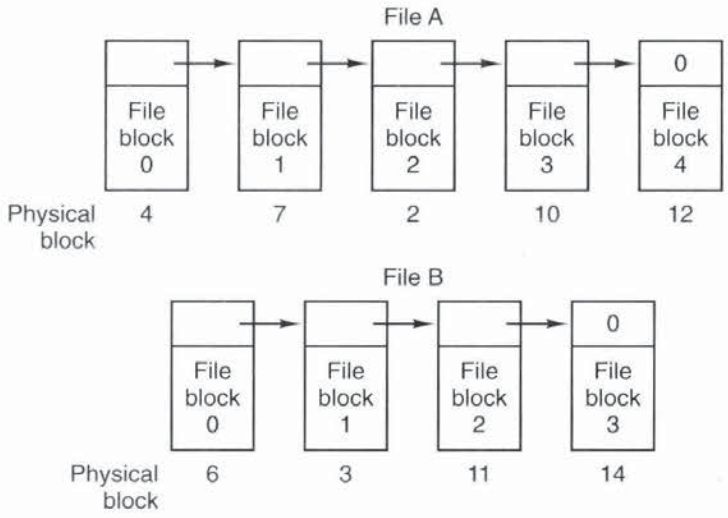


Figure 6-13. Storing a file as a linked list of disk blocks.

Linked List Allocation Using a Table in Memory

Both disadvantages of the linked list allocation can be eliminated by taking the pointer word from each disk block and putting it in a table in memory. Figure 6-14 shows what the table looks like for the example of Fig. 6-13. In both figures, we have two files. File *A* uses disk blocks 4, 7, 2, 10, and 12, in that order, and file *B* uses disk blocks 6, 3, 11, and 14, in that order. Using the table of Fig. 6-14, we can start with block 4 and follow the chain all the way to the end. The same can be done starting with block 6. Both chains are terminated with a special marker (e.g., −1) that is not a valid block number. Such a table in main memory is called a **FAT** (**File Allocation Table**).

Using this organization, the entire block is available for data. Furthermore, random access is much easier. Although the chain must still be followed to find a given offset within the file, the chain is entirely in memory, so it can be followed without making any disk references. Like the previous method, it is sufficient for the directory entry to keep a single integer (the starting block number) and still be able to locate all the blocks, no matter how large the file is.

The primary disadvantage of this method is that the entire table must be in memory all the time to make it work. With a 20-GB disk and a 1-KB block size, the table needs 20 million entries, one for each of the 20 million disk blocks. Each entry has to be a minimum of 3 bytes. For speed in lookup, they should be 4 bytes. Thus the table will take up 60 MB or 80 MB of main memory all the time, depending on whether the system is optimized for space or time. Conceivably the table could be put in pageable memory, but it would still occupy a great deal of virtual memory and disk space as well as generating extra paging traffic.

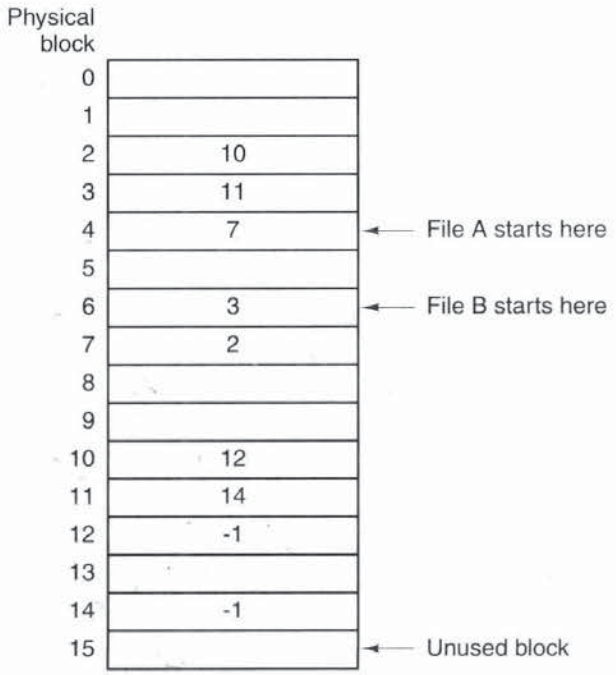


Figure 6-14. Linked list allocation using a file allocation table in main memory.

I-nodes

Our last method for keeping track of which blocks belong to which file is to associate with each file a data structure called an **i-node** (**index-node**), which lists the attributes and disk addresses of the file's blocks. A simple example is depicted in Fig. 6-15. Given the i-node, it is then possible to find all the blocks of the file. The big advantage of this scheme over linked files using an in-memory table is that the i-node need only be in memory when the corresponding file is open. If each i-node occupies n bytes and a maximum of k files may be open at once, the total memory occupied by the array holding the i-nodes for the open files is only kn bytes. Only this much space need be reserved in advance.

This array is usually far smaller than the space occupied by the file table described in the previous section. The reason is simple. The table for holding the linked list of all disk blocks is proportional in size to the disk itself. If the disk has n blocks, the table needs n entries. As disks grow larger, this table grows linearly with them. In contrast, the i-node scheme requires an array in memory whose size is proportional to the maximum number of files that may be open at once. It does not matter if the disk is 1 GB or 10 GB or 100 GB.

One problem with i-nodes is that if each one has room for a fixed number of disk addresses, what happens when a file grows beyond this limit? One solution is to reserve the last disk address not for a data block, but instead for the address

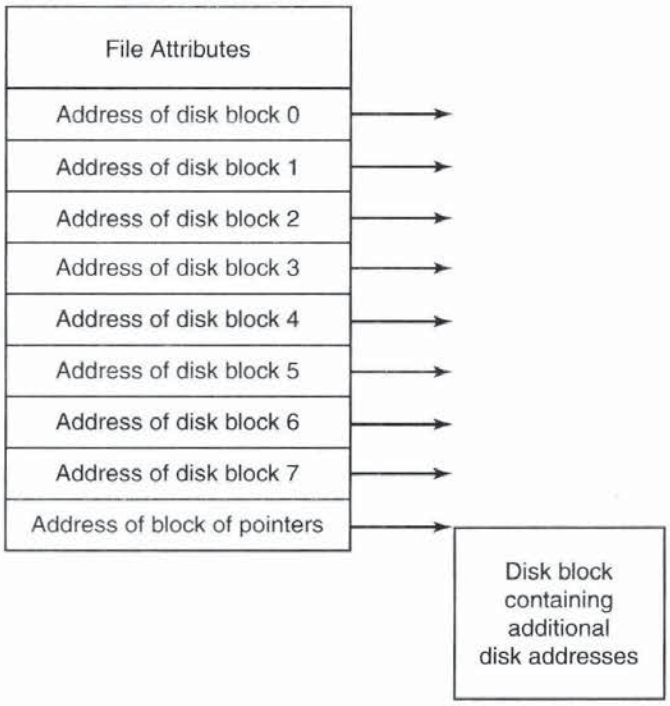


Figure 6-15. An example i-node.

of a block containing more disk block addresses, as shown in Fig. 6-15. Even more advanced would be two or more such blocks containing disk addresses or even disk blocks pointing to other disk blocks full of addresses. We will come back to i-nodes when studying UNIX later.

6.3.3 Implementing Directories

Before a file can be read, it must be opened. When a file is opened, the operating system uses the path name supplied by the user to locate the directory entry. The directory entry provides the information needed to find the disk blocks. Depending on the system, this information may be the disk address of the entire file (contiguous allocation), the number of the first block (both linked list schemes), or the number of the i-node. In all cases, the main function of the directory system is to map the ASCII name of the file onto the information needed to locate the data.

A closely related issue is where the attributes should be stored. Every file system maintains file attributes, such as each file's owner and creation time, and they must be stored somewhere. One obvious possibility is to store them directly in the directory entry. Many systems do precisely that. This option is shown in Fig. 6-16(a) In this simple design, a directory consists of a list of fixed-size

entries, one per file, containing a (fixed-length) file name, a structure of the file attributes, and one or more disk addresses (up to some maximum) telling where the disk blocks are.

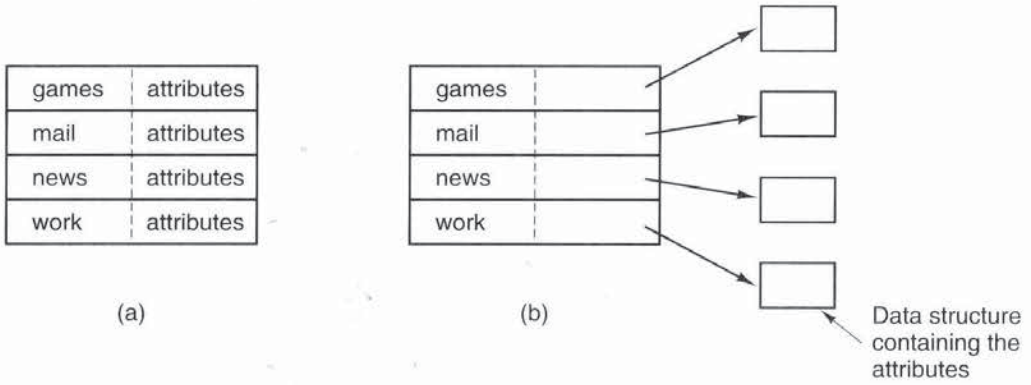


Figure 6-16. (a) A simple directory containing fixed-size entries with the disk addresses and attributes in the directory entry. (b) A directory in which each entry just refers to an i-node.

For systems that use i-nodes, another possibility for storing the attributes is in the i-nodes, rather than in the directory entries. In that case, the directory entry can be shorter: just a file name and an i-node number. This approach is illustrated in Fig. 6-16(b). As we shall see later, this method has certain advantages over putting them in the directory entry. The two approaches shown in Fig. 6-16 correspond to MS-DOS/Windows and UNIX, respectively, as we will see later in this chapter.

So far we have assumed that files have short, fixed-length names. In MS-DOS files have a 1-8 character base name and an optional extension of 1-3 characters. In UNIX Version 7, file names were 1-14 characters, including any extensions. However, nearly all modern operating systems support longer, variable-length file names. How can these be implemented?

The simplest approach is to set a limit on file name length, typically 255 characters, and then use one of the designs of Fig. 6-16 with 255 characters reserved for each file name. This approach is simple, but wastes a great deal of directory space, since few files have such long names. For efficiency reasons, a different structure is desirable.

One alternative is to give up the idea that all directory entries are the same size. With this method, each directory entry contains a fixed portion, typically starting with the length of the entry, and then followed by data with a fixed format, usually including the owner, creation time, protection information, and other attributes. This fixed-length header is followed by the actual file name, however long it may be, as shown in Fig. 6-17(a) in big-endian format (e.g., SPARC). In this example we have three files, *project-budget*, *personnel*, and *foo*. Each file name is terminated by a special character (usually 0), which is represented in the

figure by a box with a cross in it. To allow each directory entry to begin on a word boundary, each file name is filled out to an integral number of words, shown by shaded boxes in the figure.

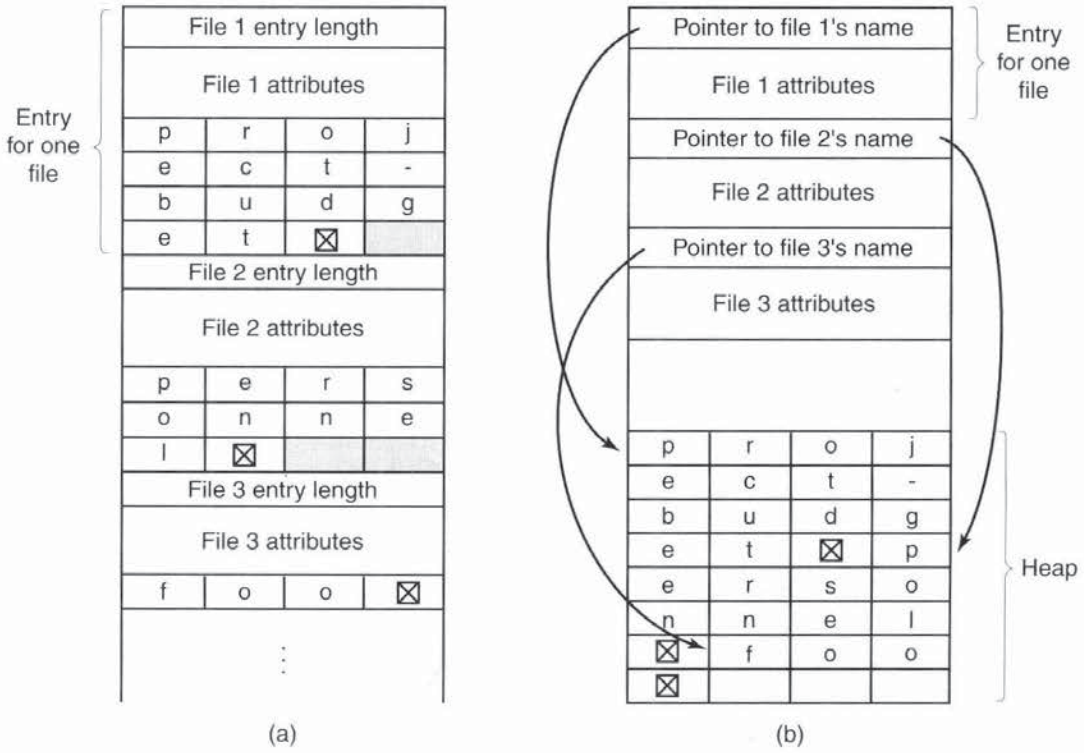


Figure 6-17. Two ways of handling long file names in a directory. (a) In-line. (b) In a heap.

A disadvantage of this method is that when a file is removed, a variable-sized gap is introduced into the directory into which the next file to be entered may not fit. This problem is the same one we saw with contiguous disk files, only now compacting the directory is feasible because it is entirely in memory. Another problem is that a single directory entry may span multiple pages, so a page fault may occur while reading a file name.

Another way to handle variable-length names is to make the directory entries themselves all fixed length and keep the file names together in a heap at the end of the directory, as shown in Fig. 6-17(b). This method has the advantage that when an entry is removed, the next file entered will always fit there. Of course, the heap must be managed and page faults can still occur while processing file names. One minor win here is that there is no longer any real need for file names to begin at word boundaries, so no filler characters are needed after file names in Fig. 6-17(b) and they are in Fig. 6-17(a).

In all of the designs so far, directories are searched linearly from beginning to end when a file name has to be looked up. For extremely long directories, linear

searching can be slow. One way to speed up the search is to use a hash table in each directory. Call the size of the table n. To enter a file name, the name is hashed onto a value between 0 and $n - 1$, for example, by dividing it by n and taking the remainder. Alternatively, the words comprising the file name can be added up and this quantity divided by n, or something similar.

Either way, the table entry corresponding to the hash code is inspected. If it is unused, a pointer is placed there to the file entry. File entries follow the hash table. If that slot is already in use, a linked list is constructed, headed at that table entry and threading through all entries with the same hash value.

Looking up a file follows the same procedure. The file name is hashed to select a hash table entry. All the entries on the chain headed at that slot are checked to see if the file name is present. If the name is not on the chain, the file is not present in the directory.

Using a hash table has the advantage of much faster lookup, but the disadvantage of a more complex administration. It is only really a serious candidate in systems where it is expected that directories will routinely contain hundreds or thousands of files.

A completely different way to speed up searching large directories is to cache the results of searches. Before starting a search, a check is first made to see if the file name is in the cache. If so, it can be located quickly, avoiding a long lookup. Of course, caching only works if a relatively small number of files comprise the majority of the lookups.

6.3.4 Shared Files

When several users are working together on a project, they often need to share files. As a result, it is often convenient for a shared file to appear simultaneously in different directories belonging to different users. Figure 6-18 shows the file system of Fig. 6-9 again, only with one of C's files now present in one of B's directories as well. The connection between B's directory and the shared file is called a **link**. The file system itself is now a **Directed Acyclic Graph**, or **DAG**, rather than a tree.

Sharing files is convenient, but it also introduces some problems. To start with, if directories really do contain disk addresses, then a copy of the disk addresses will have to be made in B's directory when the file is linked. If either B or C subsequently appends to the file, the new blocks will be listed only in the directory of the user doing the append. The changes will not be visible to the other user, thus defeating the purpose of sharing.

This problem can be solved in two ways. In the first solution, disk blocks are not listed in directories, but in a little data structure associated with the file itself. The directories would then point just to the little data structure. This is the approach used in UNIX (where the little data structure is the i-node).

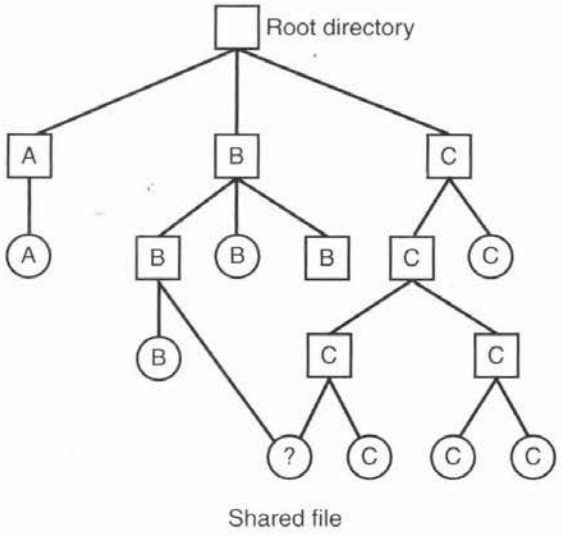


Figure 6-18. File system containing a shared file.

In the second solution, *B* links to one of *C*'s files by having the system create a new file, of type LINK, and entering that file in *B's* directory. The new file contains just the path name of the file to which it is linked. When *B* reads from the linked file, the operating system sees that the file being read from is of type LINK, looks up the name of the file, and reads that file. This approach is called **symbolic linking**.

Each of these methods has its drawbacks. In the first method, at the moment that *B* links to the shared file, the i-node records the file's owner as *C*. Creating a link does not change the ownership (see Fig. 6-19), but it does increase the link count in the i-node, so the system knows how many directory entries currently point to the file.

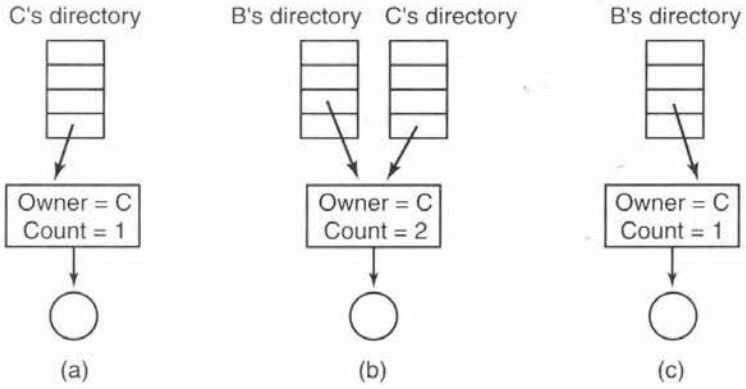


Figure 6-19. (a) Situation prior to linking. (b) After the link is created. (c) After the original owner removes the file.

If *C* subsequently tries to remove the file, the system is faced with a problem. If it removes the file and clears the i-node, *B* will have a directory entry pointing to an invalid i-node. If the i-node is later reassigned to another file, *B*'s link will point to the wrong file. The system can see from the count in the i-node that the file is still in use, but there is no way for it to find all the directory entries for the file, in order to erase them. Pointers to the directories cannot be stored in the i-node because there can be an unlimited number of directories.

The only thing to do is remove *C*'s directory entry, but leave the i-node intact, with count set to 1, as shown in Fig. 6-19(c). We now have a situation in which *B* is the only user having a directory entry for a file owned by *C*. If the system does accounting or has quotas, *C* will continue to be billed for the file until *B* decides to remove it, if ever, at which time the count goes to 0 and the file is deleted.

With symbolic links this problem does not arise because only the true owner has a pointer to the i-node. Users who have linked to the file just have path names, not i-node pointers. When the *owner* removes the file, it is destroyed. Subsequent attempts to use the file via a symbolic link will fail when the system is unable to locate the file. Removing a symbolic link does not affect the file at all.

The problem with symbolic links is the extra overhead required. The file containing the path must be read, then the path must be parsed and followed, component by component, until the i-node is reached. All of this activity may require a considerable number of extra disk accesses. Furthermore, an extra i-node is needed for each symbolic link, as is an extra disk block to store the path, although if the path name is short, the system could store it in the i-node itself, as an optimization. Symbolic links have the advantage that they can be used to link to files on machines anywhere in the world, by simply providing the network address of the machine where the file resides in addition to its path on that machine.

There is also another problem introduced by links, symbolic or otherwise. When links are allowed, files can have two or more paths. Programs that start at a given directory and find all the files in that directory and its subdirectories will locate a linked file multiple times. For example, a program that dumps all the files in a directory and its subdirectories onto a tape may make multiple copies of a linked file. Furthermore, if the tape is then read into another machine, unless the dump program is clever, the linked file will be copied twice onto the disk, instead of being linked.

6.3.5 Disk Space Management

Files are normally stored on disk, so management of disk space is a major concern to file system designers. Two general strategies are possible for storing an *n* byte file: *n* consecutive bytes of disk space are allocated, or the file is split up into a number of (not necessarily) contiguous blocks. The same trade-off is present in memory management systems between pure segmentation and paging.

As we have seen, storing a file as a contiguous sequence of bytes has the obvious problem that if a file grows, it will probably have to be moved on the disk. The same problem holds for segments in memory, except that moving a segment in memory is a relatively fast operation compared to moving a file from one disk position to another. For this reason, nearly all file systems chop files up into fixed-size blocks that need not be adjacent.

Block Size

Once it has been decided to store files in fixed-size blocks, the question arises of how big the block should be. Given the way disks are organized, the sector, the track and the cylinder are obvious candidates for the unit of allocation (although these are all device dependent, which is a minus). In a paging system, the page size is also a major contender.

Having a large allocation unit, such as a cylinder, means that every file, even a 1-byte file, ties up an entire cylinder. Studies (Mullender and Tanenbaum, 1984) have shown that the median file size in UNIX environments is about 1 KB, so allocating a 32-KB block for each file would waste 31/32 or 97 percent of the total disk space.

On the other hand, using a small allocation unit means that each file will consist of many blocks. Reading each block normally requires a seek and a rotational delay, so reading a file consisting of many small blocks will be slow.

As an example, consider a disk with 131,072 bytes per track, a rotation time of 8.33 msec, and an average seek time of 10 msec. The time in milliseconds to read a block of k bytes is then the sum of the seek, rotational delay, and transfer times:

$$10 + 4.165 + (k/131072) \times 8.33$$

The solid curve of Fig. 6-20 shows the data rate for such a disk as a function of block size. To compute the space efficiency, we need to make an assumption about the mean file size. A recent measurement at the author's department, which has 1000 users and over 1 million UNIX disk files, gives a median size of 1680 bytes, meaning that half the files are smaller than 1680 bytes and half are larger. As an aside, the median is a better metric than the mean because a very small number of files can influence the mean enormously, but not the median. (The mean is in fact 10,845 bytes, due in part to a few 100 MB hardware manuals that happen to be online). For simplicity, let us assume all files are 2 KB, which leads to the dashed curve in Fig. 6-20 for the disk space efficiency.

The two curves can be understood as follows. The access time for a block is completely dominated by the seek time and rotational delay, so given that it is going to cost 14 msec to access a block, the more data that are fetched, the better. Hence the data rate goes up with block size (until the transfers take so long that

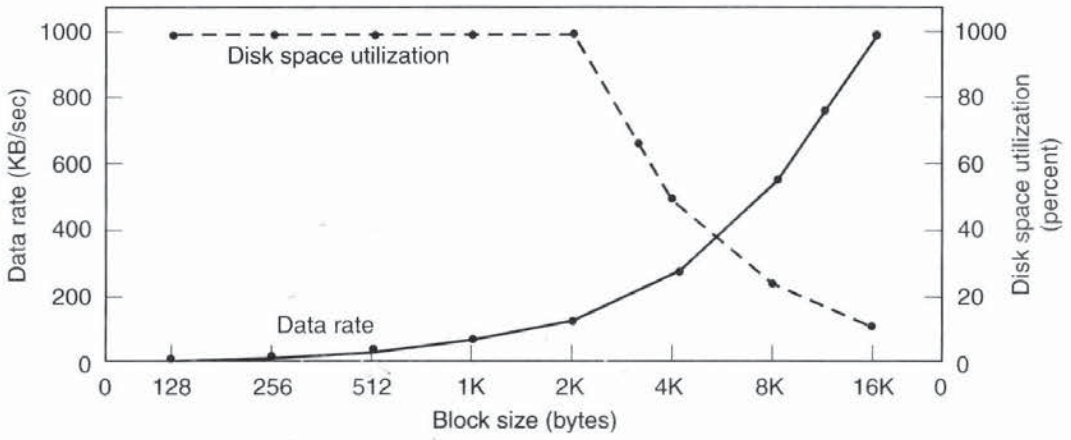


Figure 6-20. The solid curve (left-hand scale) gives the data rate of a disk. The dashed curve (right-hand scale) gives the disk space efficiency. All files are 2 KB.

the transfer time begins to dominate). With small blocks that are powers of two and 2-KB files, no space is wasted in a block. However, with 2-KB files and 4 KB or larger blocks, some disk space is wasted. In reality, few files are a multiple of the disk block size, so some space is always wasted in the last block of a file.

What the curves show, however, is that performance and space utilization are inherently in conflict. Small blocks are bad for performance but good for disk space utilization. A compromise size is needed. For this data, 4 KB might be a good choice, but some operating systems made their choices a long time ago, when the disk parameters and file sizes were different. For UNIX, 1 KB is commonly used. For MS-DOS the block size can be any power of two from 512 bytes to 32 KB, but is determined by the disk size and for reasons unrelated to these arguments (the maximum number of blocks on a disk partition is 2^{16}, which forces large blocks on large disks).

In an experiment to see if Windows NT file usage was appreciably different from UNIX file usage, Vogels made measurements on files at Cornell University (Vogels, 1999). He observed that NT file usage is more complicated than on UNIX. He wrote:

> *When we type a few characters in the* notepad *text editor, saving this to a file will trigger 26 system calls, including 3 failed open attempts, 1 file overwrite and 4 additional open and close sequences.*

Nevertheless, he observed a median size (weighted by usage) of files just read at 1 KB, files just written as 2.3 KB and files read and written as 4.2 KB. Given the fact that Cornell has much more large-scale scientific computing than the author's institution and the difference in measurement technique (static versus dynamic), the results are reasonably consistent with a median file size of around 2 KB.

Keeping Track of Free Blocks

Once a block size has been chosen, the next issue is how to keep track of free blocks. Two methods are widely used, as shown in Fig. 6-21. The first one consists of using a linked list of disk blocks, with each block holding as many free disk block numbers as will fit. With a 1-KB block and a 32-bit disk block number, each block on the free list holds the numbers of 255 free blocks. (One slot is needed for the pointer to the next block). A 16-GB disk needs a free list of maximum 16,794 blocks to hold all 2^{24} disk block numbers. Often free blocks are used to hold the free list.

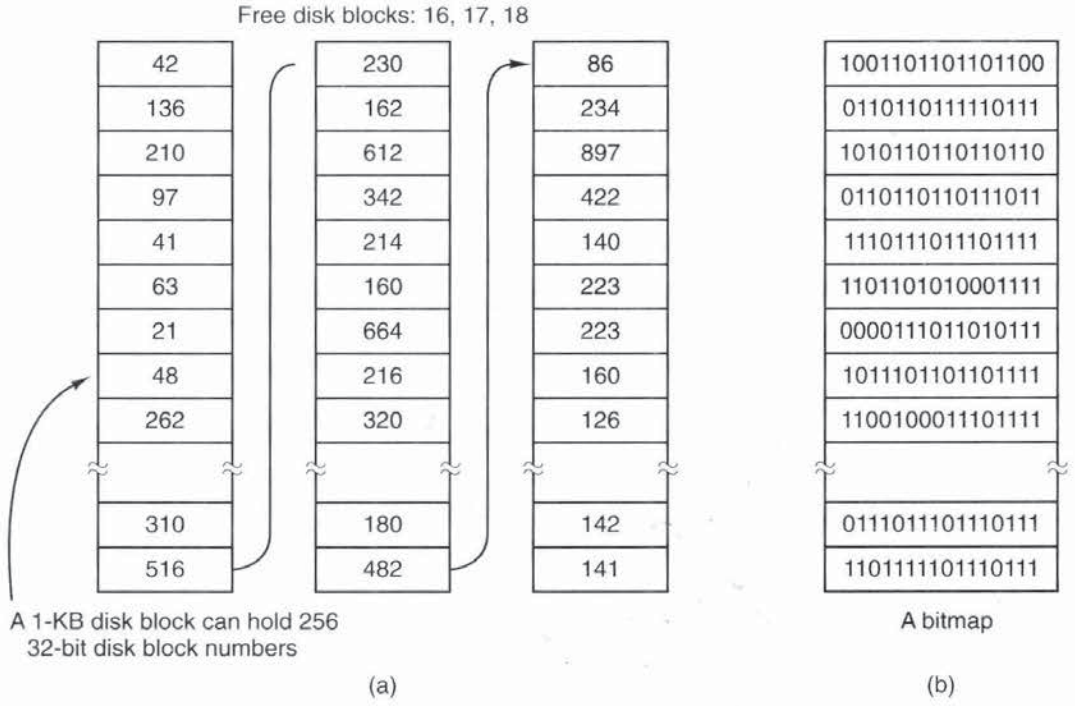


Figure 6-21. (a) Storing the free list on a linked list. (b) A bitmap.

The other free space management technique is the bitmap. A disk with n blocks requires a bitmap with n bits. Free blocks are represented by 1s in the map, allocated blocks by 0s (or vice versa). A 16-GB disk has 2^{24} 1-KB blocks and thus requires 2^{24} bits for the map, which requires 2048 blocks. It is not surprising that the bitmap requires less space, since it uses 1 bit per block, versus 32 bits in the linked list model. Only if the disk is nearly full (i.e., has few free blocks) will the linked list scheme require fewer blocks than the bitmap. On the other hand, if there are many blocks free, some of them can be borrowed to hold the free list without any loss of disk capacity.

When the free list method is used, only one block of pointers need be kept in main memory. When a file is created, the needed blocks are taken from the block

of pointers. When it runs out, a new block of pointers is read in from the disk. Similarly, when a file is deleted, its blocks are freed and added to the block of pointers in main memory. When this block fills up, it is written to disk.

Under certain circumstances, this method leads to unnecessary disk I/O. Consider the situation of Fig. 6-22(a), in which the block of pointers in memory has room for only two more entries. If a three-block file is freed, the pointer block overflows and has to be written to disk, leading to the situation of Fig. 6-22(b). If a three-block file is now written, the full block of pointers has to be read in again, taking us back to Fig. 6-22(a). If the three-block file just written was a temporary file, when it is freed, another disk write is needed to write the full block of pointers back to the disk. In short, when the block of pointers is almost empty, a series of short-lived temporary files can cause a lot of disk I/O.

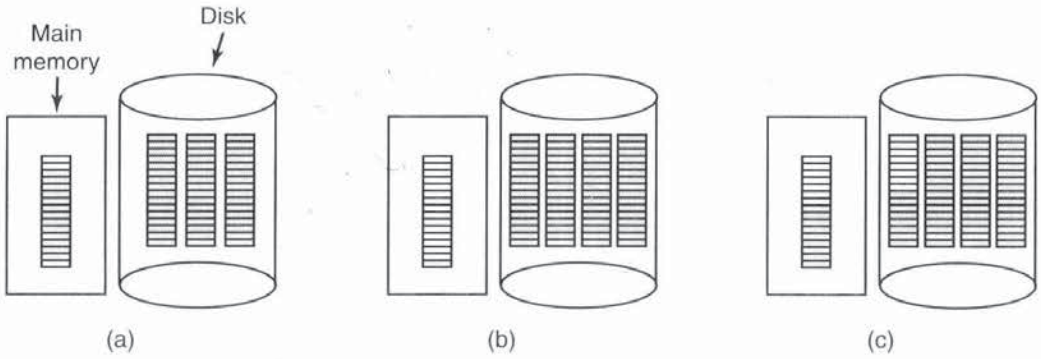


Figure 6-22. (a) An almost-full block of pointers to free disk blocks in memory and three blocks of pointers on disk. (b) Result of freeing a three-block file. (c) An alternative strategy for handling the three free blocks. The shaded entries represent pointers to free disk blocks.

An alternative approach that avoids most of this disk I/O is to split the full block of pointers. Thus instead of going from Fig. 6-22(a) to Fig. 6-22(b), we go from Fig. 6-22(a) to Fig. 6-22(c) when three blocks are freed. Now the system can handle a series of temporary files without doing any disk I/O. If the block in memory fills up, it is written to the disk and the half-full block from disk is read in. The idea here is to keep most of the pointer blocks on disk full (to minimize disk usage), but keep the one in memory about half full, so it can handle both file creation and file removal without disk I/O on the free list.

With a bitmap, it is also possible to keep just one block in memory, going to disk for another one only when it becomes full or empty. An additional benefit of this approach is that by doing all the allocation from a single block of the bitmap, the disk blocks will be close together, thus minimizing disk arm motion. Since the bitmap is a fixed-size data structure, if the kernel is (partially) paged, the bitmap can be put in virtual memory and have pages of it paged in as needed.

Disk Quotas

To prevent people from hogging too much disk space, multiuser operating systems often provide a mechanism for enforcing disk quotas. The idea is that the system administrator assigns each user a maximum allotment of files and blocks, and the operating system makes sure that the users do not exceed their quotas. A typical mechanism is described below.

When a user opens a file, the attributes and disk addresses are located and put into an open file table in main memory. Among the attributes is an entry telling who the owner is. Any increases in the file's size will be charged to the owner's quota.

A second table contains the quota record for every user with a currently open file, even if the file was opened by someone else. This table is shown in Fig. 6-23. It is an extract from a quota file on disk for the users whose files are currently open. When all the files are closed, the record is written back to the quota file.

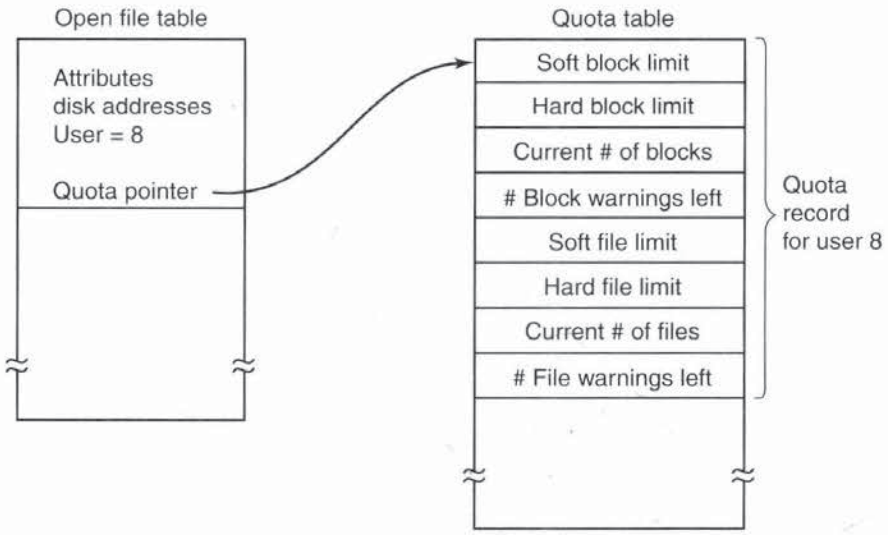


Figure 6-23. Quotas are kept track of on a per-user basis in a quota table.

When a new entry is made in the open file table, a pointer to the owner's quota record is entered into it, to make it easy to find the various limits. Every time a block is added to a file, the total number of blocks charged to the owner is incremented, and a check is made against both the hard and soft limits. The soft limit may be exceeded, but the hard limit may not. An attempt to append to a file when the hard block limit has been reached will result in an error. Analogous checks also exist for the number of files.

When a user attempts to log in, the system checks the quota file to see if the user has exceeded the soft limit for either number of files or number of disk blocks. If either limit has been violated, a warning is displayed, and the count of

warnings remaining is reduced by one. If the count ever gets to zero, the user has ignored the warning one time too many, and is not permitted to log in. Getting permission to log in again will require some discussion with the system administrator.

This method has the property that users may go above their soft limits during a login session, provided they remove the excess before logging out. The hard limits may never be exceeded.

6.3.6 File System Reliability

Destruction of a file system is often a far greater disaster than destruction of a computer. If a computer is destroyed by fire, lightning surges, or a cup of coffee poured onto the keyboard, it is annoying and will cost money, but generally a replacement can be purchased with a minimum of fuss. Inexpensive personal computers can even be replaced within an hour by just going to the dealer (except at universities, where issuing a purchase order takes three committees, five signatures, and 90 days).

If a computer's file system is irrevocably lost, whether due to hardware, software, or rats gnawing on the floppy disks, restoring all the information will be difficult, time consuming, and in many cases, impossible. For the people whose programs, documents, customer files, tax records, databases, marketing plans, or other data are gone forever, the consequences can be catastrophic. While the file system cannot offer any protection against physical destruction of the equipment and media, it can help protect the information. In this section we will look at some of the issues involved in safeguarding the file system.

Floppy disks are generally perfect when they leave the factory, but they can develop bad blocks during use. Hard disks frequently have bad blocks right from the start: it is just too expensive to manufacture them completely free of all defects. As we saw in Chap. 5, bad blocks are generally handled by the controller by replacing bad sectors with spares provided for that purpose. Despite this technique, there are other reliability issues which we will consider below.

Backups

Most people do not think making backups of their files is worth the time and effort—until one fine day their disk abruptly dies, at which time most of them undergo a deathbed conversion. Companies, however, (usually) well understand the value of their data and generally do a backup at least once a day, usually to tape. Modern tapes hold tens or sometimes even hundreds of gigabytes and cost pennies per gigabyte. Nevertheless, Making backups is not quite as trivial as it sounds, so we will examine some of the related issues below.

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

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

The first one covers getting the computer running again after a disk crash, fire, flood, or other natural catastrophe. In practice, these things do not happen very often, which is why many people do not bother with backups. These people also tend not to have fire insurance on their houses for the same reason.

The second reason is that users often accidentally remove files that they later need again. This problem occurs so often that when a file is "removed" in Windows, it is not deleted at all, but just moved to a special directory, the **recycle bin**, so it can be fished out and restored easily later. Backups take this principle further and allow files that were removed days, even weeks ago, to be restored from old backup tapes.

Making a backup takes a long time and occupies a large amount of space, so doing it efficiently and conveniently is important. These considerations raise the following issues. First, should the entire file system be backed up or only part of it? At many installations, the executable (binary) programs are kept in a limited part of the file system tree. It is not necessary to back up these files if they can all be reinstalled from the manufacturers' CD-ROMs. Also, most systems have a directory for temporary files. There is usually no reason to back it up either. In UNIX, all the special files (I/O devices) are kept in a directory */dev*. Not only is backing up this directory not necessary, it is downright dangerous because the backup program would hang forever if it tried to read each of these to completion. In short, it is usually desirable to back up only specific directories and everything in them rather than the entire file system.

Second, it is wasteful to back up files that have not changed since the last backup, which leads to the idea of **incremental dumps**. The simplest form of incremental dumping is to make a complete dump (backup) periodically, say weekly or monthly, and to make a daily dump of only those files that have been modified since the last full dump. Even better is to dump only those files that have changed since they were last dumped. While this scheme minimizes dumping time, it makes recovery more complicated because first the most recent full dump has to be restored, followed by all the incremental dumps in reverse order. To ease recovery, more sophisticated incremental dumping schemes are often used.

Third, since immense amounts of data are typically dumped, it may be desirable to compress the data before writing them to tape. However, with many compression algorithms, a single bad spot on the backup tape can foil the decompression algorithm and make an entire file or even an entire tape unreadable. Thus the decision to compress the backup stream must be carefully considered.

Fourth, it is difficult to perform a backup on an active file system. If files and directories are being added, deleted, and modified during the dumping process, the resulting dump may be inconsistent. However, since making a dump may take hours, it may be necessary to take the system offline for much of the night to make the backup, something that is not always acceptable. For this reason, algorithms have been devised for making rapid snapshots of the file system state by copying critical data structures, and then requiring future changes to files and directories to copy the blocks instead of updating them in place (Hutchinson et al., 1999). In this way, the file system is effectively frozen at the moment of the snapshot, so it can be backed up at leisure afterward.

Fifth and last, making backups introduces many nontechnical problems into an organization. The best online security system in the world may be useless if the system administrator keeps all the backup tapes in his office and leaves it open and unguarded whenever he walks down the hall to get output from the printer. All a spy has to do is pop in for a second, put one tiny tape in his pocket, and saunter off jauntily. Goodbye security. Also, making a daily backup has little use if the fire that burns down the computers also burns up all the backup tapes. For this reason, backup tapes should be kept off-site, but that introduces more security risks. For a thorough discussion of these and other practical administration issues, see (Nemeth et al., 2000). Below we will discuss only the technical issues involved in making file system backups.

Two strategies can be used for dumping a disk to tape: a physical dump or a logical dump. A **physical dump** starts at block 0 of the disk, writes all the disk blocks onto the output tape in order, and stops when it has copied the last one. Such a program is so simple that it can probably be made 100% bug free, something that can probably not be said about any other useful program.

Nevertheless, it is worth making several comments about physical dumping. For one thing, there is no value in backing up unused disk blocks. If the dumping program can get access to the free block data structure, it can avoid dumping unused blocks. However, skipping unused blocks requires writing the number of each block in front of the block (or the equivalent), since it is no longer true that block k on the tape was block k on the disk.

A second concern is dumping bad blocks. If all bad blocks are remapped by the disk controller and hidden from the operating system as we described in Sec. 5.4.4, physical dumping works fine. On the other hand, if they are visible to the operating system and maintained in one or more "bad block files" or bitmaps, it is absolutely essential that the physical dumping program get access to this information and avoid dumping them to prevent endless disk read errors during the dumping process.

The main advantages of physical dumping are simplicity and great speed (basically, it can run at the speed of the disk). The main disadvantages are the inability to skip selected directories, make incremental dumps, and restore individual files upon request. For these reasons, most installations make logical dumps.

A **logical dump** starts at one or more specified directories and recursively dumps all files and directories found there that have changed since some given base date (e.g., the last backup for an incremental dump or system installation for a full dump). Thus in a logical dump, the dump tape gets a series of carefully identified directories and files, which makes it easy to restore a specific file or directory upon request.

Since logical dumping is the most common form, let us examine a common algorithm in detail using the example of Fig. 6-24 to guide us. Most UNIX systems use this algorithm. In the figure we see a file tree with directories (squares) and files (circles). The shaded items have been modified since the base date and thus need to be dumped. The unshaded ones do not need to be dumped.

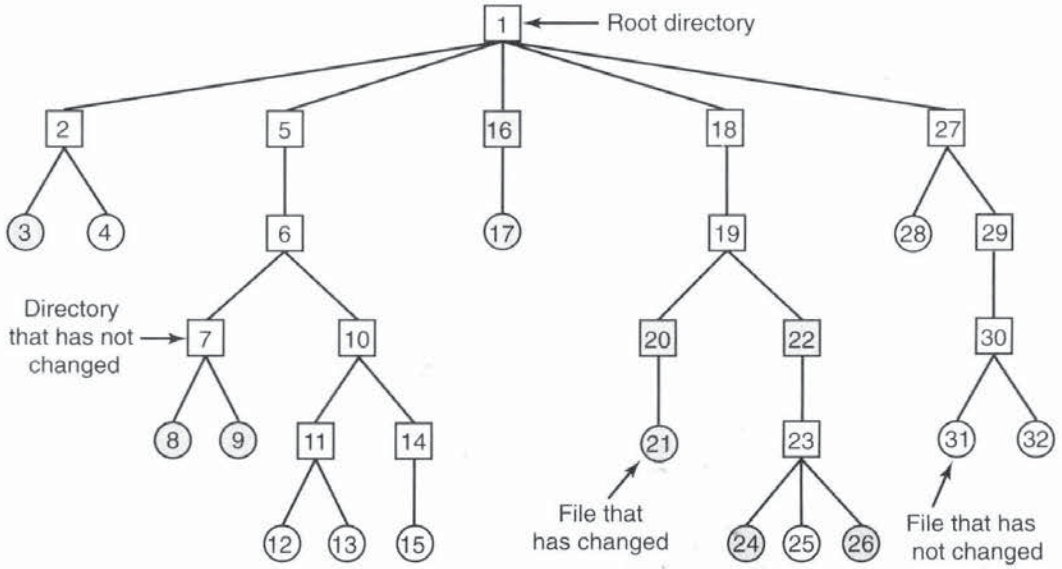


Figure 6-24. A file system to be dumped. The squares are directories and the circles are files. The shaded items have been modified since the last dump. Each directory and file is labeled by its i-node number.

This algorithm also dumps all directories (even unmodified ones) that lie on the path to a modified file or directory for two reasons. First, to make it possible to restore the dumped files and directories to a fresh file system on a different computer. In this way, the dump and restore programs can be used to transport entire file systems between computers.

The second reason for dumping unmodified directories above modified files is to make it possible to incrementally restore a single file (possibly to handle recovery from stupidity). Suppose that a full file system dump is done Sunday evening and an incremental dump is done on Monday evening. On Tuesday the directory */usr/jhs/proj/nr3* is removed, along with all the directories and files under it. On Wednesday morning bright and early the user wants to restore the file */usr/jhs/proj/nr3/plans/summary* However, is not possible to just restore the

file *summary* because there is no place to put it. The directories *nr3* and *plans* must be restored first. To get their owners, modes, times, etc., correct, these directories must be present on the dump tape 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. 6-25(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. 6-25(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. In 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.



Figure 6-25. Bit maps used by the logical dumping algorithm.

At this point it is known which directories and files must be dumped. These are the ones marked in Fig. 6-25(b). Phase 3 consists of scanning the i-nodes in numerical order and dumping all the directories that are marked for dumping. These are shown in Fig. 6-25(c). Each directory is prefixed by the directory's attributes (owner, times, etc.) so they can be restored. Finally, in phase 4, the files marked in Fig. 6-25(d) are also dumped, again prefixed by their attributes. This completes the dump.

Restoring a file system from the dump tapes 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 tape, 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 not be restored. Core files often have a large hole 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 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).

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 the problem of inconsistent file systems, most computers have a utility program that checks file system consistency. For example, UNIX has *fsck* and Windows has *scandisk*. This utility can be run whenever the system is booted, especially after a crash. The description below tells how *fsck* works. *Scandisk* 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. 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. 6-26(a). However, as a result of a crash, the tables might look like Fig. 6-26(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 do 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.

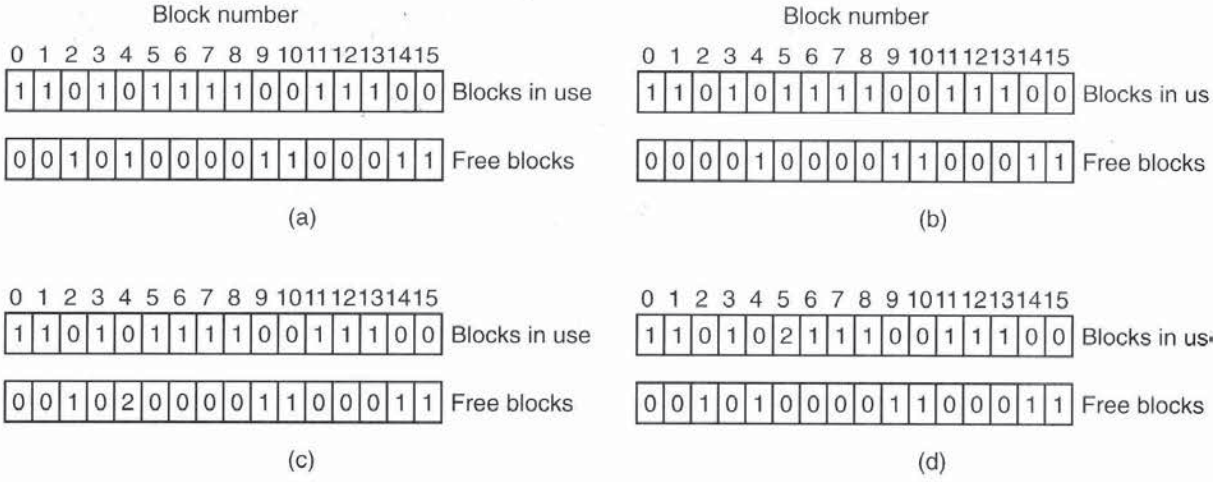


Figure 6-26. File system states. (a) Consistent. (b) Missing block. (c) Duplicate block in free list. (d) Duplicate data block.

Another situation that might occur is that of Fig. 6-26(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. 6-26(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.

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 file 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 it 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 MS-DOS and some other systems, when a file is removed, all that happens is that a bit is set in the directory or i-node marking the file as removed. No disk blocks are returned to the free list until they are actually needed. Thus, if the user discovers the error immediately, it is possible to run a special utility program that "unremoves" (i.e., restores) the removed files. In Windows, files that are removed are placed in the recycle bin, from which they can later be retrieved if need be. Of course, no storage is reclaimed until they are actually deleted from this directory.

6.3.7 File System Performance

Access to disk is much slower than access to memory. Reading a memory word might take 10 nsec. Reading from a hard disk might proceed at 10 MB/sec, which is forty 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 *cacher*, 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. 6-27. 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 the collision chain can be followed.

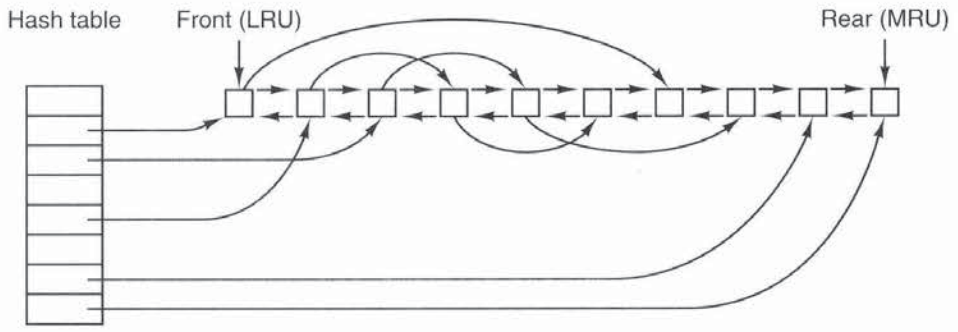


Figure 6-27. The buffer cache data structures.

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. 6-27, 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 of this list. 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 the crashes and file system consistency discussed in the previous section. If a critical block, such as an i-node block, is read into the cache and modified, but not rewritten to the disk, a crash will leave the file system in an inconsistent state. If the i-node block is put at the end of the LRU chain, it may be quite a while before it reaches the front and is rewritten to the disk.

Furthermore, some blocks, such as i-node blocks, are rarely referenced two times within a short interval. These considerations lead to a modified LRU scheme, taking two factors into account:

1. Is the block likely to be needed again soon?
2. Is the block essential to the consistency of the file system?

For both questions, blocks can be divided into categories such as i-node blocks, indirect blocks, directory blocks, full data blocks, and partially full data blocks.

Blocks that will probably not be needed again soon go on the front, rather than the rear of the LRU list, so their buffers will be reused quickly. Blocks that might be needed again soon, such as a partly full block that is being written, go on the end of the list, so they will stay around for a long time.

The second question is independent of the first one. If the block is essential to the file system consistency (basically, everything except data blocks), and it has been modified, it should be written to disk immediately, regardless of which end of the LRU list it is put on. By writing critical blocks quickly, we greatly reduce the probability that a crash will wreck the file system. While a user may be unhappy if one of his files is ruined in a crash, he is likely to be far more unhappy if the whole file system is lost.

Even with this measure to keep the file system integrity intact, it is undesirable to keep data blocks in the cache too long before writing them out. Consider the plight of someone who is using a personal computer to write a book. Even if our writer periodically tells the editor to write the file being edited to the disk, there is a good chance that everything will still be in the cache and nothing on the disk. If the system crashes, the file system structure will not be corrupted, but a whole day's work will be lost.

This situation need not happen very often before we have a fairly unhappy user. Systems take two approaches to dealing with it. The UNIX way is to have a system call, sync, which forces all the modified blocks out onto the disk immediately. When the system is started up, a program, usually called *update*, is started up in the background to sit in an endless loop issuing sync calls, sleeping for 30 sec between calls. As a result, no more than 30 seconds of work is lost due to a crash.

The MS-DOS way is to write every modified block to disk as soon as it has been written. Caches in which all modified blocks are written back to the disk immediately are called **write-through caches**. They require more disk I/O than nonwrite-through caches. The difference between these two approaches can be seen when a program writes a 1-KB block full, one character at a time. UNIX will collect all the characters in the cache and write the block out once every 30 seconds, or whenever the block is removed from the cache. MS-DOS will make a disk access for every character written. Of course, most programs do internal buffering, so they normally write not a character, but a line or a larger unit on each write system call.

A consequence of this difference in caching strategy is that just removing a (floppy) disk from a UNIX system without doing a sync will almost always result in lost data, and frequently in a corrupted file system as well. With MS-DOS, no problem arises. These differing strategies were chosen because UNIX was developed in an environment in which all disks were hard disks and not removable, whereas MS-DOS started out in the floppy disk world. As hard disks became the norm, the UNIX approach, with its better efficiency, became the norm, and is also used now on Windows for hard disks.

Block Read Ahead

A second technique for improving perceived file system performance is to try to get blocks into the cache before they are needed to increase the hit rate. In particular, many files are read sequentially. When the file system is asked to produce block k in a file, it does that, but when it is finished, it makes a sneaky check in the cache to see if block $k + 1$ is already there. If it is not, it schedules a read for block $k + 1$ in the hope that when it is needed, it will have already arrived in the cache. At the very least, it will be on the way.

Of course, this read ahead strategy only works for files that are being read sequentially. If a file is being randomly accessed, read ahead does not help. In fact, it hurts by tying up disk bandwidth reading in useless blocks and removing potentially useful blocks from the cache (and possibly tying up more disk bandwidth writing them back to disk if they are dirty). To see whether read ahead is worth doing, the file system can keep track of the access patterns to each open file. For example, a bit associated with each file can keep track of whether the file is in "sequential access mode" or "random access mode." Initially, the file is given the benefit of the doubt and put in sequential access mode. However, whenever a seek is done, the bit is cleared. If sequential reads start happening again, the bit is set once again. In this way, the file system can make a reasonable guess about whether it should read ahead or not. If it gets it wrong once it a while, it is not a disaster, just a little bit of wasted disk bandwidth.

Reducing Disk Arm Motion

Caching and read ahead are not the only ways to increase file system performance. Another important technique is to reduce the amount of disk arm motion by putting blocks that are likely to be accessed in sequence close to each other, preferably in the same cylinder. When an output file is written, the file system has to allocate the blocks one at a time, as they are needed. If the free blocks are recorded in a bitmap, and the whole bitmap is in main memory, it is easy enough to choose a free block as close as possible to the previous block. With a free list, part of which is on disk, it is much harder to allocate blocks close together.

However, even with a free list, some block clustering can be done. The trick is to keep track of disk storage not in blocks, but in groups of consecutive blocks. If sectors consist of 512 bytes, the system could use 1-KB blocks (2 sectors) but allocate disk storage in units of 2 blocks (4 sectors). This is not the same as having a 2-KB disk blocks, since the cache would still use 1-KB blocks and disk transfers would still be 1 KB but reading a file sequentially on an otherwise idle system would reduce the number of seeks by a factor of two, considerably improving performance. A variation on the same theme is to take account of rotational positioning. When allocating blocks, the system attempts to place consecutive blocks in a file in the same cylinder.

Another performance bottleneck in systems that use i-nodes or anything equivalent to i-nodes is that reading even a short file requires two disk accesses: one for the i-node and one for the block. The usual i-node placement is shown in Fig. 6-28(a). Here all the i-nodes are near the beginning of the disk, so the average distance between an i-node and its blocks will be about half the number of cylinders, requiring long seeks.

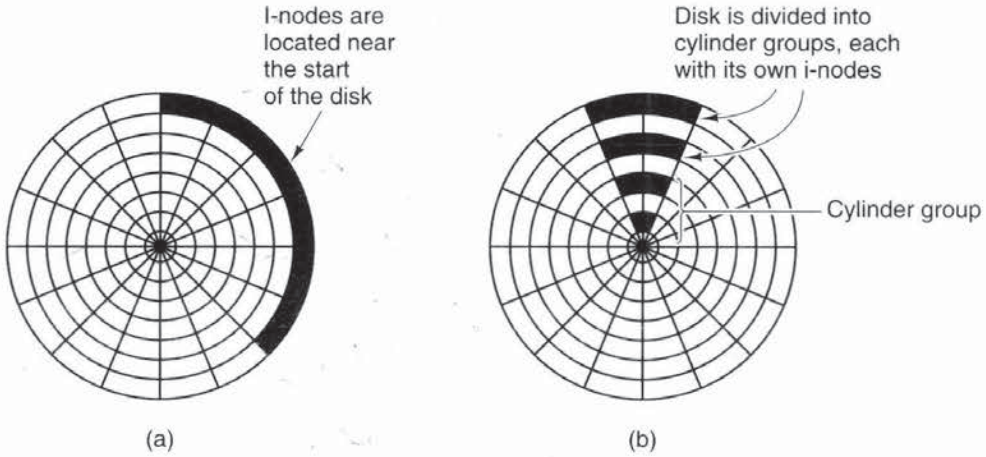


Figure 6-28. (a) I-nodes placed at the start of the disk. (b) Disk divided into cylinder groups, each with its own blocks and i-nodes.

One easy performance improvement is to put the i-nodes in the middle of the disk, rather than at the start, thus reducing the average seek between the i-node and the first block by a factor of two. Another idea, shown in Fig. 6-28(b), is to divide the disk into cylinder groups, each with its own i-nodes, blocks, and free list (McKusick et al., 1984). When creating a new file, any i-node can be chosen, but an attempt is made to find a block in the same cylinder group as the i-node. If none is available, then a block in a nearby cylinder group is used.

6.3.8 Log-Structured File Systems

Changes in technology are putting pressure on current file systems. In particular, CPUs keep getting faster, disks are becoming much bigger and cheaper (but not much faster), and memories are growing exponentially in size. The one parameter that is not improving by leaps and bounds is disk seek time. The combination of these factors means that a performance bottleneck is arising in many file systems. Research done at Berkeley attempted to alleviate this problem by designing a completely new kind of file system, LFS (the **Log-structured File System**). In this section we will briefly describe how LFS works. For a more complete treatment, see (Rosenblum and Ousterhout, 1991).

The idea that drove the LFS design is that as CPUs get faster and RAM memories get larger, disk caches are also increasing rapidly. Consequently, it is now

possible to satisfy a very substantial fraction of all read requests directly from the file system cache, with no disk access needed. It follows from this observation, that in the future, most disk accesses will be writes, so the read-ahead mechanism used in some file systems to fetch blocks before they are needed no longer gains much performance.

To make matters worse, in most file systems, writes are done in very small chunks. Small writes are highly inefficient, since a 50-μsec disk write is often preceded by a 10-msec seek and a 4-msec rotational delay. With these parameters, disk efficiency drops to a fraction of 1 percent.

To see where all the small writes come from, consider creating a new file on a UNIX system. To write this file, the i-node for the directory, the directory block, the i-node for the file, and the file itself must all be written. While these writes can be delayed, doing so exposes the file system to serious consistency problems if a crash occurs before the writes are done. For this reason, the i-node writes are generally done immediately.

From this reasoning, the LFS designers decided to re-implement the UNIX file system in such a way as to achieve the full bandwidth of the disk, even in the face of a workload consisting in large part of small random writes. The basic idea is to structure the entire disk as a log. Periodically, and when there is a special need for it, all the pending writes being buffered in memory are collected into a single segment and written to the disk as a single contiguous segment at the end of the log. A single segment may thus contain i-nodes, directory blocks, and data blocks, all mixed together. At the start of each segment is a segment summary, telling what can be found in the segment. If the average segment can be made to be about 1 MB, almost the full bandwidth of the disk can be utilized.

In this design, i-nodes still exist and have the same structure as in UNIX, but they are now scattered all over the log, instead of being at a fixed position on the disk. Nevertheless, when an i-node is located, locating the blocks is done in the usual way. Of course, finding an i-node is now much harder, since its address cannot simply be calculated from its i-number, as in UNIX. To make it possible to find i-nodes, an i-node map, indexed by i-number, is maintained. Entry *i* in this map points to i-node *i* on the disk. The map is kept on disk, but it is also cached, so the most heavily used parts will be in memory most of the time.

To summarize what we have said so far, all writes are initially buffered in memory, and periodically all the buffered writes are written to the disk in a single segment, at the end of the log. Opening a file now consists of using the map to locate the i-node for the file. Once the i-node has been located, the addresses of the blocks can be found from it. All of the blocks will themselves be in segments, somewhere in the log.

If disks were infinitely large, the above description would be the entire story. However, real disks are finite, so eventually the log will occupy the entire disk, at which time no new segments can be written to the log. Fortunately, many existing segments may have blocks that are no longer needed, for example, if a file is

overwritten, its i-node will now point to the new blocks, but the old ones will still be occupying space in previously written segments.

To deal with this problem, LFS has a **cleaner** thread that spends its time scanning the log circularly to compact it. It starts out by reading the summary of the first segment in the log to see which i-nodes and files are there. It then checks the current i-node map to see if the i-nodes are still current and file blocks are still in use. If not, that information is discarded. The i-nodes and blocks that are still in use go into memory to be written out in the next segment. The original segment is then marked as free, so the log can use it for new data. In this manner, the cleaner moves along the log, removing old segments from the back and putting any live data into memory for rewriting in the next segment. Consequently, the disk is a big circular buffer, with the writer thread adding new segments to the front and the cleaner thread removing old ones from the back.

The bookkeeping here is nontrivial, since when a file block is written back to a new segment, the i-node of the file (somewhere in the log) must be located, updated, and put into memory to be written out in the next segment. The i-node map must then be updated to point to the new copy. Nevertheless, it is possible to do the administration, and the performance results show that all this complexity is worthwhile. Measurements given in the papers cited above show that LFS outperforms UNIX by an order of magnitude on small writes, while having a performance that is as good as or better than UNIX for reads and large writes.

6.4 EXAMPLE FILE SYSTEMS

In the following sections we will discuss several example file systems, ranging from quite simple to highly sophisticated. Since modern UNIX file systems and Windows 2000's native file system are covered in the chapter on UNIX (Chap. 10) and the chapter on Windows 2000 (Chap. 11) we will not cover those systems here. We will, however, examine their predecessors below.

6.4.1 CD-ROM File Systems

As our first example of a file system, let us consider the file systems used on CD-ROMs. These systems are particularly simple because they were designed for write-once media. Among other things, for example, they have no provision for keeping track of free blocks because on a CD-ROM files cannot be freed or added after the disk has been manufactured. Below we will take a look at the main CD-ROM file system type and two extensions to it.

The ISO 9660 File System

The most common standard for CD-ROM file systems was adopted as an International Standard in 1988 under the name **ISO 9660**. Virtually every CD-ROM currently on the market is compatible with this standard, sometimes with

the extensions to be discussed below. One of the goals of this standard was to make every CD-ROM readable on every computer, independent of the byte ordering used and independent of the operating system used. As a consequence, some limitations were placed on the file system to make it possible for the weakest operating systems then in use (such as MS-DOS) to read it.

CD-ROMs do not have concentric cylinders the way magnetic disks do. Instead there is a single continuous spiral containing the bits in a linear sequence (although seeks across the spiral are possible). The bits along the spiral are divided into logical blocks (also called logical sectors) of 2352 bytes. Some of these are for preambles, error correction, and other overhead. The payload portion of each logical block is 2048 bytes. When used for music, CDs have leadins, leadouts, and intertrack gaps, but these are not used for data CD-ROMs. Often the position of a block along the spiral is quoted in minutes and seconds. It can be converted to a linear block number using the conversion factor of 1 sec = 75 blocks.

ISO 9660 supports CD-ROM sets with as many as $2^{16} - 1$ CDs in the set. The individual CD-ROMs may also be partitioned into logical volumes (partitions). However, below we will concentrate on ISO 9660 for a single unpartitioned CD-ROM.

Every CD-ROM begins with 16 blocks whose function is not defined by the ISO 9660 standard. A CD-ROM manufacturer could use this area for providing a bootstrap program to allow the computer to be booted from the CD-ROM, or for some other purpose. Next comes one block containing the **primary volume descriptor**, which contains some general information about the CD-ROM. Among this information are the system identifier (32 bytes), volume identifier (32 bytes), publisher identifier (128 bytes), and data preparer identifier (128 bytes). The manufacturer can fill in these fields in any desired way, except that only upper case letters, digits, and a very small number of punctuation marks may be used to ensure cross-platform compatibility.

The primary volume descriptor also contains the names of three files, which may contain the abstract, copyright notice, and bibliographic information, respectively. In addition, certain key numbers are also present, including the logical block size (normally 2048, but 4096, 8192, and larger powers of two are allowed in certain cases), the number of blocks on the CD-ROM, and the creation and expiration dates of the CD-ROM. Finally, the primary volume descriptor also contains a directory entry for the root directory, telling where to find it on the CD-ROM (i.e., which block it starts at). From this directory, the rest of the file system can be located.

In addition to the primary volume descriptor, a CD-ROM may contain a supplementary volume descriptor. It contains similar information to the primary, but that will not concern us here.

The root directory, and all other directories for that matter, consists of a variable number of entries, the last of which contains a bit marking it as the final one.

The directory entries themselves are also variable length. Each directory entry consists of 10 to 12 fields, some of which are in ASCII and others of which are numerical fields in binary. The binary fields are encoded twice, once in little-endian format (used on Pentiums, for example) and once in big-endian format (used on SPARCs, for example). Thus a 16-bit number uses 4 bytes and a 32-bit number uses 8 bytes. The use of this redundant coding was necessary to avoid hurting anyone's feelings when the standard was developed. If the standard had dictated little endian, then people from companies with big-endian products would have felt like second-class citizens and would not have accepted the standard. The emotional content of a CD-ROM can thus be quantified and measured exactly in kilobytes/hour of wasted space.

The format of an ISO 9660 directory entry is illustrated in Fig. 6-29. Since directory entries have variable lengths, the first field is a byte telling how long the entry is. This byte is defined to have the high-order bit on the left to avoid any ambiguity.

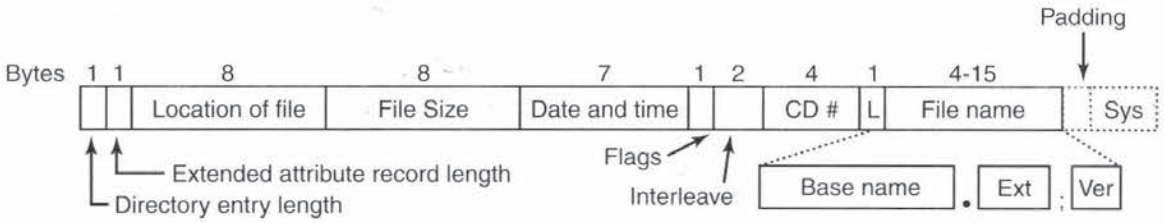


Figure 6-29. The ISO 9660 directory enty.

Directory entries may optionally have an extended attributes. If this feature is used for a directory entry, the second byte tells how long the extended attributes are.

Next comes the starting block of the file itself. Files are stored as contiguous runs of blocks, so a file's location is completely specified by the starting block and the size, which is contained in the next field.

The date and time that the CD-ROM was recorded is stored in the next field, with separate bytes for the year, month, day, hour, minute, second, and time zone. Years begin to count at 1900, which means that CD-ROMs will suffer from a Y2156 problem because the year following 2155 will be 1900. This problem could have been delayed by defining the origin of time to be 1988 (the year the standard was adopted). Had that been done, the problem would have been postponed until 2244. Every 88 extra years helps.

The *Flags* field contains a few miscellaneous bits, including one to hide the entry in listings (a feature copied from MS-DOS), one to distinguish an entry that is a file from an entry that is a directory, one to enable the use of the extended attributes, and one to mark the last entry in a directory. A few other bits are also

present in this field but they will not concern us here. The next field deals with interleaving pieces of files in a way that is not used in the simplest version of ISO 9660, so we will not consider it further.

The next field tells which CD-ROM the file is located on. It is permitted that a directory entry on one CD-ROM refers to a file located on another CD-ROM in the set. In this way it is possible to build a master directory on the first CD-ROM that lists all the files on all the CD-ROMs in the complete set.

The field marked *L* in Fig. 6-29 gives the size of the file name in bytes. It is followed by the file name itself. A file name consists of a base name, a dot, an extension, a semicolon, and a binary version number (1 or 2 bytes). The base name and extension may use upper case letters, the digits 0–9, and the underscore character. All other characters are forbidden to make sure that every computer can handle every file name. The base name can be up to eight characters; the extension can be up to three characters. These choices were dictated by the need to be MS-DOS compatible. A file name may be present in a directory multiple times, as long as each one has a different version number.

The last two fields are not always present. The *Padding* field is used to force every directory entry to be an even number of bytes, to align the numeric fields of subsequent entries on 2-byte boundaries. If padding is needed, a 0 byte is used. Finally, we have the *System use* field. Its function and size are undefined, except that it must be an even number of bytes. Different systems use it in different ways. The Macintosh keeps Finder flags here, for example.

Entries within a directory are listed in alphabetical order except for the first two entries. The first entry is for the directory itself. The second one is for its parent. In this respect, these entries are similar to the UNIX . and .. directory entries. The files themselves need not be in directory order.

There is no explicit limit to the number of entries in a directory. However, there is a limit to the depth of nesting. The maximum depth of directory nesting is eight.

ISO 9660 defines what are called three levels. Level 1 is the most restrictive and specifies that file names are limited to 8 + 3 characters as we have described, and also requires all files to be contiguous as we have described. Furthermore, it specifies that directory names be limited to eight characters with no extensions. Use of this level maximizes the chances that a CD-ROM can be read on every computer.

Level 2 relaxes the length restriction. It allows files and directories to have names of up to 31 characters, but still from the same set of characters.

Level 3 uses the same name limits as level 2, but partially relaxes the assumption that files have to be contiguous. With this level, a file may consist of several sections, each of which is a contiguous run of blocks. The same run may occur multiple times in a file and may also occur in two or more files. If large chunks of data are repeated in several files, level 3 provides some space optimization by not requiring the data to be present multiple times.

Rock Ridge Extensions

As we have seen, ISO 9660 is highly restrictive in several ways. Shortly after it came out, people in the UNIX community began working on an extension to make it possible to represent UNIX file systems on a CD-ROM. These extensions were named Rock Ridge, after a town in the Gene Wilder movie *Blazing Saddles*, probably because one of the committee members liked the film.

The extensions use the *System use* field in order to make Rock Ridge CD-ROMs readable on any computer. All the other fields retain their normal ISO 9660 meaning. Any system not aware of the Rock Ridge extensions just ignores them and sees a normal CD-ROM.

The extensions are divided up into the following fields:

1. PX - POSIX attributes.
2. PN - Major and minor device numbers.
3. SL - Symbolic link.
4. NM - Alternative name.
5. CL - Child location.
6. PL - Parent location.
7. RE - Relocation.
8. TF - Time stamps.

The *PX* field contains the standard UNIX *rwxrwxrwx* permission bits for the owner, group, and others. It also contains the other bits contained in the mode word, such as the SETUID and SETGID bits, and so on.

To allow raw devices to be represented on a CD-ROM, the *PN* field is present. It contains the major and minor device numbers associated with the file. In this way, the contents of the */dev* directory can be written to a CD-ROM and later reconstructed correctly on the target system.

The *SL* field is for symbolic links. It allows a file on one file system to refer to a file on a different file system.

Probably the most important field is *NM*. It allows a second name to be associated with the file. This name is not subject to the character set or length restrictions of ISO 9660, making it possible to express arbitrary UNIX file names on a CD-ROM.

The next three fields are used together to get around the ISO 9660 limit of directories that may only be nested eight deep. Using them it is possible to specify that a directory is to be relocated, and to tell where it goes in the hierarchy. It is effectively a way to work around the artificial depth limit.

Finally, the *TF* field contains the three timestamps included in each UNIX i-node, namely the time the file was created, the time it was last modified, and the time it was last accessed. Together, these extensions make it possible to copy a UNIX file system to a CD-ROM and then restore it correctly to a different system.

Joliet Extensions

The UNIX community was not the only group that wanted a way to extend ISO 9660. Microsoft also found it too restrictive (although it was Microsoft's own MS-DOS that caused most of the restrictions in the first place). Therefore Microsoft invented some extensions that were called **Joliet**. They were designed to allow Windows file systems to be copied to CD-ROM and then restored, in precisely the same way that Rock Ridge was designed for UNIX. Virtually all programs that run under Windows and use CD-ROMs support Joliet, including programs that burn CD-recordables. Usually, these programs offer a choice between the various ISO 9660 levels and Joliet.

The major extensions provided by Joliet are

1. Long file names.
2. Unicode character set.
3. Directory nesting deeper than eight levels.
4. Directory names with extensions

The first extension allows file names up to 64 characters. The second extension enables the use of the Unicode character set for file names. This extension is important for software intended for use in countries that do not use the Latin alphabet, such as Japan, Israel, and Greece. Since Unicode characters are two bytes, the maximum file name in Joliet occupies 128 bytes.

Like Rock Ridge, the limitation on directory nesting is removed by Joliet. Directories can be nested as deeply as needed. Finally, directory names can have extensions. It is not clear why this extension was included, since Windows directories virtually never use extensions, but maybe some day they will.

6.4.2 The CP/M File System

The first personal computers (then called microcomputers) came out in the early 1980s. A popular early type used the 8-bit Intel 8080 CPU and had 4 KB of RAM and a single 8-inch floppy disk with a capacity of 180 KB. Later versions used the slightly fancier (but still 8-bit) Zilog Z80 CPU, had up to 64 KB of RAM, and had a whopping 720-KB floppy disk as the mass storage device. Despite the slow speed and small amount of RAM, nearly all of these machines ran a surprisingly powerful disk-based operating system, called **CP/M** (**Control**

Program for Microcomputers) (Golden and Pechura, 1986). This system dominated its era as much as MS-DOS and later Windows dominated the IBM PC world. Two decades later, it has vanished without a trace (except for a small group of diehard fans), which gives reason to think that systems that now dominate the world may be essentially unknown when current babies become college students (Windows what?).

It is worth taking a look at CP/M for several reasons. First, it was a historically very important system and was the direct ancestor of MS-DOS. Second, current and future operating system designers who think that a computer needs 32 MB just to boot the operating system could probably learn a lot about simplicity from a system that ran quite well in 16 KB of RAM. Third, in the coming decades, embedded systems are going to be extremely widespread. Due to cost, space, weight, and power constraints, the operating systems used, for example, in watches, cameras, radios, and cellular telephones, are of necessity going to be lean and mean, not unlike CP/M. Of course, these systems do not have 8-inch floppy disks, but they may well have electronic disks using flash memory, and building a CP/M-like file system on such a device is straightforward.

The layout of CP/M in memory is shown in Fig. 6-30. At the top of main memory (in RAM) is the BIOS, which contains a basic library of 17 I/O calls used by CP/M (in this section we will describe CP/M 2.2, which was the standard version when CP/M was at the height of its popularity). These calls read and write the keyboard, screen, and floppy disk.

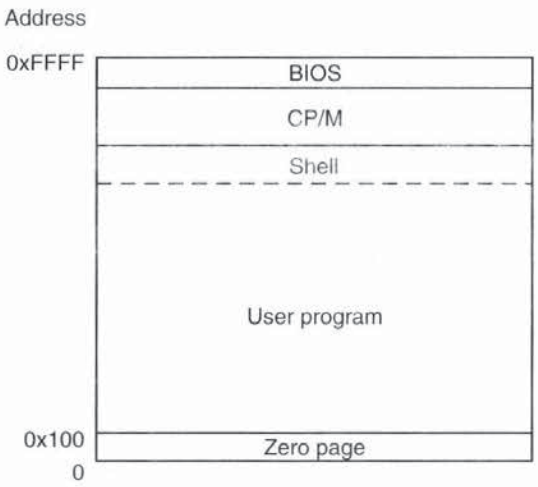


Figure 6-30. Memory layout of CP/M.

Just below the BIOS is the operating system proper. The size of the operating system in CP/M 2.2 is 3584 bytes. Amazing but true: a complete operating system in under 4 KB. Below the operating system comes the shell (command line processor), which chews up another 2 KB. The rest of memory is for user programs, except for the bottom 256 bytes, which are reserved for the hardware interrupt

vectors, a few variables, and a buffer for the current command line so user programs can get at it.

The reason for splitting the BIOS from CP/M itself (even though both are in RAM) is portability. CP/M interacts with the hardware only by making BIOS calls. To port CP/M to a new machine, all that is necessary is to port the BIOS there. Once that has been done, CP/M itself can be installed without modification.

A CP/M system has only one directory, which contains fixed-size (32-byte) entries. The directory size, although fixed for a given implementation, may be different in other implementations of CP/M All files in the system are listed in this directory. After CP/M boots, it reads in the directory and computes a bitmap containing the free disk blocks by seeing which blocks are not in any file. This bitmap, which is only 23 bytes for a 180-KB disk, is kept in memory during execution. At system shutdown time it is discarded, that is, not written back to the disk. This approach eliminates the need for a disk consistency checker (like *fsck*) and saves 1 block on the disk (percentually equivalent to saving 90 MB on a modern 16-GB disk).

When the user types a command, the shell first copies the command to a buffer in the bottom 256 bytes of memory. Then it looks up the program to be executed and reads it into memory at address 256 (above the interrupt vectors), and jumps to it. The program then begins running. It discovers its arguments by looking in the command line buffer. The program is allowed to overwrite the shell if it needs the memory. When the program finishes, it makes a system call to CP/M telling it to reload the shell (if it was overwritten) and execute it. In a nutshell, that is pretty much the whole CP/M story.

In addition to loading programs, CP/M provides 38 system calls, mostly file services, for user programs. The most important of these are reading and writing files. Before a file can be read, it must be opened. When CP/M gets an open system call, it has to read in and search the one and only directory. The directory is not kept in memory all the time to save precious RAM. When CP/M finds the entry, it immediately has the disk block numbers, since they are stored right in the directory entry, as are all the attributes. The format of a directory entry is given in Fig. 6-31.

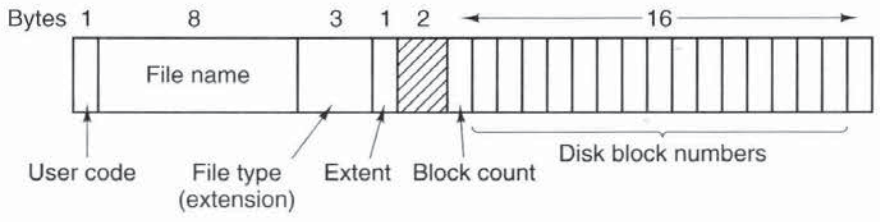


Figure 6-31. The CP/M directory entry format.

The fields in Fig. 6-31 have the following meanings. The *User code* field keeps track of which user owns the file. Although only one person can be logged

into a CP/M at any given moment, the system supports multiple users who take turns using the system. While searching for a file name, only those entries belonging to the currently logged-in user are checked. In effect, each user has a virtual directory without the overhead of managing multiple directories.

The next two fields give the name and extension of the file. The base name is up to eight characters; an optional extension of up to three characters may be present. Only upper case letters, digits, and a small number of special characters are allowed in file names. This 8 + 3 naming using upper case only was later taken over by MS-DOS.

The *Block count* field tells how many bytes this file entry contains, measured in units of 128 bytes (because I/O is actually done in 128-byte physical sectors). The last 1-KB block may not be full, so the system has no way to determine the exact size of a file. It is up to the user to put in some END-OF-FILE marker if desired. The final 16 fields contain the disk block numbers themselves. Each block is 1 KB, so that maximum file size is 16 KB. Note that physical I/O is done in units of 128-byte sectors and sizes are kept track of in sectors, but file blocks are allocated in units of 1 KB (8 sectors at a time) to avoid making the directory entry too large.

However, the CP/M designer realized that some files, even on a 180-KB floppy disk, might exceed 16 KB, so an escape hatch was built around the 16-KB limit. A file that is between 16 KB and 32 KB uses not one directory entry, but two. The first entry holds the first 16 blocks; the second entry holds the next 16 blocks. Beyond 32 KB, a third directory entry is used, and so on. The *Extent* field keeps track of the order of the directory entries so the system knows which 16 KB comes first, which comes second, and so on.

After an open call, the addresses of all the disk blocks are known, making read straightforward. The write call is also simple. It just requires allocating a free block from the bitmap in memory and then writing the block. Consecutive blocks on a file are not placed in consecutive blocks on the disk because the 8080 cannot process an interrupt and start reading the next block on time. Instead, interleaving is used to allow several blocks to be read on a single rotation.

CP/M is clearly not the last word in advanced file systems, but it is simple, fast, and can be implemented by a competent programmer in less than a week. For many embedded applications, it may be all that is needed.

6.4.3 The MS-DOS File System

To a first approximation, MS-DOS is a bigger and better version of CP/M. It runs only on Intel platforms, does not support multiprogramming, and runs only in the PC's real mode (which was originally the only mode). The shell has more features and there are more system calls, but the basic function of the operating system is still loading programs, handling the keyboard and screen, and managing the file system. It is the latter functionality that interests us here.

The MS-DOS file system was patterned closely on the CP/M file system, including the use of 8 + 3 (upper case) character file names. The first version (MS-DOS 1.0) was even limited to a single directory, like CP/M. However, starting with MS-DOS 2.0, the file system functionality was greatly expanded. The biggest improvement was the inclusion of a hierarchical file system in which directories could be nested to an arbitrary depth. This meant that the root directory (which still had a fixed maximum size) could contain subdirectories, and these could contain further subdirectories ad infinitem. Links in the style of UNIX were not permitted, so the file system formed a tree starting at the root directory.

It is common for different application programs to start out by creating a subdirectory in the root directory and put all their files there (or in subdirectories thereof), so that different applications do not conflict. Since directories are themselves just stored as files, there are no limits on the number of directories or files that may be created. Unlike CP/M, however, there is no concept of different users in MS-DOS. Consequently, the logged in user has access to all files.

To read a file, an MS-DOS program must first make an open system call to get a handle for it. The open system call specifies a path, which may be either absolute or relative to the current working directory. The path is looked up component by component until the final directory is located and read into memory. It is then searched for the file to be opened.

Although MS-DOS directories are variable sized, like CP/M, they use a fixed-size 32-byte directory entry. The format of an MS-DOS directory entry is shown in Fig. 6-32. It contains the file name, attributes, creation date and time, starting block, and exact file size. File names shorter than 8 + 3 characters are left justified and padded with spaces on the right, in each field separately. The *Attributes* field is new and contains bits to indicate that a file is read-only, needs to be archived, is hidden, or is a system file. Read-only files cannot be written. This is to protect them from accidental damage. The archived bit has no actual operating system function (i.e., MS-DOS does not examine or set it). The intention is to allow user-level archive programs to clear it upon archiving a file and to have other programs set it when modifying a file. In this way, a backup program can just examine this attribute bit on every file to see which files to back up. The hidden bit can be set to prevent a file from appearing in directory listings. Its main use is to avoid confusing novice users with files they might not understand. Finally, the system bit also hides files. In addition, system files cannot accidentally be deleted using the *del* command. The main components of MS-DOS have this bit set.

The directory entry also contains the date and time the file was created or last modified. The time is accurate only to ±2 sec because it is stored in a 2-byte field, which can store only 65,536 unique values (a day contains 86,400 unique seconds). The time field is subdivided into seconds (5 bits), minutes (6 bits), and hours (5 bits). The date counts in days using three subfields: day (5 bits), month (4 bits), and year–1980 (7 bits). With a 7-bit number for the year and time

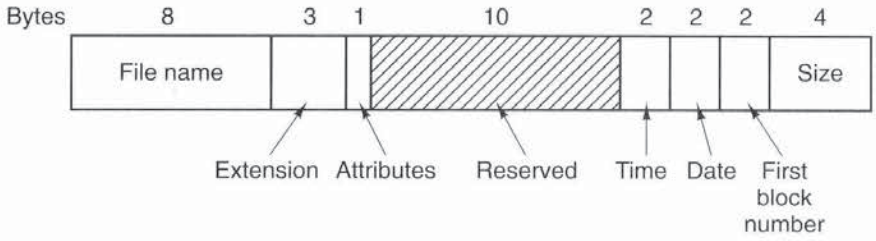


Figure 6-32. The MS-DOS directory entry.

beginning in 1980, the highest expressible year is 2107. Thus MS-DOS has a built-in Y2108 problem. To avoid catastrophe, MS-DOS users should begin with Y2108 compliance as early as possible. If MS-DOS had used the combined date and time fields as a 32-bit seconds counter, it could have represented every second exactly and delayed the catastrophe until 2116.

Unlike CP/M, which does not store the exact file size, MS-DOS does. Since a 32-bit number is used for the file size, in theory files can be as large as 4 GB. However, other limits (described below) restrict the maximum file size to 2 GB or less. A surprising large part of the entry (10 bytes) is unused.

Another way in which MS-DOS differs from CP/M is that it does not store a file's disk addresses in its directory entry, probably because the designers realized that large hard disks (by then common on minicomputers) would some day reach the MS-DOS world. Instead, MS-DOS keeps track of file blocks via a file allocation table in main memory. The directory entry contains the number of the first file block. This number is used as an index into a 64K entry FAT in main memory. By following the chain, all the blocks can be found. The operation of the FAT is illustrated in Fig. 6-14.

The FAT file system comes in three versions for MS-DOS: FAT-12, FAT-16, and FAT-32, depending on how many bits a disk address contains. Actually, FAT-32 is something of a misnomer since only the low-order 28 bits of the disk addresses are used. It should have been called FAT-28, but powers of two sound so much neater.

For all FATs, the disk block can be set to some multiple of 512 bytes (possibly different for each partition), with the set of allowed block sizes (called **cluster sizes** by Microsoft) being different for each variant. The first version of MS-DOS used FAT-12 with 512-byte blocks, giving a maximum partition size of $2^{12} \times 512$ bytes (actually only 4086×512 bytes because 10 of the disk addresses were used as special markers, such as end of file, bad block, etc. With these parameters, the maximum disk partition size was about 2 MB and the size of the FAT table in memory was 4096 entries of 2 bytes each. Using a 12-bit table entry would have been too slow.

This system worked well for floppy disks, but when hard disks came out, it became a problem. Microsoft solved the problem by allowing additional block

sizes of 1 KB, 2 KB, and 4 KB. This change preserved the structure and size of the FAT-12 table, but allowed disk partitions of up to 16 MB.

Since MS-DOS supported four disk partitions per disk drive, the new FAT-12 file system worked up to 64-MB disks. Beyond that, something had to give. What happened was the introduction of FAT-16, with 16-bit disk pointers. Additionally, block sizes of 8 KB, 16 KB, and 32 KB were permitted. (32,768 is the largest power of two that can be represented in 16 bits.) The FAT-16 table now occupied 128 KB of main memory all the time, but with the larger memories by then available, it was widely used and rapidly replaced the FAT-12 file system. The largest disk partition that can be supported by FAT-16 is 2 GB (64K entries of 32 KB each) and the largest disk 8 GB, namely four partitions of 2 GB each.

For business letters, this limit is not a problem, but for storing digital video using the DV standard, a 2-GB file holds just over 9 minutes of video. As a consequence of the fact that a PC disk can support only four partitions, the largest video that can be stored on a disk is about 38 minutes, no matter how large the disk is. This limit also means that the largest video that can be edited on line is less than 19 minutes, since both input and output files are needed.

Starting with the second release of Windows 95, the FAT-32 file system, with its 28-bit disk addresses, was introduced and the version of MS-DOS underlying Windows 95 was adapted to support FAT-32. In this system, partitions could theoretically be $2^{28} \times 2^{15}$ bytes, but they are actually limited to 2 TB (2048 GB) because internally the system keeps track of partition sizes in 512-byte sectors using a 32-bit number and $2^9 \times 2^{32}$ is 2 TB. The maximum partition size for various block sizes and all three FAT types is shown in Fig. 6-33.

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

Figure 6-33. Maximum partition size for different block sizes. The empty boxes represent forbidden combinations.

In addition to supporting larger disks, the FAT-32 file system has two other advantages over FAT-16. First, an 8-GB disk using FAT-32 can be a single partition. Using FAT-16 it has to be four partitions, which appears to the Windows user as the C:, D:, E:, and F: logical disk drives. It is up to the user to decide which file to place on which drive and keep track of what is where.

The other advantage of FAT-32 over FAT-16 is that for a given size disk partition, a smaller block size can be used. For example, for a 2-GB disk partition, FAT-16 must use 32-KB blocks, otherwise with only 64K available disk addresses, it cannot cover the whole partition. In contrast, FAT-32 can use, for example, 4-KB blocks for a 2-GB disk partition. The advantage of the smaller block size is that most files are much shorter than 32 KB. If the block size is 32 KB, a file of 10 bytes ties up 32 KB of disk space. If the average file is, say, 8 KB, then with a 32-KB block, ¾ of the disk will be wasted, not a terribly efficient way to use the disk. With an 8-KB file and a 4-KB block, there is no disk wastage, but the price paid is more RAM eaten up by the FAT. With a 4-KB block and a 2-GB disk partition, there are 512K blocks, so the FAT must have 512K entries in memory (occupying 2 MB of RAM).

MS-DOS uses the FAT to keep track of free disk blocks. Any block that is not currently allocated is marked with a special code. When MS-DOS needs a new disk block, it searches the FAT for an entry containing this code. Thus no bitmap or free list is required.

6.4.4 The Windows 98 File System

The original release of Windows 95 used the MS-DOS file system, including the use of 8 + 3 character file names and the FAT-12 and FAT-16 file systems. Starting with the second release of Windows 95, file names longer than 8 + 3 characters were permitted. In addition, FAT-32 was introduced, primarily to allow larger disk partitions larger than 2 GB and disks larger than 8 GB, which were then available. Both the long file names and FAT-32 were used in Windows 98 in the same form as in the second release of Windows 95. Below we will describe these features of the Windows 98 file system, which have been carried forward into Windows Me as well.

Since long file names are more exciting for users than the FAT structure, let us look at them first. One way to introduce long file names would have been to just invent a new directory structure. The problem with this approach is that if Microsoft had done this, people who were still in the process of converting from Windows 3 to Windows 95 or Windows 98 could not have accessed their files from both systems. A political decision was made within Microsoft that files created using Windows 98 must be accessible from Windows 3 as well (for dual-boot machines). This constraint forced a design for handling long file names that was backward compatible with the old MS-DOS 8 + 3 naming system. Since such backward compatibility constraints are not unusual in the computer industry, it is worth looking in detail at how Microsoft accomplished this goal.

The effect of this decision to be backward compatible meant that the Windows 98 directory structure had to be compatible with the MS-DOS directory structure. As we saw, this structure is just a list of 32-byte entries as shown in

Fig. 6-32. This format came directly from CP/M (which was written for the 8080), which goes to show how long (obsolete) structures can live in the computer world.

However, it was possible to now allocate the 10 unused bytes in the entries of Fig. 6-32, and that was done, as shown in Fig. 6-34. This change has nothing to do with long names, but it is used in Windows 98, so it is worth understanding.

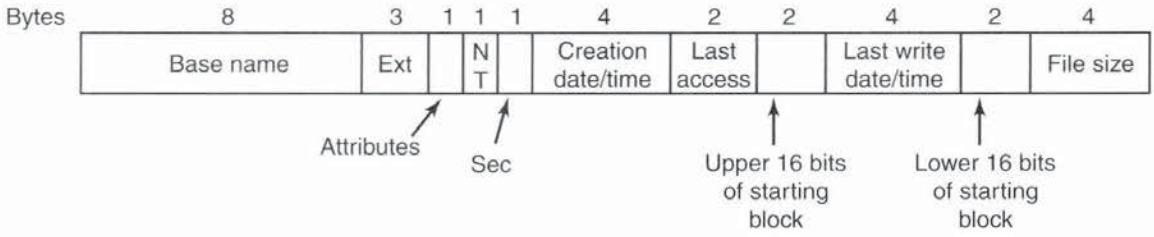


Figure 6-34. The extended MOS-DOS directory entry used in Windows 98.

The changes consist of the addition of five new fields where the 10 unused bytes used to be. The *NT* field is mostly there for some compatibility with Windows NT in terms of displaying file names in the correct case (in MS-DOS, all file names are upper case). The *Sec* field solves the problem that it is not possible to store the time of day in a 16-bit field. It provides additional bits so that the new *Creation time* field is accurate to 10 msec. Another new field is *Last access*, which stores the date (but not time) of the last access to the file. Finally, going to the FAT-32 file system means that block numbers are now 32 bits, so an additional 16-bit field is needed to store the upper 16 bits of the starting block number.

Now we come to the heart of the Windows 98 file system: how long file names are represented in a way that is backward compatible with MS-DOS. The solution chosen was to assign two names to each file: a (potentially) long file name (in Unicode, for compatibility with Windows NT), and an 8 + 3 name for compatibility with MS-DOS. Files can be accessed by either name. When a file is created whose name does not obey the MS-DOS naming rules (8 + 3 length, no Unicode, limited character set, no spaces, etc.), Windows 98 invents an MS-DOS name for it according to a certain algorithm. The basic idea is to take the first six characters of the name, convert them to upper case, if need be, and append ~1 to form the base name. If this name already exists, then the suffix ~2 is used, and so on. In addition, spaces and extra periods are deleted and certain special characters are converted to underscores. As an example, a file named *The time has come the walrus said* is assigned the MS-DOS name *THETIM~1*. If a subsequent file is created with the name *The time has come the rabbit said*, it is assigned the MS-DOS name *THETIM~2*, and so on.

Every file has an MS-DOS file name stored using the directory format of Fig. 6-34. If a file also has a long name, that name is stored in one or more directory entries directly preceding the MS-DOS file name. Each long-name entry holds up to 13 (Unicode) characters. The entries are stored in reverse order, with

the start of the file name just ahead of the MS-DOS entry and subsequent pieces before it. The format of each long-name entry is given in Fig. 6-35.

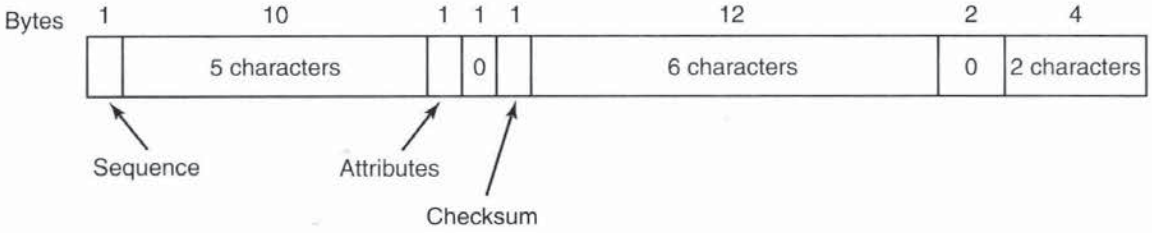


Figure 6-35. An entry for (part of) a long file name in Windows 98.

An obvious question is: "How does Windows 98 know whether a directory entry contains an MS-DOS file name or a (piece of a) long file name?" The answer lies in the *Attributes* field. For a long-name entry, this field has the value 0x0F, which represents an otherwise impossible combination of attributes. Old MS-DOS programs that read directories will just ignore it as invalid. Little do they know. The pieces of the name are sequenced using the first byte of the entry. The last part of the long name (the first entry in the sequence) is marked by adding 64 to the sequence number. Since only 6 bits are used for the sequence number, the theoretical maximum for file names is 63×13 or 819 characters. In fact they are limited to 260 characters for historical reasons.

Each long-name entry contains a *Checksum* field to avoid the following problem. First, a Windows 98 program creates a file with a long name. Second, the computer is rebooted to run MS-DOS or Windows 3. Third, an old program there then removes the MS-DOS file name from the directory but does not remove the long file name preceding it (because it does not know about it). Finally, some program creates a new file that reuses the newly-freed directory entry. At this point we have a valid sequence of long-name entries preceeding an MS-DOS file entry that has nothing to do with that file. The *Checksum* field allows Windows 98 to detect this situation by verifying that the MS-DOS file name following a long name does, in fact, belong to it. Of course, with only 1 byte being used, there is one chance in 256 that Windows 98 will not notice the file substitution.

To see an example of how long names work, consider the example of Fig. 6-36. Here we have a file called *The quick brown fox jumps over the lazy dog*. At 42-characters, it certainly qualifies as a long file name. The MS-DOS name constructed from it is *THEQUI~1* and is stored in the last entry.

Some redundancy is built into the directory structure to help detect problems in the event that an old Windows 3 program has made a mess of the directory. The sequence number byte at the start of each entry is not strictly needed since the 0x40 bit marks the first one, but it provides additional redundancy, for example. Also, the *Low* field of Fig. 6-36 (the lower half of the starting cluster) is 0 in all entries but the last one, again to avoid having old programs misinterpret it and

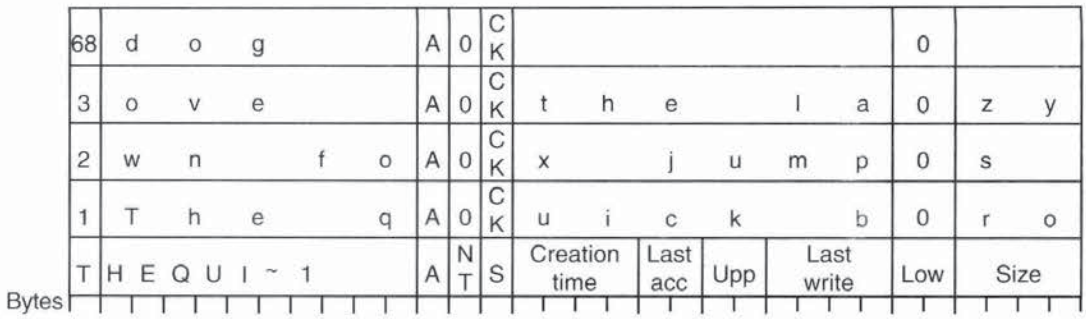


Figure 6-36. An example of how a long name is stored in Windows 98.

ruin the file system. The *NT* byte in Fig. 6-36 is used in NT and ignored in Windows 98. The *A* byte contains the attributes.

The implementation of the FAT-32 file system is conceptually similar to the implementation of the FAT-16 file system. However, instead of an array of 65,536 entries, there are as many entries as needed to cover the part of the disk with data on it. If the first million blocks are used, the table conceptually has 1 million entries. To avoid having all of them in memory at once, Windows 98 maintains a window into the table and keeps only in parts of it in memory at once.

6.4.5 The UNIX V7 File System

Even early versions of UNIX had a fairly sophisticated multiuser file system since it was derived from MULTICS. Below we will discuss the V7 file system, the one for the PDP-11 that made UNIX famous. We will examine modern versions in Chap. 10.

The file system is in the form of a tree starting at the root directory, with the addition of links, forming a directed acyclic graph. File names are up to 14 characters and can contain any ASCII characters except / (because that is the separator between components in a path) and NUL (because that is used to pad out names shorter than 14 characters). NUL has the numerical value of 0.

A UNIX directory entry contains one entry for each file in that directory. Each entry is extremely simple because UNIX uses the i-node scheme illustrated in Fig. 6-15. A directory entry contains only two fields: the file name (14 bytes) and the number of the i-node for that file (2 bytes), as shown in Fig. 6-37. These parameters limit the number of files per file system to 64K.

Like the i-node of Fig. 6-15, the UNIX i-nodes contains some attributes. The attributes contain the file size, three times (creation, last access, and last modification), owner, group, protection information, and a count of the number of directory entries that point to the i-node. The latter field is needed due to links. Whenever a new link is made to an i-node, the count in the i-node is increased. When a

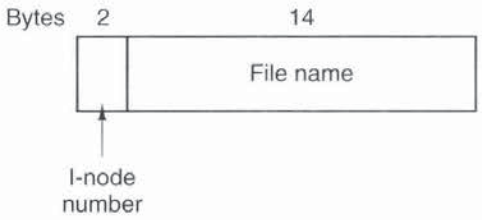


Figure 6-37. A UNIX V7 directory entry.

link is removed, the count is decremented. When it gets to 0, the i-node is reclaimed and the disk blocks are put back in the free list.

Keeping track of disk blocks is done using a generalization of Fig. 6-15 in order to handle very large files. The first 10 disk addresses are stored in the i-node itself, so for small files, all the necessary information is right in the i-node, which is fetched from disk to main memory when the file is opened. For somewhat larger files, one of the addresses in the i-node is the address of a disk block called a **single indirect block**. This block contains additional disk addresses. If this still is not enough, another address in the i-node, called a **double indirect block**, contains the address of a block that contains a list of single indirect blocks. Each of these single indirect blocks points to a few hundred data blocks. If even this is not enough, a **triple indirect block** can also be used. The complete picture is given in Fig. 6-38.

When a file is opened, the file system must take the file name supplied and locate its disk blocks. Let us consider how the path name */usr/ast/mbox* is looked up. We will use UNIX as an example, but the algorithm is basically the same for all hierarchical directory systems. First the file system locates the root directory. In UNIX its i-node is located at a fixed place on the disk. From this i-node, it locates the root directory which can be anywhere on the disk, but say block 1 in this case.

Then it reads the root directory and looks up the first component of the path, *usr*, in the root directory to find the i-node number of the file */usr*. Locating an i-node from its number is straightforward, since each one has a fixed location on the disk. From this i-node, the system locates the directory for */usr* and looks up the next component, *ast*, in it. When it has found the entry for *ast*, it has the i-node for the directory */usr/ast*. From this i-node it can find the directory itself and look up *mbox*. The i-node for this file is then read into memory and kept there until the file is closed. The lookup process is illustrated in Fig. 6-39.

Relative path names are looked up the same way as absolute ones, only starting from the working directory instead of starting from the root directory. Every directory has entries for . and .. which are put there when the directory is created. The entry . has the i-node number for the current directory, and the entry for .. has the i-node number for the parent directory. Thus, a procedure looking up *../dick/prog.c* simply looks up .. in the working directory, finds the i-node number

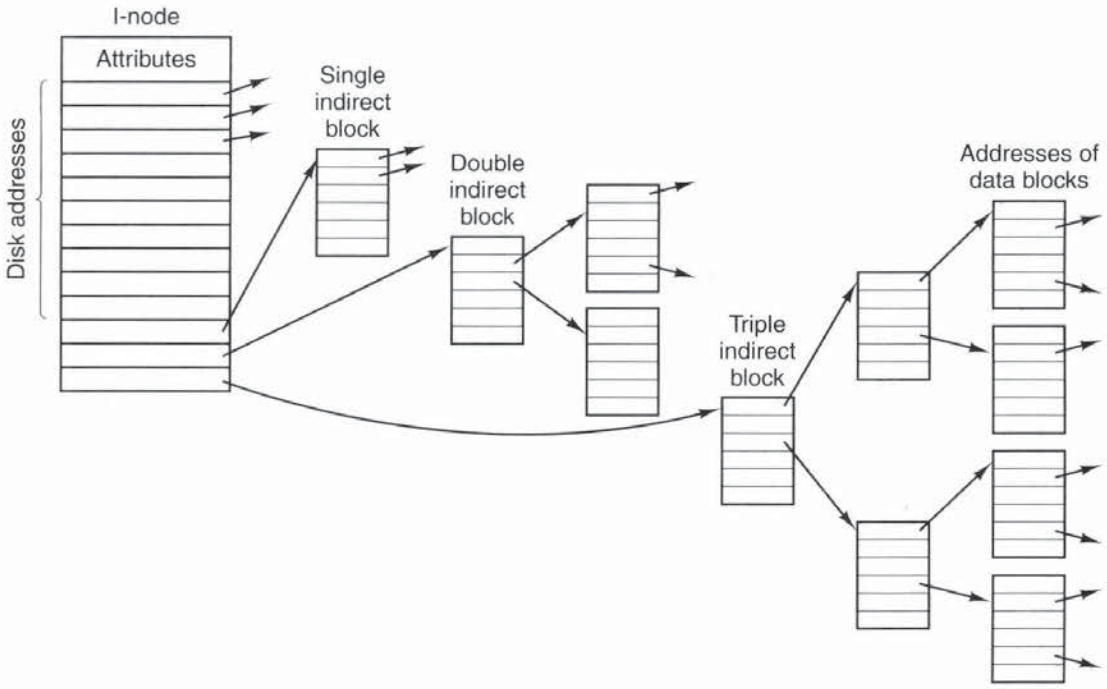


Figure 6-38. A UNIX i-node.

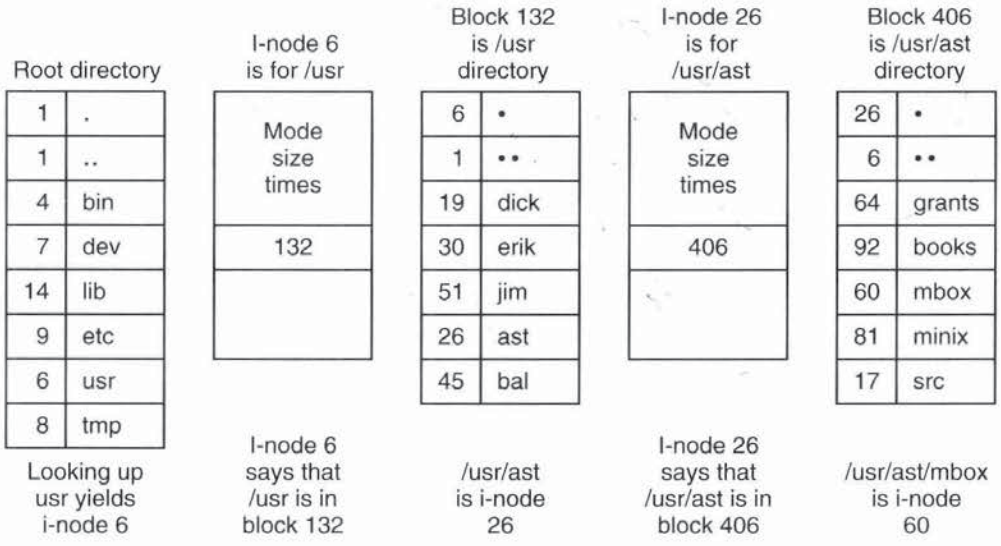


Figure 6-39. The steps in looking up */usr/ast/mbox*.

for the parent directory, and searches that directory for *dick*. No special mechanism is needed to handle these names. As far as the directory system is concerned, they are just ordinary ASCII strings, just the same as any other names.

6.5 RESEARCH ON FILE SYSTEMS

File systems have always attracted more research than other parts of the operating system and that is still the case. Some of this research relates to file system structure. Log-structured file systems and related topics are popular (Matthews et al., 1997; and Wang et al., 1999). The logical disk splits the file system into two distinct layers: the file system and the disk system (De Jonge et al., 1993). Building a file system from stackable layers is also a research topic (Heidemann and Popek, 1994).

Extensible file systems are analogous to the extensible kernels we saw in Chap 1. They allow new features to be added to the file system without having to redesign it from scratch (Karpovich et al., 1994; and Khalidi and Nelson, 1993).

Another research topic that has achieved a certain amount of popularity is making measurements of file system contents and use. People have measured what the file size distribution is, how long files live, whether all files are accessed equally, how reads compare to writes, and many other parameters (Douceur and Bolosky, 1999; Gill et al., 1994; Roselli and Lorch, 2000; and Vogels, 1999).

Other researchers have looked at file system performance and how to improve it using prefetching, caching, less copying, and other techniques. Usually, these researchers make measurements, find out where the bottlenecks are, eliminate at least one of them, and then make the measurements on the improved system to validate their results (Cao et al., 1995; Pai et al., 2000; and Patterson et al., 1995).

An area that many people rarely think about until disaster strikes is file system backup and recovery. Here, too, some new ideas have come up for better ways to do things (Chen et al., 1996; Devarakonda et al., 1996; and Hutchinson et al., 1999). Somewhat related to this is the question of what to do when a user removes a file—remove it or hide it? The Elephant file system, for example, never forgets (Santry et al., 1999a; and Santry et al., 1999b).

6.6 SUMMARY

When seen from the outside, a file system is a collection of files and directories, plus operations on them. Files can be read and written, directories can be created and destroyed, and files can be moved from directory to directory. Most modern file systems support a hierarchical directory system, in which directories may have subdirectories and these may have subsubdirectories ad infinitum.

When seen from the inside, a file system looks quite different. The file system designers have to be concerned with how storage is allocated, and how the system keeps track of which block goes with which file. Possibilities include contiguous files, linked lists, file allocation tables, and i-nodes. Different systems have different directory structures. Attributes can go in the directories or somewhere else (e.g., an i-node). Disk space can be managed using free lists of bitmaps. File system reliability is enhanced by making incremental dumps and by having a program that can repair sick file systems. File system performance is important and can be enhanced in several ways, including caching, read ahead, and carefully placing the blocks of a file close to each other. Log-structured file systems also improve performance by doing writes in large units.

Examples of file systems include ISO 9660, CP/M, MS-DOS, Windows 98, and UNIX. These differ in many ways, including how they keep track of which blocks go with which file, directory structure, and management of free disk space.

PROBLEMS

1. Give 5 different path names for the file */etc/passwd*. *Hint*: Think about the directory entries "." and "..".
2. In Windows, when a user double clicks on a file listed by Windows Explorer, a program is run and given that file as a parameter. List two different ways the operating system could know which program to run?
3. In early UNIX systems, executable files (*a.out* files) began with a very specific magic number, not one chosen at random. These files began with a header, followed by the text and data segments. Why do you think a very specific number was chosen for executable files, whereas other file types had a more-or-less random magic number as the first word?
4. In Fig. 6-4, one of the attributes is the record length. Why does the operating system ever care about this?
5. Is the open system call in UNIX absolutely essential? What would the consequences be of not having it?
6. Systems that support sequential files always have an operation to rewind files. Do systems that support random access files need this too?
7. Some operating systems provide a system call rename to give a file a new name. Is there any difference at all between using this call to rename a file, and just copying the file to a new file with the new name, followed by deleting the old one?
8. In some systems it is possible to map part of a file into memory. What restrictions must such systems impose? How is this partial mapping implemented?

9. A simple operating system only supports a single directory but allows that directory to have arbitrarily many files with arbitrarily long file names. Can something approximating a hierarchical file system be simulated? How?

10. In UNIX and Windows, random access is done by having a special system call that moves the "current position" pointer associated with a file to a given byte in the file. Propose an alternative way to do random access without having this system call.

11. Consider the directory tree of Fig. 6-10. If */usr/jim* is the working directory, what is the absolute path name for the file whose relative path name is *../ast/x*?

12. Contiguous allocation of files leads to disk fragmentation, as mentioned in the text, because some space in the last disk block will be wasted in files whose length is not an integral number of blocks. Is this internal fragmentation or external fragmentation? Make an analogy with something discussed in the previous chapter.

13. One way to use contiguous allocation of the disk and not suffer from holes is to compact the disk every time a file is removed. Since all files are contiguous, copying a file requires a seek and rotational delay to read the file, followed by the transfer at full speed. Writing the file back requires the same work. Assuming a seek time of 5 msec, a rotational delay of 4 msec, a transfer rate of 8 MB/sec, and an average file size of 8 KB, how long does it take to read a file into main memory then write it back to the disk at a new location? Using these numbers, how long would it take to compact half of a 16-GB disk?

14. In light of the answer to the previous question, does compacting the disk ever make any sense?

15. Some digital consumer devices need to store data, for example as files. Name a modern device that requires file storage and for which contiguous allocation would be a fine idea.

16. How does MS-DOS implement random access to files?

17. Consider the i-node shown in Fig. 6-15. If it contains 10 direct addresses of 4 bytes each and all disk blocks are 1024 KB, what is the largest possible file?

18. It has been suggested that efficiency could be improved and disk space saved by storing the data of a short file within the i-node. For the i-node of Fig. 6-15, how many bytes of data could be stored inside the i-node?

19. Two computer science students, Carolyn and Elinor, are having a discussion about i-nodes. Carolyn maintains that memories have gotten so large and so cheap that when a file is opened, it is simpler and faster just to fetch a new copy of the i-node into the i-node table, rather than search the entire table to see if it is already there. Elinor disagrees. Who is right?

20. Name one advantage of hard links over symbolic links and one advantage of symbolic links over hard links.

21. Free disk space can be kept track of using a free list or a bitmap. Disk addresses require D bits. For a disk with B blocks, F of which are free, state the condition under which the free list uses less space than the bitmap. For D having the value 16 bits, express your answer as a percentage of the disk space that must be free.

22. The beginning of a free space bitmap looks like this after the disk partition is first formatted: 1000 0000 0000 0000 (the first block is used by the root directory). The system always searches for free blocks starting at the lowest numbered block, so after writing file *A*, which uses 6 blocks, the bitmap looks like this: 1111 1110 0000 0000. Show the bitmap after each of the following additional actions:

(a) File *B* is written, using 5 blocks
(b) File *A* is deleted
(c) File *C* is written, using 8 blocks
(d) File *B* is deleted.

23. What would happen if the bitmap or free list containing the information about free disk blocks was completely lost due to a crash? Is there any way to recover from this disaster, or is it bye-bye disk? Discuss your answer for UNIX and the FAT-16 file system separately.

24. Oliver Owl's night job at the university computing center is to change the tapes used for overnight data backups. While waiting for each tape to complete, he works on writing his thesis that proves Shakespeare's plays were written by extraterrestrial visitors. His text processor runs on the system being backed up since that is the only one they have. Is there a problem with this arrangement?

25. We discussed making incremental dumps in some detail in the text. In Windows it is easy to tell when to dump a file because every file has an archive bit. This bit is missing in UNIX. How do UNIX backup programs know which files to dump?

26. Suppose that file 21 in Fig. 6-24 was not modified since the last dump. In what way would the four bitmaps of Fig. 6-25 be different?

27. It has been suggested that the first part of each UNIX file be kept in the same disk block as its i-node. What good would this do?

28. Consider Fig. 6-26. Is it possible that for some particular block number the counter in *both* lists have the value 2? How should this problem be corrected?

29. The performance of a file system depends upon the cache hit rate (fraction of blocks found in the cache). If it takes 1 msec to satisfy a request from the cache, but 40 msec to satisfy a request if a disk read is needed, give a formula for the mean time required to satisfy a request if the hit rate is h. Plot this function for values of h from 0 to 1.0.

30. A floppy disk has 40 cylinders. A seek takes 6 msec per cylinder moved. If no attempt is made to put the blocks of a file close to each other, two blocks that are logically consecutive (i.e., follow one another in the file) will be about 13 cylinders apart, on the average. If, however, the operating system makes an attempt to cluster related blocks, the mean interblock distance can be reduced to 2 cylinders (for example). How long does it take to read a 100-block file in both cases, if the rotational latency is 100 msec and the transfer time is 25 msec per block?

31. Consider the idea behind Fig. 6-20, but now for a disk with a mean seek time of 8 msec, a rotational rate of 15,000 rpm, and 262,144 bytes per track. What are the data rates for block sizes of 1 KB, 2 KB, and 4 KB, respectively?

32. A certain file system uses 2-KB disk blocks. The median file size is 1 KB. If all files were exactly 1 KB, what fraction of the disk space would be wasted? Do you think the wastage for a real file system will be higher than this number or lower than it? Explain your answer.

33. CP/M was designed to run with a small floppy disk as the mass storage device. Suppose that it were to be ported to a modern computer with a large hard disk. What is the largest possible disk that could be supported without changing the size of the directory format shown in Fig. 6-31? The fields within the directory and other system parameters may be changed if necessary. What changes would you make?

34. The MS-DOS FAT-16 table contains 64K entries. Suppose that one of the bits had been needed for some other purpose and that the table contained exactly 32,768 entries instead. With no other changes, what would the largest MS-DOS file have been under this conditions?

35. Files in MS-DOS have to compete for space in the FAT-16 table in memory. If one file uses k entries, that is k entries that are not available to any other file, what constraint does this place on the total length of all files combined?

36. A UNIX file system has 1-KB blocks and 4-byte disk addresses. What is the maximum file size if i-nodes contain 10 direct entries, and one single, double, and triple indirect entry each?

37. How many disk operations are needed to fetch the i-node for the file */usr/ast/courses/os/handout.t*? Assume that the i-node for the root directory is in memory, but nothing else along the path is in memory. Also assume that all directories fit in one disk block.

38. In many UNIX systems, the i-nodes are kept at the start of the disk. An alternative design is to allocate an i-node when a file is created and put the i-node at the start of the first block of the file. Discuss the pros and cons of this alternative.

39. Write a program that reverses the bytes of a file, so that the last byte is now first and the first byte is now last. It must work with an arbitrarily long file, but try to make it reasonably efficient.

40. Write a program that starts at a given directory and descends the file tree from that point recording the sizes of all the files it finds. When it is all done, it should print a histogram of the file sizes using a bin width specified as a parameter (e.g., with 1024, file sizes of 0 to 1023 go in one bin, 1024 to 2047 go in the next bin, etc.).

41. Write a program that scans all directories in a UNIX file system and finds and locates all i-nodes with a hard link count of two or more. For each such file, it lists together all file names that point to the file.

42. Write a new version of the UNIX *ls* program. This version takes as an argument one or more directory names and for each directory lists all the files in that directory, one line per file. Each field should be formatted in a reasonable way given its type. List only the first disk address, if any.

43. Write CP/M in C or C++. Look on the Web to find information about it.

7

MULTIMEDIA OPERATING SYSTEMS

Digital movies, video clips, and music are becoming an increasingly common way to present information and entertainment using a computer. Audio and video files can be stored on a disk and played back on demand. However, their characteristics are very different from the traditional text files that current file systems were designed for. As a consequence, new kinds of file systems are needed to handle them. Stronger yet, storing and playing back audio and video puts new demands on the scheduler and other parts of the operating system as well. In the sections that follow, we will study many of these issues and their implications for operating systems that are designed to handle multimedia.

Usually, digital movies go under the name **multimedia**, which literally means more than one medium. Under this definition, this book is a multimedia work. After all, it contains two media: text and images (the figures). However, most people use the term "multimedia" to mean a document containing two or more *continuous* media, that is media that must be played back over some time interval. In this book, we will use the term multimedia in this sense.

Another term that is somewhat ambiguous is "video." In a technical sense, it is just the image portion of a movie (as opposed to the sound portion). In fact, camcorders and televisions often have two connectors, one labeled "video" and one labeled "audio," since the signals are separate. However, the term "digital video" normally refers to the complete product, with both image and sound. Below we will use the term "movie" to refer to the complete product. Note that a movie in this sense need not be a two-hour long film produced by a Hollywood

studio at a cost exceeding that of a Boeing 747. A 30-sec news clip downloaded from CNN's home page over the Internet is also a movie under our definition. We will also call these "video clips" when we are referring to very short movies.

7.1 INTRODUCTION TO MULTIMEDIA

Before getting into the technology of multimedia, a few words about its current and future uses are perhaps helpful to set the stage. On a single computer, multimedia often means playing a prerecorded movie from a **DVD** (**Digital Versatile Disk**). DVDs are optical disks that use the same 120-mm polycarbonate (plastic) blanks that CD-ROMs use, but are recorded at a higher density, giving a capacity of between 5 GB and 17 GB, depending on the format.

Another use of multimedia is for downloading video clips over the Internet. Many Web pages have items that can be clicked on to download short movies. At 56 Kbps, downloading even a short video clip takes a long time, but as faster distribution technologies take over, such as cable TV and **ADSL** (**Asymmetric Digital Subscriber Loop**), the presence of video clips on the Internet will skyrocket.

Another area in which multimedia must be supported is in the creation of videos themselves. Multimedia editing systems exist and for best performance need to run on an operating system that supports multimedia as well as traditional work.

Yet another arena where multimedia is becoming important is in computer games. Games often run short video clips to depict some kind of action. The clips are usually short, but there are many of them and the correct one is selected dynamically, depending on some action the user has taken. These are increasingly sophisticated

Finally, the holy grail of the multimedia world is **video on demand**, by which people mean the ability for consumers at home to select a movie using their television remote control (or mouse) and have it displayed on their TV set (or computer monitor) on the spot. To enable video on demand, a special infrastructure is needed. In Fig. 7-1 we see two possible video-on-demand infrastructures. Each one contains three essential components: one or more video servers, a distribution network, and a set-top box in each house for decoding the signal. The **video server** is a powerful computer that stores many movies in its file system and plays them back on demand. Sometimes mainframes are used as video servers, since connecting, say, 1000 large disks to a mainframe is straightforward, whereas connecting 1000 disks of any kind to a personal computer is a serious problem. Much of the material in the following sections is about video servers and their operating systems.

The distribution network between the user and the video server must be capable of transmitting data at high speed and in real time. The design of such networks is interesting and complex, but falls outside the scope of this book. We

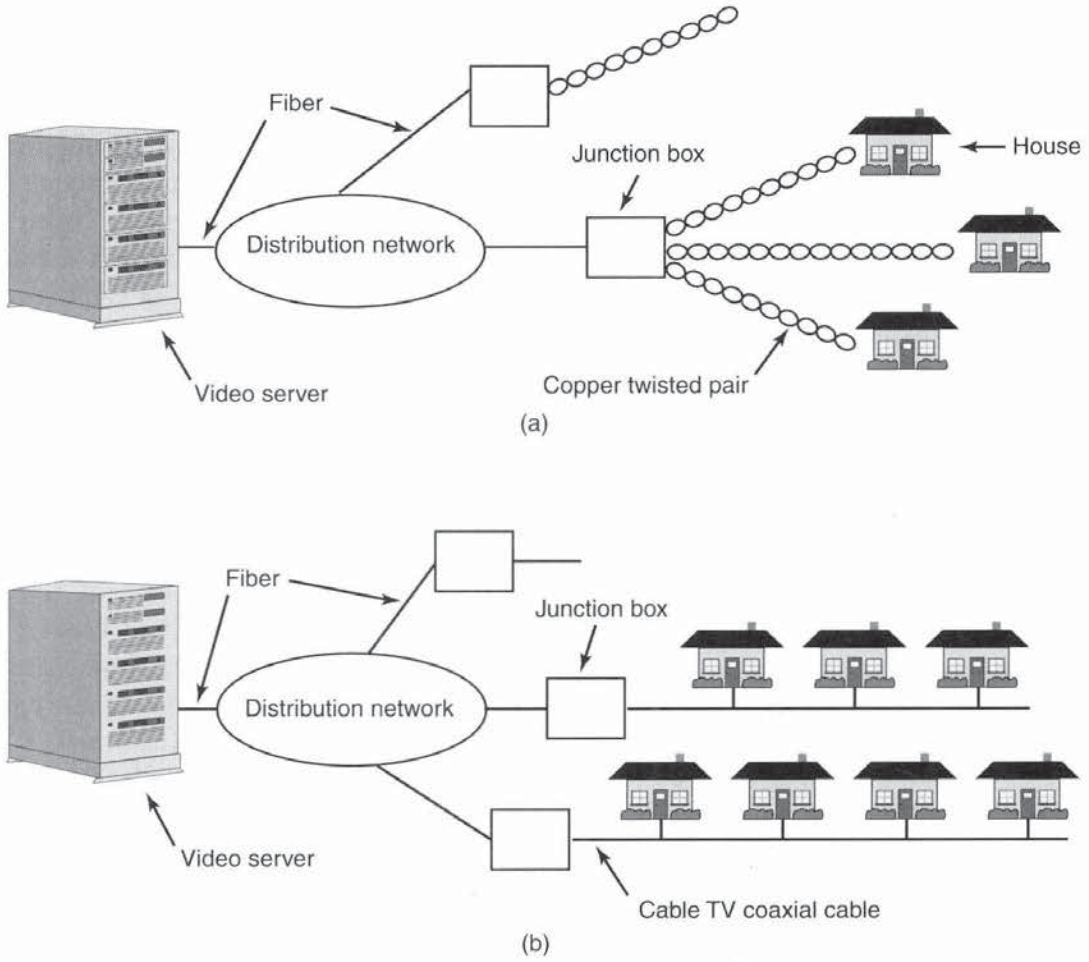


Figure 7-1. Video on demand using different local distribution technologies. (a) ADSL. (b) Cable TV.

will not say any more about them except to note that these networks always use fiber optics from the video server down to a junction box in each neighborhood where customers live. In ADSL systems, which are provided by telephone companies, the existing twisted-pair telephone line provides the last kilometer or so of transmission. In cable TV systems, which are provided by cable operators, existing cable TV wiring is used for the local distribution. ADSL has the advantage of giving each user a dedicated channel, hence guaranteed bandwidth, but the bandwidth is low (a few megabits/sec) due to limitations of existing telephone wire. Cable TV uses high-bandwidth coaxial cable (at gigabits/sec), but many users have to share the same cable, giving contention for it and no guaranteed bandwidth to any individual user.

The last piece of the system is the **set-top box**, where the ADSL or TV cable terminates. This device is, in fact, a normal computer, with certain special chips

for video decoding and decompression. As a minimum, it contains a CPU, RAM, ROM, and interface to ADSL or the cable.

An alternative to a set-top box is to use the customer's existing PC and display the movie on the monitor. Interestingly enough, the reason set-top boxes are even considered given that most customers probably already have a computer, is that video-on-demand operators expect that people will want to watch movies in their living rooms, which usually contain a TV but rarely a computer. From a technical perspective, using a personal computer instead of a set-top box makes far more sense since it is more powerful, has a large disk, and has a far higher resolution display. Either way, we will often make a distinction between the video server and the client process at the user end that decodes and displays the movie. In terms of system design, however, it does not matter much if the client process runs on a set-top box or on a PC. For a desktop video editing system, all the processes run on the same machine, but we will continue to use the terminology of server and client to make it clear which process is doing what.

Getting back to multimedia itself, it has two key characteristics that must be well understood to deal with it successfully:

1. Multimedia uses extremely high data rates.
2. Multimedia requires real-time playback.

The high data rates come from the nature of visual and acoustic information. The eye and the ear can process prodigious amounts of information per second, and have to be fed at that rate to produce an acceptable viewing experience. The data rates of a few digital multimedia sources and some common hardware devices are listed in Fig. 7-2. We will discuss some of these encoding formats later in this chapter. What should be noted is the high data rates multimedia requires, the need for compression, and the amount of storage that is required. For example, an uncompressed 2-hour HDTV movie fills a 570-GB file. A video server that stores 1000 such movies needs 570 TB of disk space, a nontrivial amount by current standards. What is also of note is that without data compression, current hardware cannot keep up with the data rates produced. We will examine video compression later in this chapter.

The second demand that multimedia puts on a system is the need for real-time data delivery. The video portion of a digital movie consists of some number of frames per second. The NTSC system, used in North and South America and Japan, runs at 30 frames/sec (29.97 for the purist), whereas the PAL and SECAM systems, used in most of the rest of the world, runs at 25 frames/sec (25.00 for the purist). Frames must be delivered at precise intervals of ca. 33.3 msec or 40 msec, respectively, or the movie will look choppy.

Officially NTSC stands for National Television Standards Committee, but the poor way color was hacked into the standard when color television was invented

Source	Mbps	GB/hr
Telephone (PCM)	0.064	0.03
MP3 music	0.14	0.06
Audio CD	1.4	0.62
MPEG-2 movie (640 × 480)	4	1.76
Digital camcorder (720 × 480)	25	11
Uncompressed TV (640 × 480)	221	97
Uncompressed HDTV (1280 × 720)	648	288

Device	Mbps
Fast Ethernet	100
EIDE disk	133
ATM OC-3 network	156
SCSI UltraWide disk	320
IEEE 1394 (FireWire)	400
Gigabit Ethernet	1000
SCSI Ultra-160 disk	1280

Figure 7-2. Some data rates for multimedia and high-performance I/O devices. Note that 1 Mbps is 10^6 bits/sec but 1 GB is 2^{30} bytes.

has led to the industry joke that it really stands for Never Twice the Same Color. PAL stands for Phase Alternating Line. Technically it is the best of the systems. SECAM is used in France (and was intended to protect French TV manufacturers from foreign competition) and stands for SEquentiel Couleur Avec Memoire. SECAM is also used in Eastern Europe because when television was introduced there, the then-Communist governments wanted to keep everyone from watching German (PAL) television, so they chose an incompatible system.

The ear is more sensitive than the eye, so a variance of even a few milliseconds in delivery times will be noticeable. Variability in delivery rates is called **jitter** and must be strictly bounded for good performance. Note that jitter is not the same as delay. If the distribution network of Fig. 7-1 uniformly delays all the bits by exactly 5.000 sec, the movie will start slightly later, but will look fine. On the other hand, if it randomly delays frames by between 100 and 200 msec, the movie will look like an old Charlie Chaplin film, no matter who is starring.

The real-time properties required to play back multimedia acceptably are often described by **quality of service** parameters. They include average bandwidth available, peak bandwidth available, minimum and maximum delay (which together bound the jitter), and bit loss probability. For example, a network operator could offer a service guaranteeing an average bandwidth of 4 Mbps, 99% of the transmission delays in the interval 105 to 110 msec, and a bit loss rate of 10^{-10}, which would be fine for MPEG-2 movies. The operator could also offer a cheaper, lower-grade service, with an average bandwidth of 1 Mbps (e.g., ADSL), in which case the quality would have to be compromised somehow, possibly by lowering the resolution, dropping the frame rate, or discarding the color information and showing the movie in black and white.

The most common way to provide quality of service guarantees is to reserve capacity in advance for each new customer. The resources reserved include a portion of the CPU, memory buffers, disk transfer capacity, and network bandwidth. If a new customer comes along and wants to watch a movie, but the video server

or network calculates that it does not have sufficient capacity for another customer, it has to reject the new customer to avoid degrading the service to current customers. As a consequence, multimedia servers need resource reservation schemes and an **admission control algorithm** to decide when they can handle more work.

7.2 MULTIMEDIA FILES

In most systems, an ordinary text file consists of a linear sequence of bytes without any structure that the operating system knows about or cares about. With multimedia, the situation is more complicated. To start with, video and audio are completely different. They are captured by distinct devices (CCD chip versus microphone), have a different internal structure (video has 25-30 frames/sec; audio has 44,100 samples/sec), and they are played back by different devices (monitor versus loudspeakers).

Furthermore, most Hollywood movies are now aimed at a worldwide audience, most of which does not speak English. The latter point is dealt with in one of two ways. For some countries, an additional sound track is produced, with the voices dubbed into the local language (but not the sound effects). In Japan, all televisions have two sound channels to allow the viewer to listen to foreign films in either the original language or in Japanese. A button on the remote control is used for language selection. In still other countries, the original sound track is used, with subtitles in the local language.

In addition, many TV movies now provide closed-caption subtitles in English as well, to allow English-speaking but hearing-impaired people to watch the movie. The net result is that a digital movie may actually consist of many files: one video file, multiple audio files, and multiple text files with subtitles in various languages. DVDs have the capability for storing up to 32 language and subtitle files. A simple set of multimedia files is shown in Fig. 7-3. We will explain the meaning of fast forward and fast backward later in this chapter.

As a consequence, the file system needs to keep track of multiple "subfiles" per file. One possible scheme is to manage each subfile as a traditional file (e.g., using an i-node to keep track of its blocks) and to have a new data structure that lists all the subfiles per multimedia file. Another way is to invent a kind of two-dimensional i-node, with each column listing the blocks of each subfile. In general, the organization must be such that the viewer can dynamically choose which audio and subtitle tracks to use at the time the movie is viewed.

In all cases, some way to keep the subfiles synchronized is also needed so that when the selected audio track is played back it remains in sync with the video. If the audio and video get even slightly out of sync, the viewer may hear an actor's words before or after his lips move, which is easily detected and fairly annoying.

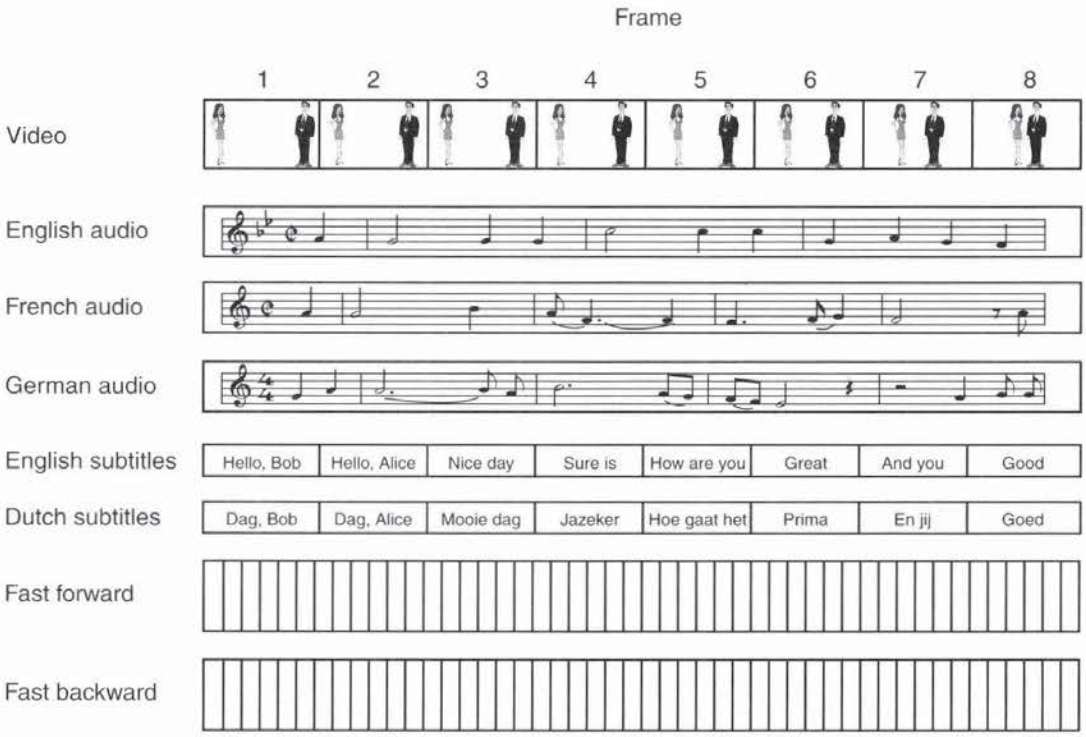


Figure 7-3. A movie may consist of several files.

To better understand how multimedia files are organized, it is necessary to understand how digital audio and video work in some detail. We will now give an introduction to these topics.

7.2.1 Audio Encoding

An audio (sound) wave is a one-dimensional acoustic (pressure) wave. When an acoustic wave enters the ear, the eardrum vibrates, causing the tiny bones of the inner ear to vibrate along with it, sending nerve pulses to the brain. These pulses are perceived as sound by the listener. In a similar way, when an acoustic wave strikes a microphone, the microphone generates an electrical signal, representing the sound amplitude as a function of time.

The frequency range of the human ear runs from 20 Hz to 20,000 Hz, although some animals, notably dogs, can hear higher frequencies. The ear hears logarithmically, so the ratio of two sounds with amplitudes A and B is conventionally expressed in **dB** (**decibels**) according to the formula

$$\text{dB} = 20 \log_{10}(A/B)$$

If we define the lower limit of audibility (a pressure of about 0.0003 dyne/cm^2) for a 1-kHz sine wave as 0 dB, an ordinary conversation is about 50 dB and the

pain threshold is about 120 dB, a dynamic range of a factor of 1 million. To avoid any confusion, *A* and *B* above are *amplitudes*. If we were to use the power level, which is proportional to the square of the amplitude, the coefficient of the logarithm would be 10, not 20.

Audio waves can be converted to digital form by an **ADC** (**Analog Digital Converter**). An ADC takes an electrical voltage as input and generates a binary number as output. In Fig. 7-4(a) we see an example of a sine wave. To represent this signal digitally, we can sample it every ΔT seconds, as shown by the bar heights in Fig. 7-4(b). If a sound wave is not a pure sine wave, but a linear superposition of sine waves where the highest frequency component present is f, then it is sufficient to make samples at a frequency $2f$. This result was proven mathematically by H. Nyquist in 1924. Sampling more often is of no value since the higher frequencies that such sampling could detect are not present.

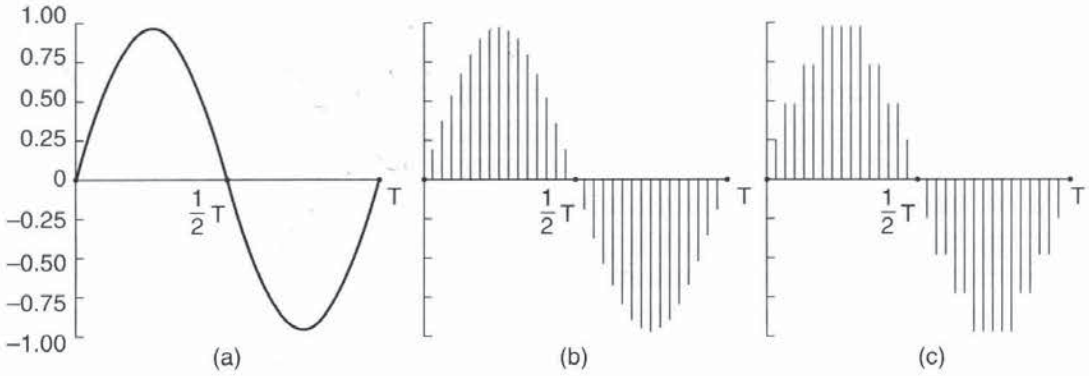


Figure 7-4. (a) A sine wave. (b) Sampling the sine wave. (c) Quantizing the samples to 4 bits.

Digital samples are never exact. The samples of Fig. 7-4(c) allow only nine values, from −1.00 to +1.00 in steps of 0.25. Consequently, 4 bits are needed to represent all of them. An 8-bit sample would allow 256 distinct values. A 16-bit sample would allow 65,536 distinct values. The error introduced by the finite number of bits per sample is called the **quantization noise**. If it is too large, the ear detects it.

Two well-known examples of sampled sound are the telephone and audio compact discs. **Pulse code modulation** is used within the telephone system and uses 7-bit (North America and Japan) or 8-bit (Europe) samples 8000 times per second. This system gives a data rate of 56,000 bps or 64,000 bps. With only 8000 samples/sec, frequencies above 4 kHz are lost.

Audio CDs are digital with a sampling rate of 44,100 samples/sec, enough to capture frequencies up to 22,050 Hz, which is good for people, bad for dogs. The samples are 16 bits each, and are linear over the range of amplitudes. Note that 16-bit samples allow only 65,536 distinct values, even though the dynamic range

of the ear is about 1 million when measured in steps of the smallest audible sound. Thus using only 16 bits per sample introduces some quantization noise (although the full dynamic range is not covered—CDs are not supposed to hurt). With 44,100 samples/sec of 16 bits each, an audio CD needs a bandwidth of 705.6 Kbps for monaural and 1.411 Mbps for stereo (see Fig. 7-2). Audio compression is possible based on psychoacoustic models of how human hearing works. A compression of 10x is possible using the MPEG layer 3 (MP3) system. Portable music players for this format have been common in recent years.

Digitized sound can easily be processed by computers in software. Dozens of programs exist for personal computers to allow users to record, display, edit, mix, and store sound waves from multiple sources. Virtually all professional sound recording and editing is digital nowadays.

7.2.2 Video Encoding

The human eye has the property that when an image is flashed on the retina, it is retained for some number of milliseconds before decaying. If a sequence of images is flashed at 50 or more images/sec, the eye does not notice that it is looking at discrete images. All video- and film-based motion picture systems exploit this principle to produce moving pictures.

To understand video systems, it is best to start with simple, old-fashioned black-and-white television. To represent the two-dimensional image in front of it as a one-dimensional voltage as a function of time, the camera scans an electron beam rapidly across the image and slowly down it, recording the light intensity as it goes. At the end of the scan, called a **frame**, the beam retraces. This intensity as a function of time is broadcast, and receivers repeat the scanning process to reconstruct the image. The scanning pattern used by both the camera and the receiver is shown in Fig. 7-5. (As an aside, CCD cameras integrate rather than scan, but some cameras and all CRT monitors do scan.)

The exact scanning parameters vary from country to country. NTSC has 525 scan lines, a horizontal to vertical aspect ratio of 4:3, and 30 frames/sec. The European PAL and SECAM systems have 625 scan lines, the same aspect ratio of 4:3, and 25 frames/sec. In both systems, the top few and bottom few lines are not displayed (to approximate a rectangular image on the original round CRTs). Only 483 of the 525 NTSC scan lines (and 576 of the 625 PAL/SECAM scan lines) are displayed.

While 25 frames/sec is enough to capture smooth motion, at that frame rate many people, especially older ones, will perceive the image to flicker (because the old image has faded off the retina before the new one appears). Rather than increase the frame rate, which would require using more scarce bandwidth, a different approach is taken. Instead of displaying the scan lines in order fromtop to bottom, first all the odd scan lines are displayed, then the even ones are displayed. Each of these half frames is called a **field**. Experiments have shown that although

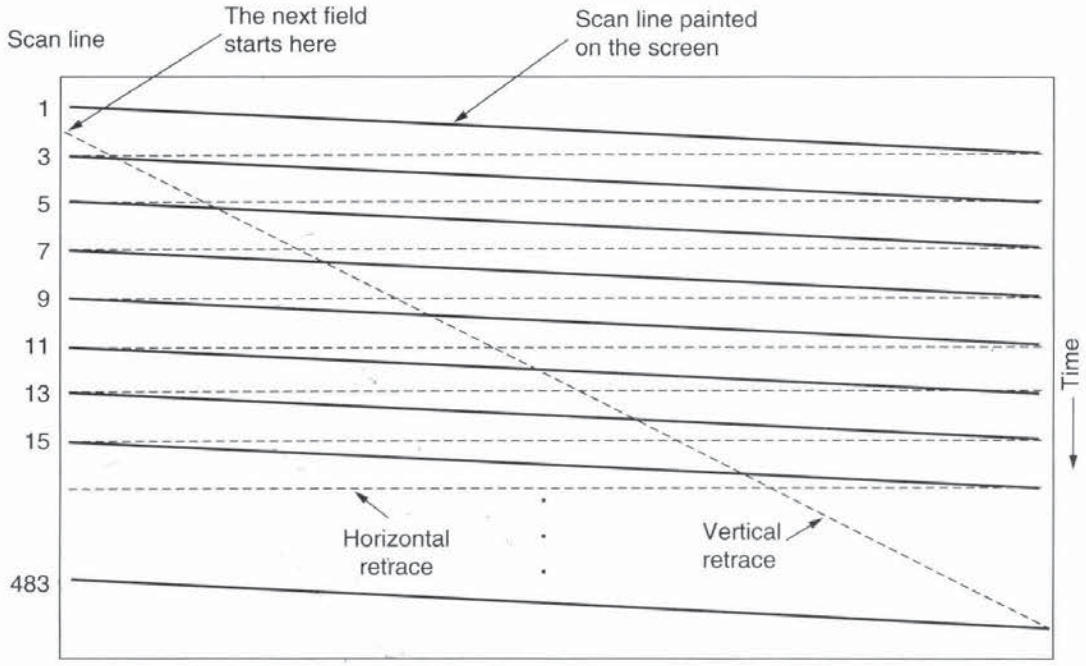


Figure 7-5. The scanning pattern used for NTSC video and television.

people notice flicker at 25 frames/sec, they do not notice it at 50 fields/sec. This technique is called **interlacing**. Noninterlaced television or video is said to be **progressive**.

Color video uses the same scanning pattern as monochrome (black and white), except that instead of displaying the image with one moving beam, three beams moving in unison are used. One beam is used for each of the three additive primary colors: red, green, and blue (RGB). This technique works because any color can be constructed from a linear superposition of red, green, and blue with the appropriate intensities. However, for transmission on a single channel, the three color signals must be combined into a single **composite** signal.

To allow color transmissions to be viewed on black-and-white receivers, all three systems linearly combine the RGB signals into a **luminance** (brightness) signal, and two **chrominance** (color) signals, although they all use different coefficients for constructing these signals from the RGB signals. Interestingly enough, the eye is much more sensitive to the luminance signal than to the chrominance signals, so the latter need not be transmitted as accurately. Consequently, the luminance signal can be broadcast at the same frequency as the old black-and-white signal, so it can be received on black-and-white television sets. The two chrominance signals are broadcast in narrow bands at higher frequencies. Some television sets have knobs or controls labeled brightness, hue, and saturation (or brightness, tint and color) for controlling these three signals separately.

Understanding luminance and chrominance is necessary for understanding how video compression works.

So far we have looked at analog video. Now let us turn to digital video. The simplest representation of digital video is a sequence of frames, each consisting of a rectangular grid of picture elements, or **pixels**. For color video, 8 bits per pixel are used for each of the RGB colors, giving 16 million colors, which is enough. The human eye cannot even distinguish this many colors, let alone more.

To produce smooth motion, digital video, like analog video, must display at least 25 frames/sec. However, since good quality computer monitors often rescan the screen from images stored in video RAM at 75 times per second or more, interlacing is not needed Consequently, all computer monitors use progressive scanning. Just repainting (i.e., redrawing) the same frame three times in a row is enough to eliminate flicker.

In other words, smoothness of motion is determined by the number of *different* images per second, whereas flicker is determined by the number of times the screen is painted per second. These two parameters are different. A still image painted at 20 frames/sec will not show jerky motion but it will flicker because one frame will decay from the retina before the next one appears. A movie with 20 different frames per second, each of which is painted four times in a row at 80 Hz, will not flicker, but the motion will appear jerky.

The significance of these two parameters becomes clear when we consider the bandwidth required for transmitting digital video over a network. Current computer monitors all use the 4:3 aspect ratio so they can use inexpensive, mass-produced picture tubes designed for the consumer television market. Common configurations are 640×480 (VGA), 800×600 (SVGA), and 1024×768 (XGA). An XGA display with 24 bits per pixel and 25 frames/sec needs to be fed at 472 Mbps. Doubling this rate to avoid flicker is not attractive. A better solution is to transmit 25 frames/sec and have the computer store each one and paint it twice. Broadcast television does not use this strategy because television sets do not have memory, and in any event, analog signals cannot be stored in RAM without first converting them to digital form, which requires extra hardware. As a consequence, interlacing is needed for broadcast television but not for digital video.

7.3 VIDEO COMPRESSION

It should be obvious by now that manipulating multimedia material in uncompressed form is completely out of the question—it is much too big. The only hope is that massive compression is possible. Fortunately, a large body of research over the past few decades has led to many compression techniques and algorithms that make multimedia transmission feasible. In this section we will study some methods for compressing multimedia data, especially images. For more detail, see (Fluckiger, 1995; and Steinmetz and Nahrstedt, 1995).

All compression systems require two algorithms: one for compressing the data at the source, and another for decompressing it at the destination. In the literature, these algorithms are referred to as the **encoding** and **decoding** algorithms, respectively. We will use this terminology here, too.

These algorithms have certain asymmetries that are important to understand. First, for many applications, a multimedia document, say, a movie will only be encoded once (when it is stored on the multimedia server) but will be decoded thousands of times (when it is viewed by customers). This asymmetry means that it is acceptable for the encoding algorithm to be slow and require expensive hardware provided that the decoding algorithm is fast and does not require expensive hardware. On the other hand, for real-time multimedia, such as video conferencing, slow encoding is unacceptable. Encoding must happen on-the-fly, in real time.

A second asymmetry is that the encode/decode process need not be invertible. That is, when compressing a file, transmitting it, and then decompressing it, the user expects to get the original back, accurate down to the last bit. With multimedia, this requirement does not exist. It is usually acceptable to have the video signal after encoding and then decoding be slightly different than the original. When the decoded output is not exactly equal to the original input, the system is said to be **lossy**. All compression systems used for multimedia are lossy because they give much better compression.

7.3.1 The JPEG Standard

The **JPEG** (**Joint Photographic Experts Group**) standard for compressing continuous-tone still pictures (e.g., photographs) was developed by photographic experts working under the joint auspices of ITU, ISO, and IEC, another standards body. It is important for multimedia because, to a first approximation, the multimedia standard for moving pictures, MPEG, is just the JPEG encoding of each frame separately, plus some extra features for interframe compression and motion compensation. JPEG is defined in International Standard 10918. It has four modes and many options, but we will only be concerned with the way it is used for 24-bit RGB video and will leave out many of the details.

Step 1 of encoding an image with JPEG is block preparation. For the sake of specificity, let us assume that the JPEG input is a 640×480 RGB image with 24 bits/pixel, as shown in Fig. 7-6(a). Since using luminance and chrominance gives better compression, the luminance and two chrominance signals are computed from the RGB values. For NTSC they are called Y, I, and Q, respectively. For PAL they are called Y, U, and V, respectively, and the formulas are different. Below we will use the NTSC names, but the compression algorithm is the same.

Separate matrices are constructed for Y, I, and Q, each with elements in the range 0 to 255. Next, square blocks of four pixels are averaged in the I and Q matrices to reduce them to 320×240. This reduction is lossy, but the eye barely

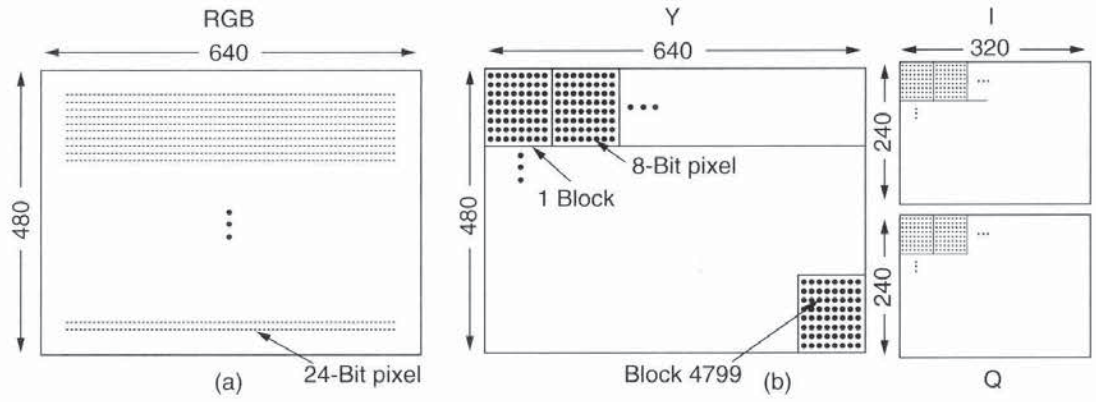


Figure 7-6. (a) RGB input data. (b) After block preparation.

notices it since the eye responds to luminance more than to chrominance. Nevertheless, it compresses the data by a factor of two. Now 128 is subtracted from each element of all three matrices to put 0 in the middle of the range. Finally, each matrix is divided up into 8×8 blocks. The Y matrix has 4800 blocks; the other two have 1200 blocks each, as shown in Fig. 7-6(b).

Step 2 of JPEG is to apply a DCT (Discrete Cosine Transformation) to each of the 7200 blocks separately. The output of each DCT is an 8×8 matrix of DCT coefficients. DCT element (0, 0) is the average value of the block. The other elements tell how much spectral power is present at each spatial frequency. In theory, a DCT is lossless, but in practice using floating-point numbers and transcendental functions always introduces some roundoff error that results in a little information loss. Normally, these elements decay rapidly with distance from the origin, (0, 0), as suggested by Fig. 7-7(b).

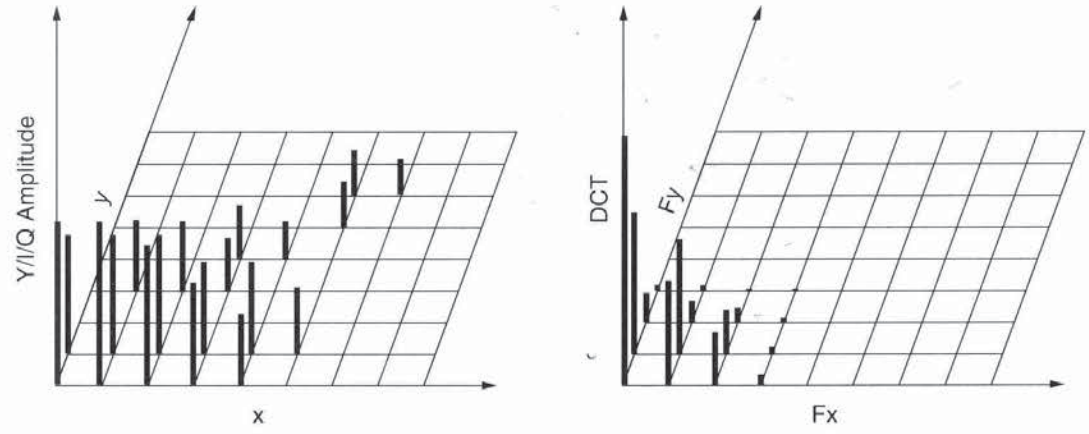


Figure 7-7. (a) One block of the Y matrix. (b) The DCT coefficients.

Once the DCT is complete, JPEG moves on to step 3, which is called **quantization**, in which the less important DCT coefficients are wiped out. This (lossy)

transformation is done by dividing each of the coefficients in the 8×8 DCT matrix by a weight taken from a table. If all the weights are 1, the transformation does nothing. However, if the weights increase sharply from the origin, higher spatial frequencies are dropped quickly.

An example of this step is given in Fig. 7-8. Here we see the initial DCT matrix, the quantization table, and the result obtained by dividing each DCT element by the corresponding quantization table element. The values in the quantization table are not part of the JPEG standard. Each application must supply its own quantization table, giving it the ability to control its own loss-compression trade-off.

DCT Coefficients

150	80	40	14	4	2	1	0
92	75	36	10	6	1	0	0
52	38	26	8	7	4	0	0
12	8	6	4	2	1	0	0
4	3	2	0	0	0	0	0
2	2	1	1	0	0	0	0
1	1	0	0	0	0	0	0
0	0	0	0	0	0	0	0

Quantized coefficients

150	80	20	4	1	0	0	0
92	75	18	3	1	0	0	0
26	19	13	2	1	0	0	0
3	2	2	1	0	0	0	0
1	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0

Quantization table

1	1	2	4	8	16	32	64
1	1	2	4	8	16	32	64
2	2	2	4	8	16	32	64
4	4	4	4	8	16	32	64
8	8	8	8	8	16	32	64
16	16	16	16	16	16	32	64
32	32	32	32	32	32	32	64
64	64	64	64	64	64	64	64

Figure 7-8. Computation of the quantized DCT coefficients.

Step 4 reduces the (0, 0) value of each block (the one in the upper left-hand corner) by replacing it with the amount it differs from the corresponding element in the previous block. Since these elements are the averages of their respective blocks, they should change slowly, so taking the differential values should reduce most of them to small values. No differentials are computed from the other values. The (0, 0) values are referred to as the DC components; the other values are the AC components.

Step 5 linearizes the 64 elements and applies run-length encoding to the list. Scanning the block from left to right and then top to bottom will not concentrate the zeros together, so a zig zag scanning pattern is used, as shown in Fig. 7-9. In this example, the zig zag pattern ultimately produces 38 consecutive 0s at the end of the matrix. This string can be reduced to a single count saying there are 38 zeros.

Now we have a list of numbers that represent the image (in transform space). Step 6 Huffman encodes the numbers for storage or transmission.

JPEG may seem complicated, but that is because it *is* complicated. Still, since it often produces a 20:1 compression or better, it is widely used. Decoding a JPEG image requires running the algorithm backward. JPEG is roughly symmetric: it takes about as long to decode an image as to encode it.

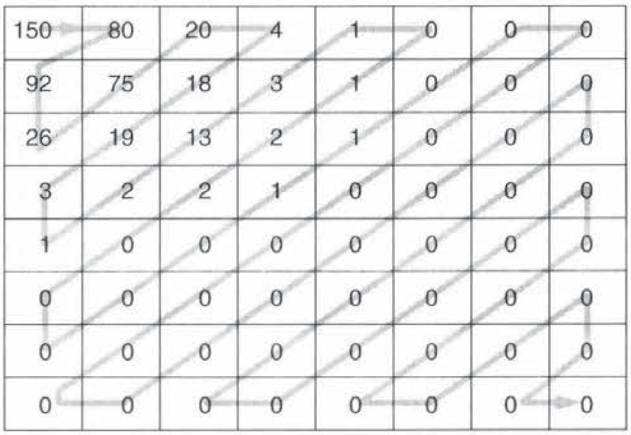


Figure 7-9. The order in which the quantized values are transmitted.

7.3.2 The MPEG Standard

Finally, we come to the heart of the matter: the **MPEG** (**Motion Picture Experts Group**) standards. These are the main algorithms used to compress videos and have been international standards since 1993. MPEG-1 (International Standard 11172) was designed for video recorder-quality output (352×240 for NTSC) using a bit rate of 1.2 Mbps. MPEG-2 (International Standard 13818) was designed for compressing broadcast quality video into 4 to 6 Mbps, so it could fit in a NTSC or PAL broadcast channel.

Both versions take advantages of the two kinds of redundancies that exist in movies: spatial and temporal. Spatial redundancy can be utilized by simply coding each frame separately with JPEG. Additional compression can be achieved by taking advantage of the fact that consecutive frames are often almost identical (temporal redundancy). The **DV** (**Digital Video**) system used by digital camcorders uses only a JPEG-like scheme because encoding has to be done in real time and it is much faster to just encode each frame separately. The consequences of this decision can be seen in Fig. 7-2: although digital camcorders have a lower data rate than uncompressed video, they are not nearly as good as full MPEG-2. (To keep the comparison honest, note that DV camcorders sample the luminance with 8 bits and each chrominance signal with 2 bits, but there is still a factor of five compression using the JPEG-like encoding.)

For scenes where the camera and background are stationary and one or two actors are moving around slowly, nearly all the pixels will be identical from frame to frame. Here, just subtracting each frame from the previous one and running JPEG on the difference would do fine. However, for scenes where the camera is panning or zooming, this technique fails badly. What is needed is some way to compensate for this motion. This is precisely what MPEG does; in fact, this is the main difference between MPEG and JPEG.

MPEG-2 output consists of three different kinds of frames that have to be processed by the viewing program:

1. I (Intracoded) frames: Self-contained JPEG-encoded still pictures.
2. P (Predictive) frames: Block-by-block difference with the last frame.
3. B (Bidirectional) frames: Differences with the last and next frame.

I-frames are just still pictures coded using JPEG, also using full-resolution luminance and half-resolution chrominance along each axis. It is necessary to have I-frames appear in the output stream periodically for three reasons. First, MPEG can be used for television broadcasting, with viewers tuning in at will. If all frames depended on their predecessors going back to the first frame, anybody who missed the first frame could never decode any subsequent frames. This would make it impossible for viewers to tune in after the movie had started. Second, if any frame were received in error, no further decoding would be possible. Third, without I-frames, while doing a fast forward or rewind, the decoder would have to calculate every frame passed over so it would know the full value of the one it stopped on. With I-frames, it is possible to skip forward or backward until an I-frame is found and start viewing there. For these reasons, I-frames are inserted into the output once or twice per second.

P-frames, in contrast, code interframe differences. They are based on the idea of **macroblocks**, which cover 16×16 pixels in luminance space and 8×8 pixels in chrominance space. A macroblock is encoded by searching the previous frame for it or something only slightly different from it.

An example of where P-frames would be useful is given in Fig. 7-10. Here we see three consecutive frames that have the same background, but differ in the position of one person. The macroblocks containing the background scene will match exactly, but the macroblocks containing the person will be offset in position by some unknown amount and will have to be tracked down.



Figure 7-10. Three consecutive video frames.

The MPEG standard does not specify how to search, how far to search, or how good a match has to be to count. This is up to each implementation. For example, an implementation might search for a macroblock at the current position in the previous frame, and all other positions offset $\pm\Delta x$ in the x direction and $\pm\Delta y$

in the y direction. For each position, the number of matches in the luminance matrix could be computed. The position with the highest score would be declared the winner, provided it was above some predefined threshold. Otherwise, the macroblock would be said to be missing. Much more sophisticated algorithms are also possible, of course.

If a macroblock is found, it is encoded by taking the difference with its value in the previous frame (for luminance and both chrominances). These difference matrices are then subject to the JPEG encoding. The value for the macroblock in the output stream is then the motion vector (how far the macroblock moved from its previous position in each direction), followed by the JPEG-encoded differences with the one in the previous frame. If the macroblock is not located in the previous frame, the current value is encoded with JPEG, just as in an I-frame.

B-frames are similar to P-frames, except that they allow the reference macroblock to be in either a previous frame or in a succeeding frame, either in an I-frame or in a P-frame. This additional freedom allows improved motion compensation, and is also useful when objects pass in front of, or behind, other objects. For example, in a baseball game, when the third baseman throws the ball to first base, there may be some frame where the ball obscures the head of the moving second baseman in the background. In the next frame, the head may be partially visible to the left of the ball, with the next approximation of the head being derived from the following frame when the ball is now past the head. B-frames allow a frame to be based on a future frame.

To do B-frame encoding, the encoder needs to hold three decoded frames in memory at once: the past one, the current one, and the future one. To simplify decoding, frames must be present in the MPEG stream in dependency order, rather than in display order. Thus even with perfect timing, when a video is viewed over a network, buffering is required on the user's machine to reorder the frames for proper display. Due to this difference between dependency order and display order, trying to play a movie backward will not work without considerable buffering and complex algorithms.

7.4 MULTIMEDIA PROCESS SCHEDULING

Operating systems that support multimedia differ from traditional ones in three main ways: process scheduling, the file system, and disk scheduling. We will start with process scheduling here and continue with the other topics in subsequent sections.

7.4.1 Scheduling Homogeneous Processes

The simplest kind of video server is one that can support the display of a fixed number of movies, all using the same frame rate, video resolution, data rate, and other parameters. Under these circumstances, a simple, but effective scheduling

algorithm is as follows. For each movie, there is a single process (or thread) whose job it is to read the movie from the disk one frame at a time and then transmit that frame to the user. Since all the processes are equally important, have the same amount of work to do per frame, and block when they have finished processing the current frame, round-robin scheduling does the job just fine. The only addition needed to standard scheduling algorithms is a timing mechanism to make sure each process runs at the correct frequency.

One way to achieve the proper timing is to have a master clock that ticks at, say, 30 times per second (for NTSC). At every tick, all the processes are run sequentially, in the same order. When a process has completed its work, it issues a suspend system call that releases the CPU until the master clock ticks again. When that happens, all the processes are run again in the same order. As long as the number of processes is small enough that all the work can be done in one frame time, round-robin scheduling is sufficient.

7.4.2 General Real-Time Scheduling

Unfortunately, this model is rarely applicable in reality. The number of users changes as viewers come and go, frame sizes vary wildly due to the nature of video compression (I-frames are much larger than P- or B-frames), and different movies may have different resolutions. As a consequence, different processes may have to run at different frequencies, with different amounts of work, and with different deadlines by which the work must be completed.

These considerations lead to a different model: multiple processes competing for the CPU, each with its own work and deadlines. In the following models, we will assume that the system knows the frequency at which each process must run, how much work it has to do, and what its next deadline is. (Disk scheduling is also an issue, but we will consider that later.) The scheduling of multiple competing processes, some or all of which have deadlines that must be met is called **real-time scheduling**.

As an example of the kind of environment a real-time multimedia scheduler works in, consider the three processes, *A*, *B*, and *C* shown in Fig. 7-11. Process *A* runs every 30 msec (approximately NTSC speed). Each frame requires 10 msec of CPU time. In the absence of competition, it would run in the bursts *A*1, *A*2, *A*3, etc., each one starting 30 msec after the previous one. Each CPU burst handles one frame and has a deadline: it must complete before the next one is to start.

Also shown in Fig. 7-11 are two other processes, *B* and *C*. Process *B* runs 25 times/sec (e.g., PAL) and process *C* runs 20 times/sec (e.g., a slowed down NTSC or PAL stream intended for a user with a low-bandwidth connection to the video server). The computation time per frame is shown as 15 msec and 5 msec for *B* and *C*, respectively, just to make the scheduling problem more general than having all of them the same.

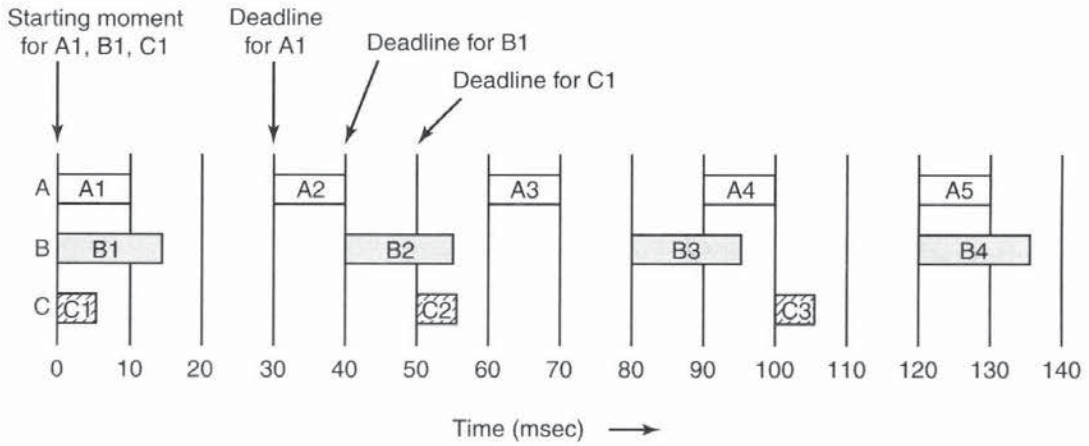


Figure 7-11. Three periodic processes, each displaying a movie. The frame rates and processing requirements per frame are different for each movie.

The scheduling question now is how to schedule A, B, and C to make sure they meet their respective deadlines. Before even looking for a scheduling algorithm, we have to see if this set of processes is schedulable at all. Recall from Sec. 2.5.4, that if process i has period P_i msec and requires C_i msec of CPU time per frame, the system is schedulable if and only if

$$\sum_{i=1}^{m} \frac{C_i}{P_i} \leq 1$$

where m is the number of processes, in this case, 3. Note that P_i/C_i is just the fraction of the CPU being used by process i. For the example of Fig. 7-11, A is eating 10/30 of the CPU, B is eating 15/40 of the CPU, and C is eating 5/50 of the CPU. Together these fractions add to 0.808 of the CPU, so the system of processes is schedulable.

So far we assumed that there is one process per stream. Actually, there might be two (or more processes) per stream, for example, one for audio and one for video. They may run at different rates and may consume differing amounts of CPU time per burst. Adding audio processes to the mix does not change the general model, however, since all we are assuming is that there are m processes, each running at a fixed frequency with a fixed amount of work needed on each CPU burst.

In some real-time systems, processes are preemptable and in others they are not. In multimedia systems, processes are generally preemptable, meaning that a process that is in danger of missing its deadline may interrupt the running processes before the running process has finished with its frame. When it is done, the previous process can continue. This behavior is just multiprogramming, as we have seen before. We will study preemptable real-time scheduling algorithms because there is no objection to them in multimedia systems and they give better

performance than nonpreemptable ones. The only concern is that if a transmission buffer is being filled in little bursts, the buffer is completely full by the deadline so it can be sent to the user in a single operation. Otherwise jitter might be introduced.

Real-time algorithms can be either static or dynamic. Static algorithms assign each process a fixed priority in advance and then do prioritized preemptive scheduling using those priorities. Dynamic algorithms do not have fixed priorities. Below we will study an example of each type.

7.4.3 Rate Monotonic Scheduling

The classic static real-time scheduling algorithm for preemptable, periodic processes is **RMS** (**Rate Monotonic Scheduling**) (Liu and Layland, 1973). It can be used for processes that meet the following conditions:

1. Each periodic process must complete within its period.
2. No process is dependent on any other process.
3. Each process needs the same amount of CPU time on each burst.
4. Any nonperiodic processes have no deadlines.
5. Process preemption occurs instantaneously and with no overhead.

The first four conditions are reasonable. The last one is not, of course, but it makes modeling the system much easier. RMS works by assigning each process a fixed priority equal to the frequency of occurrence of its triggering event. For example, a process that must run every 30 msec (33 times/sec) gets priority 33, a process that must run every 40 msec (25 times/sec) gets priority 25, and a process that must run every 50 msec (20 times/sec) gets priority 20. The priorities are thus linear with the rate (number of times/second the process runs). This is why it is called rate monotonic. At run time, the scheduler always runs the highest priority ready process, preempting the running process if need be. Liu and Layland proved that RMS is optimal among the class of static scheduling algorithms.

Figure 7-12 shows how rate monotonic scheduling works in the example of Fig. 7-11. Processes *A*, *B*, and *C* have static priorities, 33, 25, and 20, respectively, which means that whenever *A* needs to run, it runs, preempting any other process currently using the CPU. Process *B* can preempt *C*, but not *A*. Process *C* has to wait until the CPU is otherwise idle in order to run.

In Fig. 7-12, initially all three processes are ready to run. The highest priority one, *A*, is chosen, and allowed to run until it completes at 15 msec, as shown in the RMS line. After it finishes, *B* and *C* are run in that order. Together, these processes take 30 msec to run, so when *C* finishes, it is time for *A* to run again. This rotation goes on until the system goes idle at $t = 70$.

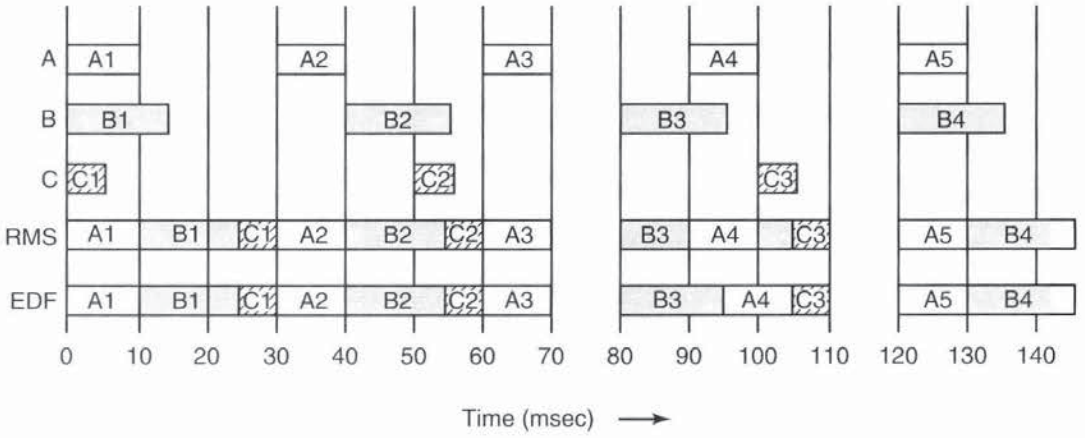


Figure 7-12. An example of RMS and EDF real-time scheduling.

At $t = 80$ B becomes ready and runs. However, at $t = 90$, a higher priority process, A, becomes ready, so it preempts B and runs until it is finished, at $t = 100$. At that point the system can choose between finishing B or starting C, so it chooses the highest priority process, B.

7.4.4 Earliest Deadline First Scheduling

Another popular real-time scheduling algorithm is **Earliest Deadline First**. EDF is a dynamic algorithm that does not require processes to be periodic, as does the rate monotonic algorithm. Nor does it require the same run time per CPU burst, as does RMS. Whenever a process needs CPU time, it announces its presence and its deadline. The scheduler keeps a list of runnable processes, sorted on deadline. The algorithm runs the first process on the list, the one with the closest deadline. Whenever a new process becomes ready, the system checks to see if its deadline occurs before that of the currently running process. If so, the new process preempts the current one.

An example of EDF is given in Fig. 7-12. Initially all three processes are ready. They are run in the order of their deadlines. A must finish by $t = 30$, B must finish by $t = 40$, and C must finish by $t = 50$, so A has the earliest deadline and thus goes first. Up until $t = 90$ the choices are the same as RMS. At $t = 90$, A becomes ready again, and its deadline is $t = 120$, the same as B's deadline. The scheduler could legitimately choose either one to run, but since in practice, preempting B has some nonzero cost associated with it, it is better to let B continue to run.

To dispel the idea that RMS and EDF always give the same results, let us now look at another example, shown in Fig. 7-13. In this example the periods of A, B, and C are the same as before, but now A needs 15 msec of CPU time per burst instead of only 10 msec. The schedulability test computes the CPU utilization as

0.500 + 0.375 + 0.100 = 0.975. Only 2.5% of the CPU is left over, but in theory the CPU is not oversubscribed and it should be possible to find a legal schedule.

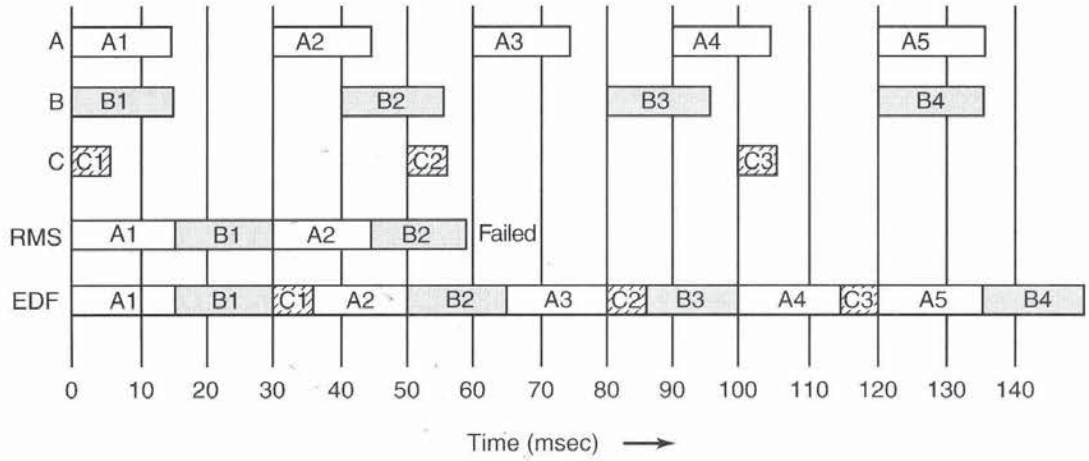


Figure 7-13. Another example of real-time scheduling with RMS and EDF.

With RMS, the priorities of the three processes are still 33, 25, and 20 as only the period matters, not the run time. This time, $B1$ does not finish until $t = 30$, at which time A is ready to roll again. By the time A is finished, at $t = 45$, B is ready again, so having a higher priority than C, it runs and C misses its deadline. RMS fails.

Now look at how EDF handles this case. At $t = 30$, there is a contest between $A2$ and $C1$. Because $C1$'s deadline is 50 and $A2$'s deadline is 60, C is scheduled. This is different from RMS, where A's higher priority wins.

At $t = 90$ A becomes ready for the fourth time. A's deadline is the same as that of the current process (120), so the scheduler has a choice of preempting or not. As before, it is better not to preempt if it is not needed, so $B3$ is allowed to complete.

In the example of Fig. 7-13, the CPU is 100% occupied up to $t = 150$. However, eventually a gap will occur because the CPU is only 97.5% utilized. Since all the starting and ending times are multiples of 5 msec, the gap will be 5 msec. In order to achieve the required 2.5% idle time, the 5 msec gap will have to occur every 200 msec, which is why it does not show up in Fig. 7-13.

An interesting question is why RMS failed. Basically, using static priorities only works if the CPU utilization is not too high. Liu and Layland (1973) proved that for any system of periodic processes, if

$$\sum_{i=1}^{m} \frac{C_i}{P_i} \leq m(2^{1/m} - 1)$$

then RMS is guaranteed to work. For 3, 4, 5, 10, 20, and 100, the maximum permitted utilizations are 0.780, 0.757, 0.743, 0.718, 0.705, and 0.696. As $m \rightarrow \infty$,

the maximum utilization is asymptotic to ln 2. In other words, Liu and Layland proved that for three processes, RMS always works if the CPU utilization is at or below 0.780. In our first example, it was 0.808 and RMS worked, but we were just lucky. With different periods and run times, a utilization of 0.808 might fail. In the second example, the CPU utilization was so high (0.975), there was no hope that RMS could work.

In contrast, EDF always works for any schedulable set of processes. It can achieve 100% CPU utilization. The price paid is a more complex algorithm. Thus in an actual video server, if the CPU utilization is below the RMS limit, RMS can be used. Otherwise EDF should be chosen.

7.5 MULTIMEDIA FILE SYSTEM PARADIGMS

Now that we have covered process scheduling in multimedia systems, let us continue our study by looking at multimedia file systems. These file systems use a different paradigm than traditional file systems. We will first review traditional file I/O, then turn our attention to how multimedia file servers are organized. To access a file, a process first issues an open system call. If this succeeds, the caller is given some kind of token, called a file descriptor in UNIX or a handle in Windows to be used in future calls. At that point the process can issue a read system call, providing the token, buffer address, and byte count as parameters. The operating system then returns the requested data in the buffer. Additional read calls can then be made until the process is finished, at which time it calls close to close the file and return its resources.

This model does not work well for multimedia on account of the need for real-time behavior. It works especially poorly for displaying multimedia files coming off a remote video server. One problem is that the user must make the read calls fairly precisely spaced in time. A second problem is that the video server must be able to supply the data blocks without delay, something that is difficult for it to do when the requests come in unplanned and no resources have been reserved in advance.

To solve these problems, a completely different paradigm is used by multimedia file servers: they act like VCRs (Video Cassette Recorders). To read a multimedia file, a user process issues a start system call, specifying the file to be read and various other parameters, for example, which audio and subtitle tracks to use. The video server then begins sending out frames at the required rate. It is up to the user to handle them at the rate they come in. If the user gets bored with the movie, the stop system call terminates the stream. File servers with this streaming model are often called **push servers** (because they push data at the user) and are contrasted with traditional **pull servers** where the user has to pull the data in one block at a time by repeatedly calling read to get one block after another. The difference between these two models is illustrated in Fig. 7-14.

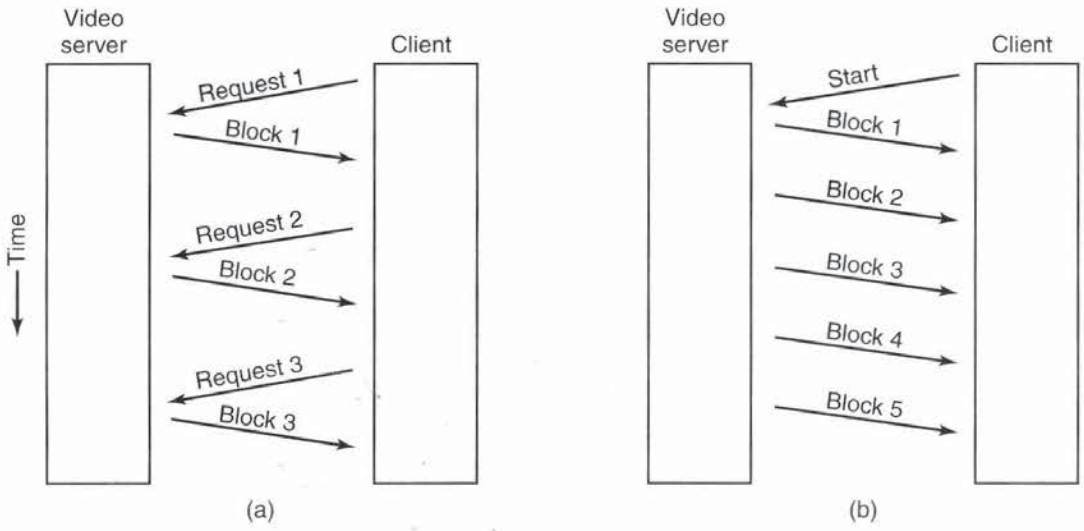


Figure 7-14. (a) A pull server. (b) A push server.

7.5.1 VCR Control Functions

Most video servers also implement standard VCR control functions, including pause, fast forward, and rewind. Pause is fairly straightforward. The user sends a message back to the video server that tells it to stop. All it has to do at that point is remember which frame goes out next. When the user tells the server to resume, it just continues from where it left off.

However, there is one complication here. To achieve acceptable performance, the server may reserve resources such as disk bandwidth and memory buffers for each outgoing stream. Continuing to tie these up while a movie is paused wastes resources, especially if the user is planning a trip to the kitchen to locate, microwave, cook, and eat a frozen pizza (especially an extra large). The resources can easily be released upon pausing, of course, but this introduces the danger that when the user tries to resume, they cannot be reacquired.

True rewind is actually easy, with no complications. All the server has to do is note that the next frame to be sent is 0. What could be easier? However, fast forward and fast backward (i.e., playing while rewinding) are much trickier. If it were not for compression, one way to go forward at 10x speed would be to just display every 10th frame. To go forward at 20x speed would require displaying every 20th frame. In fact, in the absence of compression, going forward or backward at any speed is easy. To run at k times normal speed, just display every k-th frame. To go backward at k times normal speed, do the same thing in the other direction. This approach works equally well for both pull servers and push servers.

Compression makes rapid motion either way more complicated. With a camcorder DV tape, where each frame is compressed independently of all the others,

it is possible to use this strategy, provided that the needed frame can be found quickly. Since each frame compresses by a different amount, depending on its content, each frame is a different size, so skipping ahead k frames in the file cannot be done by doing a numerical calculation. Furthermore, audio compression is done independently of video compression, so for each video frame displayed in high-speed mode, the correct audio frame must also be located (unless sound is turned off when running faster than normal). Thus fast forwarding a DV file requires an index that allows frames to be located quickly, but it is at least doable in theory.

With MPEG, this scheme does not work, even in theory, due to the use of I-, P-, and B-frames. Skipping ahead k frames (assuming that can be done at all), might land on a P-frame that is based on an I-frame that was just skipped over. Without the base frame, having the incremental changes from it (which is what a P-frame contains) is useless. MPEG requires the file to be played sequentially.

Another way to attack the problem is to actually try to play the file sequentially at 10x speed. However, doing this requires pulling data off the disk at 10x speed. At that point, the server could try to decompress the frames (something it normally does not do), figure out which frame is needed, and recompress every 10th frame as an I-frame. However, doing this puts a huge load on the server. It also requires the server to understand the compression format, something it normally does not have to know.

The alternative of actually shipping all the data over the network to the user and letting the correct frames be selected out there requires running the network at 10x speed, possibly doable, but certainly not easy given the high speed at which it normally has to operate.

All in all, there is no easy way out. The only feasible strategy requires advance planning. What can be done is build a special file containing, say, every 10th frame, and compress this file using the normal MPEG algorithm. This file is what is shown in Fig. 7-3 as "fast forward." To switch to fast forward mode, what the server must do is figure out where in the fast forward file the user currently is. For example, if the current frame is 48,210 and the fast forward file runs at 10x, the server has to locate frame 4821 in the fast forward file and start playing there at normal speed. Of course, that frame might be a P- or B-frame, but the decoding process at the client can just skip frames until it sees an I-frame. Going backward is done in an analogous way using a second specially prepared file.

When the user switches back to normal speed, the reverse trick has to be done. If the current frame in the fast forward file is 5734, the server just switches back to the regular file and continues at frame 57,340. Again, if this frame is not an I-frame, the decoding process on the client side has to ignore all frames until an I-frame is seen.

While having these two extra files does the job, the approach has some disadvantages. First, some extra disk space is required to store the additional files. Second, fast forwarding and rewinding can only be done at speeds corresponding

to the special files. Third, extra complexity is needed to switch back and forth between the regular, fast forward, and fast backward files.

7.5.2 Near Video on Demand

Having k users getting the same movie puts essentially the same load on the server as having them getting k different movies. However, with a small change in the model, great performance gains are possible. The problem with video on demand is that users can start streaming a movie at an arbitrary moment, so if there are 100 users all starting to watch some new movie at about 8 P.M., chances are that no two will start at exactly the same instant so they cannot share a stream. The change that makes optimization possible is to tell all users that movies only start on the hour and every (for example) 5 minutes thereafter. Thus if a user wants to see a movie at 8:02, he will have to wait until 8:05.

The gain here is that for a 2-hour movie, only 24 streams are needed, no matter how many customers there are. As shown in Fig. 7-15, the first stream starts at 8:00. At 8:05, when the first stream is at frame 9000, stream 2 starts. At 8:10, when the first stream is at frame 18,000 and stream 2 is at frame 9000, stream 3 starts, and so on up to stream 24, which starts at 9:55. At 10:00, stream 1 terminates and starts all over with frame 0. This scheme is called **near video on demand** because the video does not quite start on demand, but shortly thereafter.

The key parameter here is how often a stream starts. If one starts every 2 minutes, 60 streams will be needed for a two-hour movie, but the maximum waiting time to start watching will be 2 minutes. The operator has to decide how long people are willing to wait because the longer they are willing to wait, the more efficient the system, and the more movies can be shown at once. An alternative strategy is to also have a no-wait option, in which case a new stream is started on the spot, but to charge more for instant startup.

In a sense, video on demand is like using a taxi: you call it and it comes. Near video on demand is like using a bus: it has a fixed schedule and you have to wait for the next one. But mass transit only makes sense if there is a mass. In midtown Manhattan, a bus that runs every 5 minutes can count on picking up at least a few riders. A bus traveling on the back roads of Wyoming might be empty nearly all the time. Similarly, starting the latest Steven Spielberg release might attract enough customers to warrant starting a new stream every 5 minutes, but for *Gone with the Wind* it might be better to simply offer it on a demand basis.

With near video on demand, users do not have VCR controls. No user can pause a movie to make a trip to the kitchen. The best that can be done is upon returning from the kitchen, to drop back to a stream that started later, thereby repeating a few minutes of material.

Actually, there is another model for near video on demand as well. Instead of announcing in advance that some specific movie will start every 5 minutes, people can order movies whenever they want to. Every 5 minutes, the system sees which

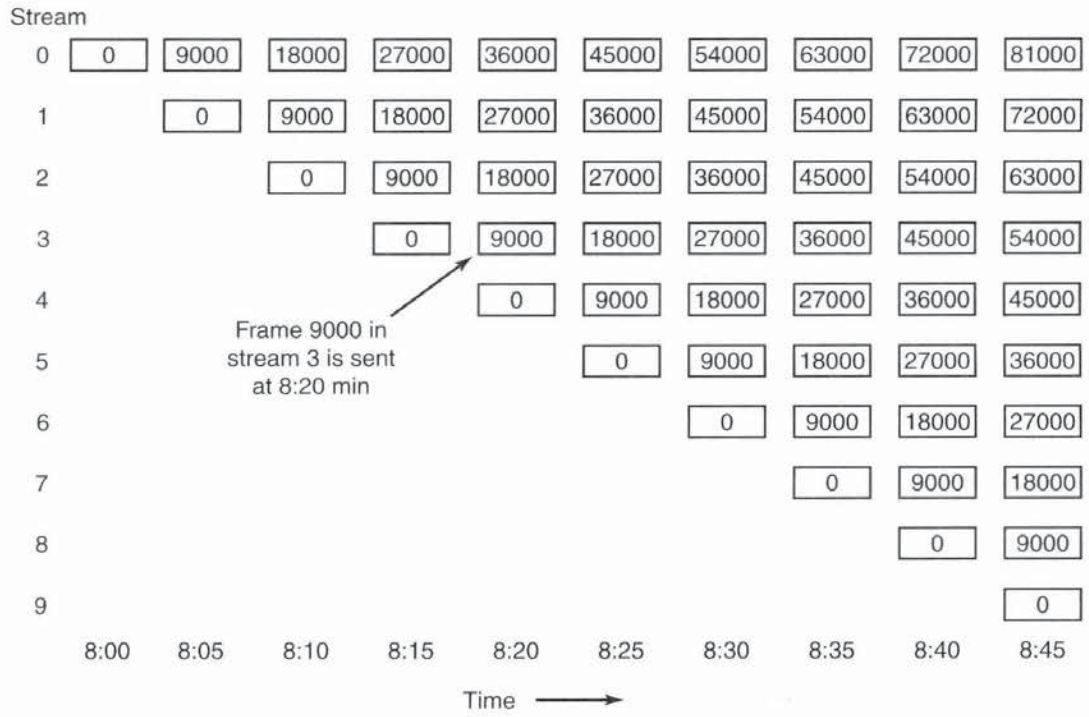


Figure 7-15. Near video on demand has a new stream starting at regular intervals, in this example every 5 minutes (9000 frames).

movies have been ordered and starts those. With this approach, a movie may start at 8:00, 8:10, 8:15, and 8:25, but not at the intermediate times, depending on demand. As a result, streams with no viewers are not transmitted, saving disk bandwidth, memory, and network capacity. On the other hand, attacking the freezer is now a bit of a gamble as there is no guarantee that there is another stream running 5 minutes behind the one the viewer was watching. Of course, the operator can provide an option for the user to display a list of all concurrent streams, but most people think their TV remote controls have more than enough buttons already and are not likely to enthusiastically welcome a few more.

7.5.3 Near Video on Demand with VCR Functions

The ideal combination would be near video on demand (for the efficiency) plus full VCR controls for every individual viewer (for the user convenience). With slight modifications to the model, such a design is possible. Below we will give a slightly simplified description of one way to achieve this goal (Abram-Profeta and Shin, 1998).

We start out with the standard near video-on-demand scheme of Fig. 7-15. However, we add the requirement that each client machine buffer the previous ΔT min and also the upcoming ΔT min locally. Buffering the previous ΔT min is

easy: just save it after displaying it. Buffering the upcoming ΔT min is harder, but can be done if clients have the ability to read two streams at once.

One way to get the buffer set up can be illustrated using an example. If a user starts viewing at 8:15, the client machine reads and displays the 8:15 stream (which is at frame 0). In parallel, it reads and stores the 8:10 stream, which is currently at the 5-min mark (i.e., frame 9000). At 8:20, frames 0 to 17,999 have been stored and the user is expecting to see frame 9000 next. From that point on, the 8:15 stream is dropped, the buffer is filled from the 8:10 stream (which is at 18,000), and the display is driven from the middle of the buffer (frame 9000). As each new frame is read, one frame is added to the end of the buffer and one frame is dropped from the beginning of the buffer. The current frame being displayed, called the **play point**, is always in the middle of the buffer. The situation 75 min into the movie is shown in Fig. 7-16(a). Here all frames between 70 min and 80 min are in the buffer. If the data rate is 4 Mbps, a 10-min buffer requires 300 million bytes of storage. With current prices, the buffer can certainly be kept on disk and possibly in RAM. If RAM is desired, but 300 million bytes is too much, a smaller buffer can be used.

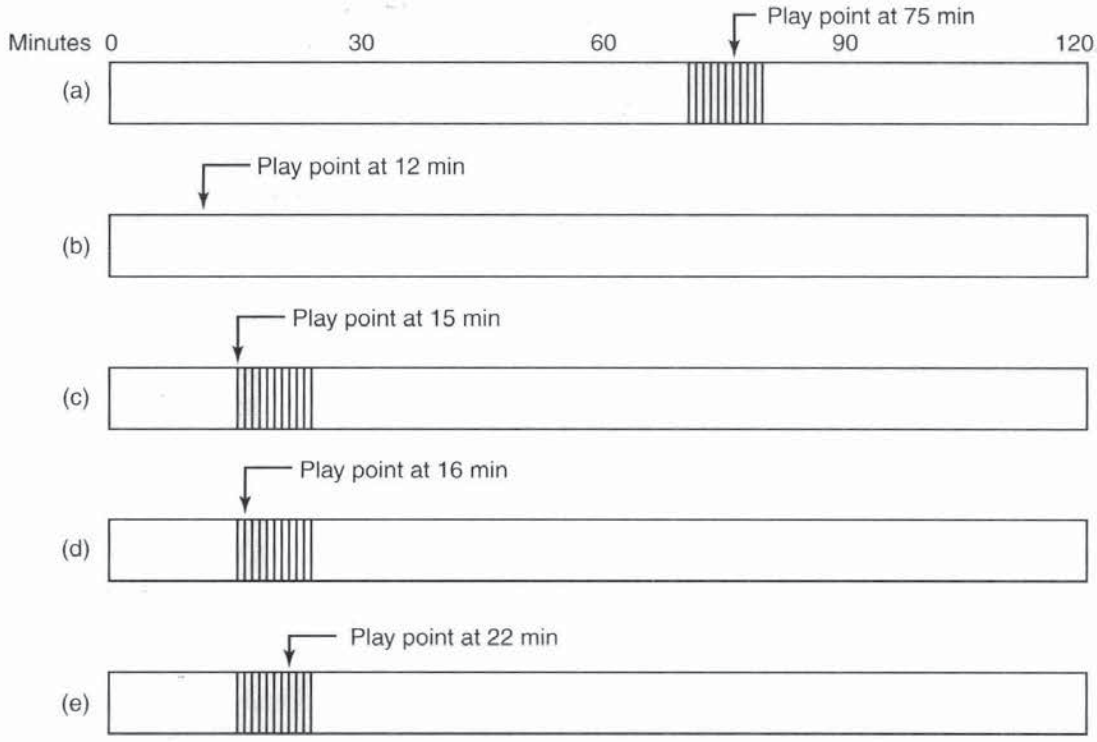


Figure 7-16. (a) Initial situation. (b) After a rewind to 12 min. (c) After waiting 3 min. (d) After starting to refill the buffer. (e) Buffer full.

Now suppose that the user decides to fast forward or fast reverse. As long as the play point stays within the range 70–80 min, the display can be fed from the

buffer. However, if the play point moves outside that interval either way, we have a problem. The solution is to turn on a private (i.e., video-on-demand) stream to service the user. Rapid motion in either direction can be handled by the techniques discussed earlier.

Normally, at some point the user will settle down and decide to watch the movie at normal speed again. At this point we can think about migrating the user over to one of the near video-on-demand streams so the private stream can be dropped. Suppose, for example, that the user decides to go back to the 12 min mark, as shown in Fig. 7-16(b). This point is far outside the buffer, so the display cannot be fed from it. Furthermore, since the switch happened (instantaneously) at 75 min, there are streams showing the movie at 5, 10, 15, and 20 min, but none at 12 min.

The solution is to continue viewing on the private stream, but to start filling the buffer from the stream currently 15 minutes into the movie. After 3 minutes, the situation is as depicted in Fig. 7-16(c). The play point is now 15 min, the buffer contains minutes 15 to 18, and the near video-on-demand streams are at 8, 13, 18, and 23 min, among others. At this point the private stream can be dropped and the display can be fed from the buffer. The buffer continues to be filled from the stream now at 18 min. After another minute, the play point is 16 min, the buffer contains minutes 15 to 19, and the stream feeding the buffer is at 19 min, as shown in Fig. 7-16(d).

After an additional 6 minutes have gone by, the buffer is full and the play point is at 22 min. The play point is not in the middle of the buffer, although that can be arranged if necessary.

7.6 FILE PLACEMENT

Multimedia files are very large, are often written only once but read many times, and tend to be accessed sequentially. Their playback must also meet strict quality of service criteria. Together, these requirements suggest different file system layouts than traditional operating systems use. We will discuss some of these issues below, first for a single disk, then for multiple disks.

7.6.1 Placing a File on a Single Disk

The most important requirement is that data can be streamed to the network or output device at the requisite speed and without jitter. For this reason, having multiple seeks during a frame is highly undesirable. One way to eliminate intrafile seeks on video servers is to use contiguous files. Normally, having files be contiguous does not work well, but on a video server that is carefully preloaded in advance with movies that do not change afterward it can work.

One complication, however, is the presence of video, audio, and text, as shown in Fig. 7-3. Even if the video, audio, and text are each stored as separate contiguous files, a seek will be needed to go from the video file to an audio file and from there to a text file, if need be. This suggests a second possible storage arrangement, with the video, audio, and text interleaved as shown in Fig. 7-17, but the entire file still contiguous. Here, the video for frame 1 is directly followed by the various audio tracks for frame 1 and then the various text tracks for frame 1. Depending on how many audio and text tracks there are, it may be simplest just to read in all the pieces for each frame in a single disk read operation and only transmit the needed parts to the user.

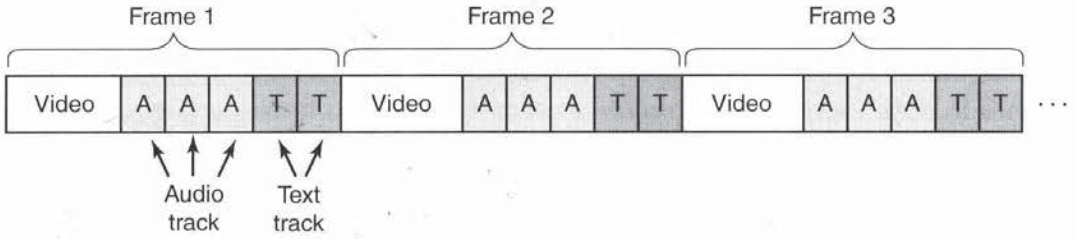


Figure 7-17. Interleaving video, audio, and text in a single contiguous file per movie.

This organization requires extra disk I/O for reading in unwanted audio and text, and extra buffer space in memory to store them. However it eliminates all seeks (on a single-user system) and does not require any overhead for keeping track of which frame is where on the disk since the whole movie is in one contiguous file. Random access is impossible with this layout, but if it is not needed, its loss is not serious. Similarly, fast forward and fast backward are impossible without additional data structures and complexity.

The advantage of having an entire movie as a single contiguous file is lost on a video server with multiple concurrent output streams because after reading a frame from one movie, the disk will have to read in frames from many other movies before coming back to the first one. Also, for a system in which movies are being written as well as being read (e.g., a system used for video production or editing), using huge contiguous files is difficult to do and not that useful.

7.6.2 Two Alternative File Organization Strategies

These observations lead to two other file placement organizations for multimedia files. The first of these, the small block model, is illustrated in Fig. 7-18(a). In this organization, the disk block size is chosen to be considerably smaller than the average frame size, even for P-frames and B-frames. For MPEG-2 at 4 Mbps with 30 frames/sec, the average frame is 16 KB, so a block size of 1 KB or 2 KB would work well. The idea here is to have a data structure, the frame index, per movie with one entry for each frame pointing to the start of

the frame. Each frame itself consists of all the video, audio, and text tracks for that frame as a contiguous run of disk blocks, as shown. In this way, reading frame k consists of indexing into the frame index to find the k-th entry, and then reading in the entire frame in one disk operation. Since different frames have different sizes, the frame size (in blocks) is needed in the frame index, but even with 1-KB disk blocks, an 8-bit field can handle a frame up to 255 KB, which is enough for an uncompressed NTSC frame, even with many audio tracks.

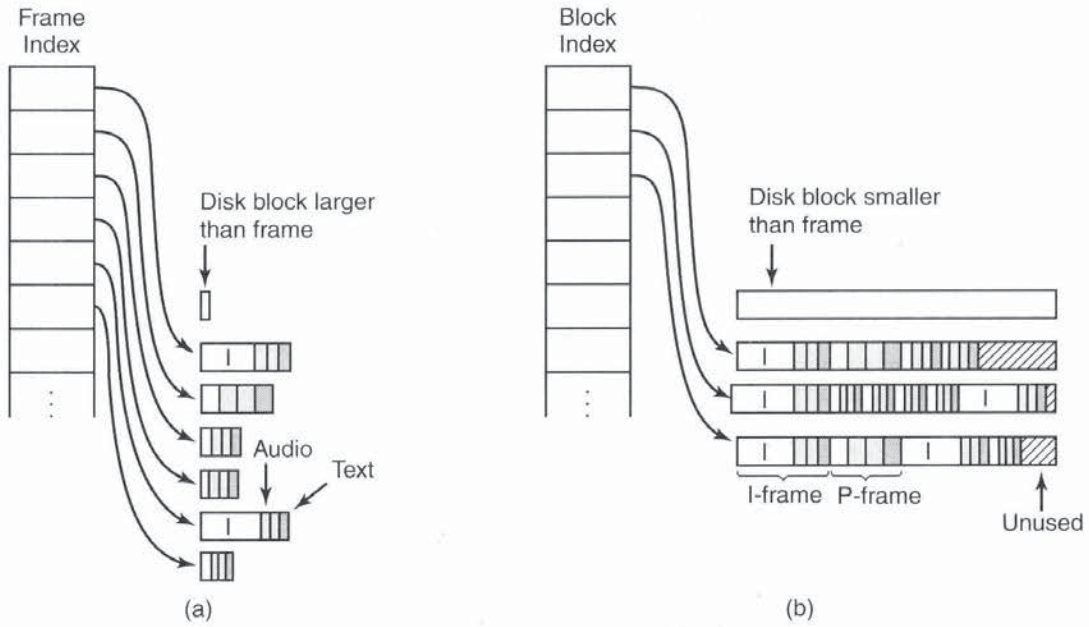


Figure 7-18. Noncontiguous movie storage. (a) Small disk blocks. (b) Large disk blocks.

The other way to store the movie is by using a large disk block (say 256 KB) and putting multiple frames in each block, as shown in Fig. 7-18(b). An index is still needed, but now it is a block index rather than a frame index. The index is, in fact, basically the same as the i-node of Fig. 6-15, possibly with the addition of information telling which frame is at the beginning of each block to make it possible to locate a given frame quickly. In general, a block will not hold an integral number of frames, so something has to be done to deal with this. Two options exist.

In the first option, which is illustrated in Fig. 7-18(b), whenever the next frame does not fit in the current block, the rest of the block is just left empty. This wasted space is internal fragmentation, the same as in virtual memory systems with fixed-size pages. On the other hand, it is never necessary to do a seek in the middle of a frame.

The other option is to fill each block to the end, splitting frames over blocks. This option introduces the need for seeks in the middle of frames, which can hurt performance, but saves disk space by eliminating internal fragmentation.

For comparison purposes, the use of small blocks in Fig. 7-18(a) also wastes some disk space because a fraction of the last block in each frame is unused. With a 1-KB disk block and a 2-hour NTSC movie consisting of 216,000 frames, the wasted disk space will only be about 108 KB out of 3.6 GB. The wasted space is harder to calculate for Fig. 7-18(b), but it will have to be much more because from time to time there will be 100 KB left at the end of a block with the next frame being an I-frame larger than that.

On the other hand, the block index is much smaller than the frame index. With a 256-KB block and an average frame of 16 KB, about 16 frames fit in a block, so a 216,000-frame movie needs only 13,500 entries in the block index, versus 216,000 for the frame index. For performance reasons, in both cases the index should list all the frames or blocks (i.e., no indirect blocks as UNIX), so tying up 13,500 8-byte entries in memory (4 bytes for the disk address, 1 byte for the frame size, and 3 bytes for the number of the starting frame) versus 216,000 5-byte entries (disk address and size only) saves almost 1 MB of RAM while the movie is playing.

These considerations lead to the following trade-offs:

1. Frame index: Heavier RAM usage while movie is playing; little disk wastage.
2. Block index (no splitting frames over blocks): Low RAM usage; major disk wastage.
3. Block index (splitting frames over blocks is allowed): Low RAM usage; no disk wastage; extra seeks

Thus the trade-offs involve RAM usage during playback, wasted disk space all the time, and performance loss during playback due to extra seeks. These problems can be attacked in various ways though. RAM usage can be reduced by paging in parts of the frame table just in time. Seeks during frame transmission can be masked by sufficient buffering, but this introduces the need for extra memory and probably extra copying. A good design has to carefully analyze all these factors and make a good choice for the application at hand.

Yet another factor here is that disk storage management is more complicated in Fig. 7-18(a) because storing a frame requires finding a consecutive run of blocks the right size. Ideally, this run of blocks should not cross a disk track boundary, but with head skew, the loss is not serious. Crossing a cylinder boundary should be avoided, however. These requirements mean that the disk's free storage has to be organized as a list of variable-sized holes, rather than a simple block list or bitmap, both of which can be used in Fig. 7-18(b).

In all cases, there is much to be said for putting all the blocks or frames of a movie within a narrow range, say a few cylinders, where possible. Such a placement means that seeks go faster so that more time will be left over for other (nonreal-time) activities or for supporting additional video streams. A constrained

placement of this sort can be achieved by dividing the disk into cylinder groups and for each group keeping separate lists or bitmaps of the free blocks. If holes are used, for example, there could be one list for 1-KB holes, one for 2-KB holes, one for holes of 3 KB to 4 KB, another for holes of size 5 KB to 8 KB, and so on. In this way it is easy to find a hole of a given size in a given cylinder group.

Another difference between these two approaches is buffering. With the small-block approach, each read gets exactly one frame. Consequently, a simple double buffering strategy works fine: one buffer for playing back the current frame and one for fetching the next one. If fixed buffers are used, each buffer has to be large enough for the biggest possible I-frame. On the other hand, if a different buffer is allocated from a pool on every frame, and the frame size is known before the frame is read in, a small buffer can be chosen for a P-frame or B-frame.

With large blocks, a more complex strategy is required because each block contains multiple frames, possibly including fragments of frames on each end of the block (depending on which option was chosen earlier). If displaying or transmitting frames requires them to be contiguous, they must be copied, but copying is an expensive operation so it should be avoided where possible. If contiguity is not required, then frames that span block boundaries can be sent out over the network or to the display device in two chunks.

Double buffering can also be used with large blocks, but using two large blocks wastes memory. One way around wasting memory is to have a circular transmission buffer slightly larger than a disk block (per stream) that feeds the network or display. When the buffer's contents drop below some threshold, a new large block is read in from the disk, the contents copied to the transmission buffer, and the large block buffer returned to a common pool. The circular buffer's size must be chosen so that when it hits the threshold, there is room for another full disk block. The disk read cannot go directly to the transmission buffer because it might have to wrap around. Here copying and memory usage are being traded off against one another.

Yet another factor in comparing these two approaches is disk performance. Using large blocks runs the disk at full speed, often a major concern. Reading in little P-frames and B-frames as separate units is not efficient. In addition, striping large blocks over multiple drives (discussed below) is possible, whereas striping individual frames over multiple drives is not.

The small-block organization of Fig. 7-18(a) is sometimes called **constant time length** because each pointer in the index represents the same number of milliseconds of playing time. In contrast, the organization of Fig. 7-18(b) is sometimes called **constant data length** because the data blocks are the same size.

Another difference between the two file organizations is that if the frame types are stored in the index of Fig. 7-18(a), it may be possible to perform a fast forward by just displaying the I-frames. However, depending on how often I-frames appear in the stream, the rate may be perceived as too fast or too slow. In any case, with the organization of Fig. 7-18(b) fast forwarding is not possible this

way. Actually reading the file sequentially to pick out the desired frames requires massive disk I/O.

A second approach is to use a special file that when played at normal speed gives the illusion of fast forwarding at 10x speed. This file can be structured the same as other files, using either a frame index or a block index. When opening a file, the system has to be able to find the fast forward file if needed. If the user hits the fast forward button, the system must instantly find and open the fast forward file and then jump to the correct place in the file. What it knows is the frame number it is currently at, but it needs the ability to locate the corresponding frame in the fast forward file. If it is currently at frame, say, 4816, and it knows the fast forward file is at 10x, then it must locate frame 482 in that file and start playing from there.

If a frame index is used, locating a specific frame is easy: just index into the frame index. If a block index is used, extra information in each entry is needed to identify which frame is in which block and a binary search of the block index has to be performed. Fast backward works in an analogous way to fast forward.

7.6.3 Placing Files for Near Video on Demand

So far we have looked at placement strategies for video on demand. For near video on demand, a different file placement strategy is more efficient. Remember that the same movie is going out as multiple staggered streams. Even if the movie is stored as a contiguous file, a seek is needed for each stream. Chen and Thapar (1997) have devised a file placement strategy to eliminate nearly all of those seeks. Its use is illustrated in Fig. 7-19 for a movie running at 30 frames/sec with a new stream starting every 5 min, as in Fig. 7-15. With these parameters, 24 concurrent streams are needed for a 2-hour movie.

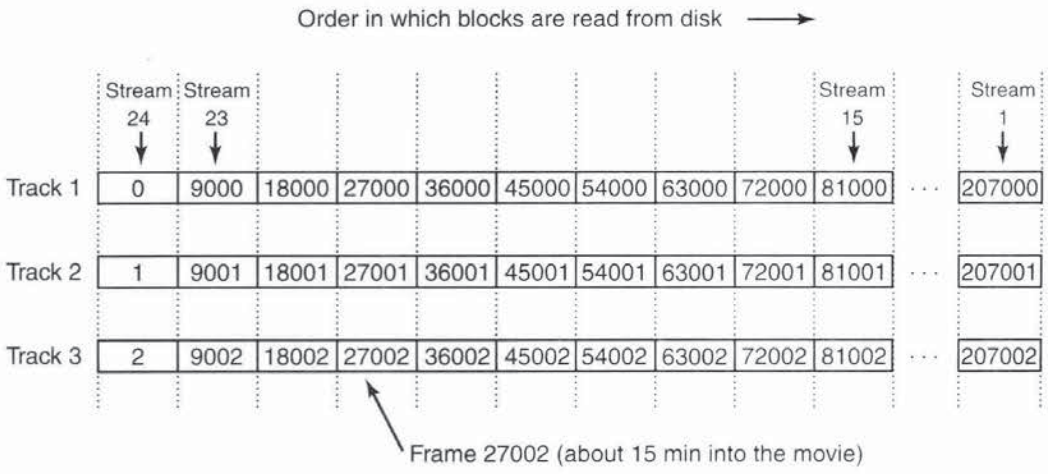


Figure 7-19. Optimal frame placement for near video on demand.

In this placement, frame sets of 24 frames are concatenated and written to the disk as a single record. They can also be read back on a single read. Consider the instant that stream 24 is just starting. It will need frame 0. Frame 23, which started 5 min earlier, will need frame 9000. Stream 22 will need frame 18,000, and so on back to stream 0 which will need frame 207,000. By putting these frames consecutively on one disk track, the video server can satisfy all 24 streams in reverse order with only one seek (to frame 0). Of course, the frames can be reversed on the disk if there is some reason to service the streams in ascending order. After the last stream has been serviced, the disk arm can move to track 2 to prepare servicing them all again. This scheme does not require the entire file to be contiguous, but still affords good performance to a number of streams at once.

A simple buffering strategy is to use double buffering. While one buffer is being played out onto 24 streams, another buffer is being loaded in advance. When the current one finishes, the two buffers are swapped and the one just used for playback is now loaded in a single disk operation.

An interesting question is how large to make the buffer. Clearly, it has to hold 24 frames. However, since frames are variable in size, it is not entirely trivial to pick the right size buffer. Making the buffer large enough for 24 I-frames is overkill, but making it large enough for 24 average frames is living dangerously.

Fortunately, for any given movie, the largest track (in the sense of Fig. 7-19) in the movie is known in advance, so a buffer of precisely that size can be chosen. However, it might just happen that in the biggest track, there are, say, 16 I-frames, whereas the next biggest track has only nine I-frames. A decision to choose a buffer large enough for the second biggest case might be wiser. Making this choice means truncating the biggest track, thus denying some streams one frame in the movie. To avoid a glitch, the previous frame can be redisplayed. No one will notice this.

Taking this approach further, if the third biggest track has only four I-frames, using a buffer capable of holding four I-frames and 20 P-frames is worth it. Introducing two repeated frames for some streams twice in the movie is probably acceptable. Where does this end? Probably with a buffer size that is big enough for 99% of the frames. Clearly, there is a trade-off here between memory used for buffers and quality of the movies shown. Note that the more simultaneous streams there are, the better the statistics are and the more uniform the frame sets will be.

7.6.4 Placing Multiple Files on a Single Disk

So far we have looked only at the placement of a single movie. On a video server, there will be many movies, of course. If they are strewn randomly around the disk, time will be wasted moving the disk head from movie to movie when multiple movies are being viewed simultaneously by different customers.

This situation can be improved by observing that some movies are more popular than others and taking popularity into account when placing movies on the disk. Although little can be said about the popularity of particular movies in general (other than noting that having big-name stars seems to help), something can be said about the relative popularity of movies in general.

For many kinds of popularity contests, such as movies being rented, books being checked out of a library, Web pages being referenced, even English words being used in a novel or the population of the largest cities, a reasonable approximation of the relative popularity follows a surprisingly predictable pattern. This pattern was discovered by a Harvard professor of linguistics, George Zipf (1902–1950) and is now called **Zipf's law**. What it states is that if the movies, books, Web pages, or words are ranked on their popularity, the probability that the next customer will choose the item ranked k-th in the list is C/k, where C is a normalization constant.

Thus the fraction of hits for the top three movies are $C/1$, $C/2$, and $C/3$, respectively, where C is computed such that the sum of all the terms is 1. In other words, if there are N movies, then

$$C/1 + C/2 + C/3 + C/4 + \cdots + C/N = 1$$

From this equation, C can be calculated. The values of C for populations with 10, 100, 1000, and 10,000 items are 0.341, 0.193, 0.134, and 0.102, respectively. For example, for 1000 movies, the probabilities for the top five movies are 0.134, 0.067, 0.045, 0.034, and 0.027, respectively.

Zipf's law is illustrated in Fig. 7-20. Just for fun, it has been applied to the populations of the 20 largest U.S. cities. Zipf's law predicts that the second largest city should have a population half of the largest city and the third largest city should be one third of the largest city, and so on. While hardly perfect, it is a surprisingly good fit.

For movies on a video server, Zipf's law states that the most popular movie is chosen twice as often as the second most popular movie, three times as often as the third most popular movie, and so on. Despite the fact that the distribution falls off fairly quickly at the beginning, it has a long tail. For example, movie 50 has a popularity of $C/50$ and movie 51 has a popularity of $C/51$, so movie 51 is 50/51 as popular as movie 50, only about a 2% difference. As one goes out further on the tail, the percent difference between consecutive movies becomes less and less. One conclusion is that the server needs a lot of movies since there is substantial demand for movies outside the top 10.

Knowing the relative popularities of the different movies makes it possible to model the performance of a video server and to use that information for placing files. Studies have shown that the best strategy is surprisingly simple and distribution independent. It is called the **organ-pipe algorithm** (Grossman and Silverman, 1973; and Wong, 1983). It consists of placing the most popular movie in the middle of the disk, with the second and third most popular movies on either side

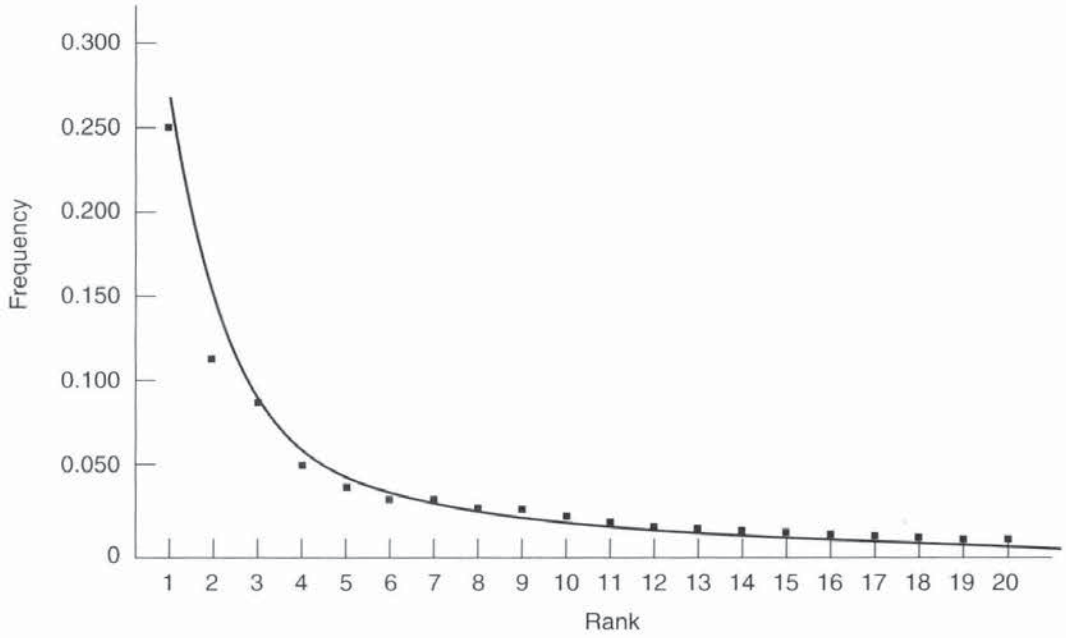


Figure 7-20. The curve gives Zipf's law for $N = 20$. The squares represent the populations of the 20 largest cities in the U.S., sorted on rank order (New York is 1, Los Angeles is 2, Chicago is 3, etc.).

of it. Outside of these come numbers four and five, and so on, as shown in Fig. 7-21. This placement works best if each movie is a contiguous file of the type shown in Fig. 7-17, but can also be used to some extent if each movie is constrained to a narrow range of cylinders. The name of the algorithm comes from the fact that a histogram of the probabilities looks like a slightly lopsided organ.

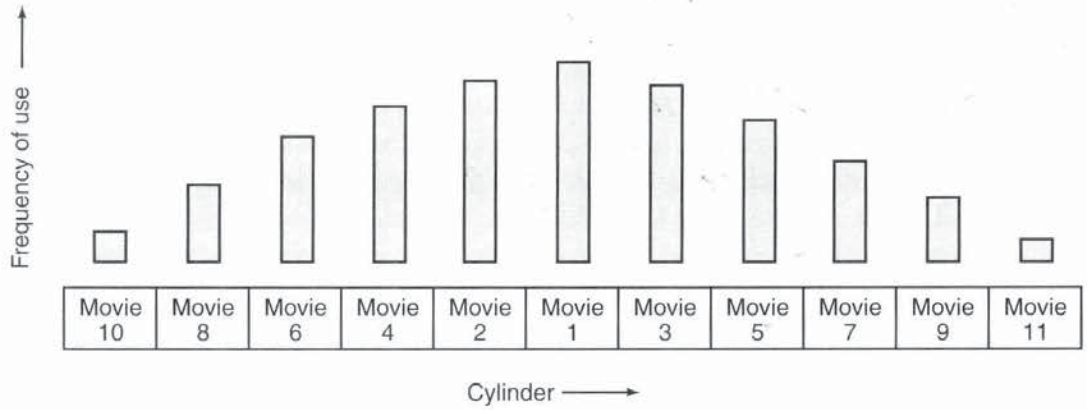


Figure 7-21. The organ-pipe distribution of files on a video server

What this algorithm does is try to keep the disk head in the middle of the disk. With 1000 movies and a Zipf's law distribution, the top five movies represent a

total probability of 0.307, which means that the disk head will stay in the cylinders allocated to the top five movies about 30% of the time, a surprisingly large amount if 1000 movies are available.

7.6.5 Placing Files on Multiple Disks

To get higher performance, video servers often have many disks that can run in parallel. Sometimes RAIDs are used, but often not because what RAIDs offer is higher reliability at the cost of performance. Video servers generally want high performance and do not care so much about correcting transient errors. Also RAID controllers can become a bottleneck if they have too many disks to handle at once.

A more common configuration is simply a large number of disks, sometimes referred to as a **disk farm**. The disks do not rotate in a synchronized way and do not contain any parity bits, as RAIDS do. One possible configuration is to put movie *A* on disk 1, movie *B* on disk 2, and so on, as shown in Fig. 7-22(a). In practice, with modern disks several movies can be placed on each disk.

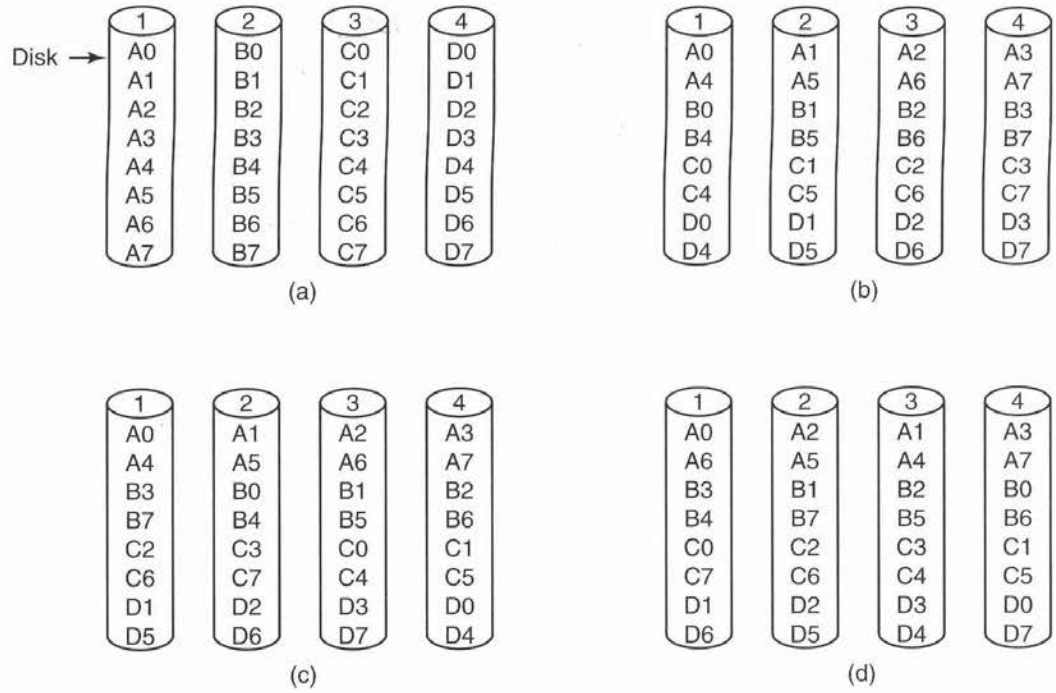


Figure 7-22. Four ways of organizing multimedia files over multiple disks. (a) No striping. (b) Same striping pattern for all files. (c) Staggered striping. (d) Random striping.

This organization is simple to implement and has straightforward failure characteristics: if one disk fails, all the movies on it become unavailable. Note that a company losing a disk full of movies is not nearly as bad as a company losing a

disk full of data because the movies can easily be reloaded on a spare disk from a DVD. A disadvantage of this approach is that the load may not be well balanced. If some disks hold movies that are currently much in demand and other disks hold less popular movies, the system will not be fully utilized. Of course, once the usage frequencies of the movies are known, it may be possible to move some of them to balance the load by hand.

A second possible organization is to stripe each movie over multiple disks, four in the example of Fig. 7-22(b). Let us assume for the moment that all frames are the same size (i.e., uncompressed). A fixed number of bytes from movie *A* is written to disk 1, then the same number of bytes is written to disk 2, and so on until the last disk is reached (in this case with unit *A*3). Then the striping continues at the first disk again with *A*4 and so on until the entire file has been written. At that point movies *B*, *C*, and *D* are striped using the same pattern.

A possible disadvantage of this striping pattern is that because all movies start on the first disk, the load across the disks may not be balanced. One way to spread the load better is to stagger the starting disks, as shown in Fig. 7-22(c). Yet another way to attempt to balance the load is to use a random striping pattern for each file, as shown in Fig. 7-22(d).

So far we have assumed that all frames are the same size. With MPEG-2 movies, this assumption is false: I-frames are much larger than P-frames. There are two ways of dealing with this complication: stripe by frame or stripe by block. When striping by frame, the first frame of movie *A* goes on disk 1 as a contiguous unit, independent of how big it is. The next frame goes on disk 2, and so on. Movie *B* is striped in a similar way, either starting at the same disk, the next disk (if staggered), or a random disk. Since frames are read one at a time, this form of striping does not speed up the reading of any given movie. However, it spreads the load over the disks much better than in Fig. 7-22(a), which may behave badly if many people decide to watch movie *A* tonight and nobody wants movie *C*. On the whole, spreading the load over all the disks makes better use of the total disk bandwidth, and thus increases the number of customers that can be served.

The other way of striping is by block. For each movie, fixed-size units are written on each of the disks in succession (or at random). Each block contains one or more frames or fragments thereof. The system can now issue requests for multiple blocks at once for the same movie. Each request asks to read data into a different memory buffer, but in such a way that when all requests have been completed, a contiguous chunk of the movie (containing many frames) is now assembled in memory contiguously. These requests can proceed in parallel. When the last request has been satisfied, the requesting process can be signaled that the work has been completed. It can then begin transmitting the data to the user. A number of frames later, when the buffer is down to the last few frames, more requests are issued to preload another buffer. This approach uses large amounts of memory for buffering in order to keep the disks busy. On a system with 1000 active users and 1-MB buffers (for example, using 256-KB blocks on each of four

disks), 1 GB of RAM is needed for the buffers. Such an amount is small potatoes on a 1000-user server and should not be a problem.

One final issue concerning striping is how many disks to stripe over. At one extreme, each movie is striped over all the disks. For example, with 2-GB movies and 1000 disks, a block of 2 MB could be written on each disk so that no movie uses the same disk twice. At the other extreme, the disks are partitioned into small groups (as in Fig. 7-22) and each movie is restricted to a single partition. The former, called **wide striping**, does a good job of balancing the load over the disks. Its main problem is that if every movie uses every disk and one disk goes down, no movie can be shown. The latter, called **narrow striping**, may suffer from hot spots (popular partitions), but loss of one disk only ruins the movies in its partition. Striping of variable-sized frames is analyzed in detail mathematically in (Shenoy and Vin, 1999).

7.7 CACHING

Traditional LRU file caching does not work well with multimedia files because the access patterns for movies are different from those of text files. The idea behind traditional LRU buffer caches is that after a block is used, it should be kept in the cache in case it is needed again quickly. For example, when editing a file, the set of blocks on which the file is written tend to be used over and over until the edit session is finished. In other words, when there is relatively high probability that a block will be reused within a short interval, it is worth keeping around to eliminate a future disk access.

With multimedia, the usual access pattern is that a movie is viewed from beginning to end sequentially. A block is unlikely to be used a second time unless the user rewinds the movie to see some scene again. Consequently, normal caching techniques do not work. However, caching can still help, but only if used differently. In the following sections we will look at caching for multimedia.

7.7.1 Block Caching

Although just keeping a block around in the hope that it may be reused quickly is pointless, the predictability of multimedia systems can be exploited to make caching useful again. Suppose that two users are watching the same movie, with one of them having started 2 sec after the other. After the first user has fetched and viewed any given block, it is very likely that the second user will need the same block 2 sec later. The system can easily keep track of which movies have only one viewer and which have two or more viewers spaced closely together in time.

Thus whenever a block is read on behalf of a movie that will be needed again shortly, it may make sense to cache it, depending on how long it has to be cached

and how tight memory is. Instead of keeping all disk blocks in the cache and discarding the least recently used one when the cache fills up, a different strategy should be used. Every movie that has a second viewer within some time ΔT of the first viewer can be marked as cachable and all its blocks cached until the second (and possibly third) viewer has used them. For other movies, no caching is done at all.

This idea can be taken a step further. In some cases it may be feasible to merge two streams. Suppose that two users are watching the same movie but with a 10-sec delay between them. Holding the blocks in the cache for 10 sec is possible but wastes memory. An alternative, but slightly sneaky, approach is to try to get the two movies in sync. This can be done by changing the frame rate for both movies. This idea is illustrated in Fig. 7-23.

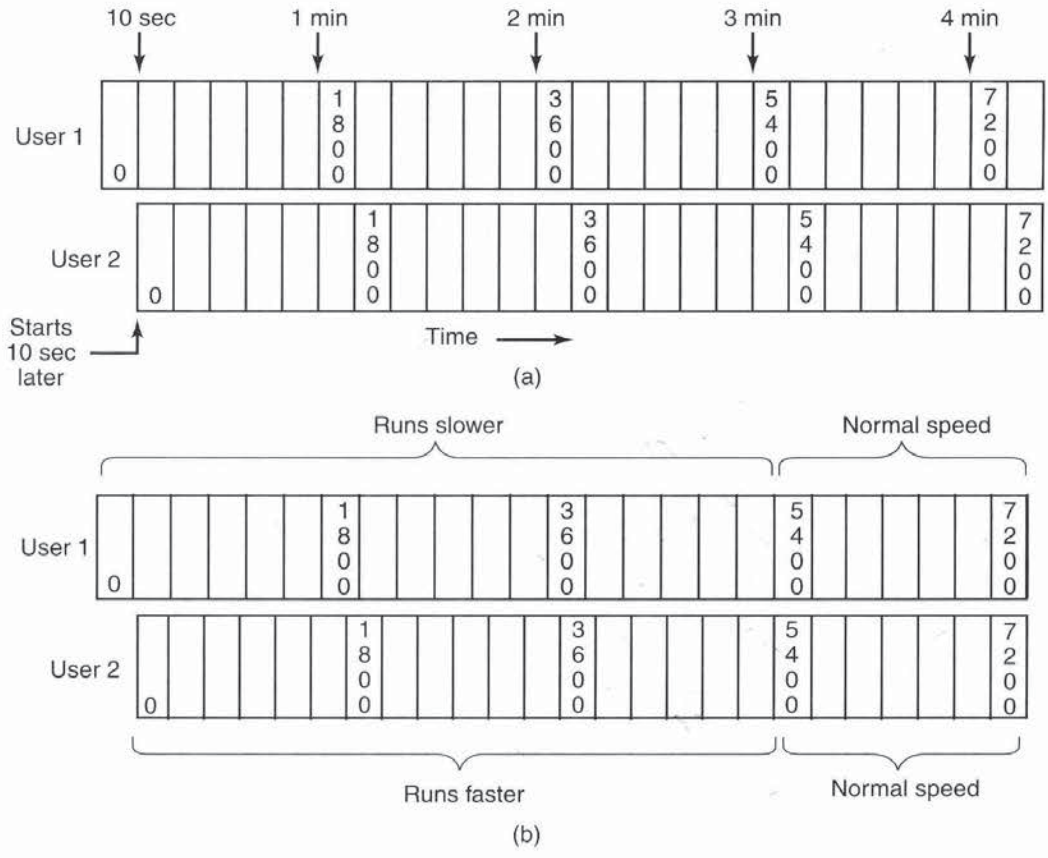


Figure 7-23. (a) Two users watching the same movie 10 sec out of sync. (b) Merging the two streams into one.

In Fig. 7-23(a), both movies run at the standard NTSC rate of 1800 frames/min. Since user 2 started 10 sec later, he continues to be 10 sec beyond for the entire movie. In Fig. 7-23(b), however, user 1's stream is slowed down when user 2 shows up. Instead of running 1800 frames/min, for the next 3 min, it

runs at 1750 frames/min. After 3 minutes, it is at frame 5550. In addition, user 2's stream is played at 1850 frames/min for the first 3 min, also putting it at frame 5550. From that point on, both play at normal speed.

During the catch-up period, user 1's stream is running 2.8% slow and user 2's stream is running 2.8% fast. It is unlikely that the users will notice this. However, if that is a concern, the catch-up period can be spread out over a longer interval than 3 minutes.

An alternative way to slow down a user to merge with another stream is to give users the option of having commercials in their movies, presumably for a lower viewing price than commercial-free movies. The user can also choose the product categories, so the commercials will be less intrusive and more likely to be watched. By manipulating the number, length, and timing of the commercials, the stream can be held back long enough to get in sync with the desired stream (Krishnan, 1999).

7.7.2 File Caching

Caching can also be useful in multimedia systems in a different way. Due to the large size of most movies (2 GB), video servers often cannot store all their movies on disk, so they keep them on DVD or tape. When a movie is needed, it can always be copied to disk, but there is a substantial startup time to locate the movie and copy it to disk. Consequently, most video servers maintain a disk cache of the most heavily requested movies. The popular movies are stored in their entirety on disk.

Another way to use caching is to keep the first few minutes of each movie on disk. That way, when a movie is requested, playback can start immediately from the disk file. Meanwhile, the movie is copied from DVD or tape to disk. By storing enough of the movie on disk all the time, it is possible to have a very high probability that the next piece of the movie has been fetched before it is needed. If all goes well, the entire movie will be on disk well before it is needed. It will then go in the cache and stay on disk in case there are more requests later. If too much time goes by without another request, the movie will be removed from the cache to make room for a more popular one.

7.8 DISK SCHEDULING FOR MULTIMEDIA

Multimedia puts different demands on the disks than traditional text-oriented applications such as compilers or word processors. In particular, multimedia demands an extremely high data rate and real-time delivery of the data. Neither of these is trivial to provide. Furthermore, in the case of a video server, there is economic pressure to have a single server handle thousands of clients simultaneously. These requirements impact the entire system. Above we looked at the file system. Now let us look at disk scheduling for multimedia.

7.8.1 Static Disk Scheduling

Although multimedia puts enormous real-time and data-rate demands on all parts of the system, it also has one property that makes it easier to handle than a traditional system: predictability. In a traditional operating system, requests are made for disk blocks in a fairly unpredictable way. The best the disk subsystem can do is perform a one-block read ahead for each open file. Other than that, all it can do is wait for requests to come in and process them on demand. Multimedia is different. Each active stream puts a well-defined load on the system that is highly predictable. For NTSC playback, every 33.3 msec, each client wants the next frame in its file and the system has 33.3 msec to provide all the frames (the system needs to buffer at least one frame per stream so that the fetching of frame $k + 1$ can proceed in parallel with the playback of frame k).

This predictable load can be used to schedule the disk using algorithms tailored to multimedia operation. Below we will consider just one disk, but the idea can be applied to multiple disks as well. For this example we will assume that there are 10 users, each one viewing a different movie. Furthermore, we will assume that all movies have the same resolution, frame rate, and other properties.

Depending on the rest of the system, the computer may have 10 processes, one per video stream, or one process with 10 threads, or even one process with one thread that handles the 10 streams in round-robin fashion. The details are not important. What is important, is that time is divided up into **rounds**, where a round is the frame time (33.3 msec for NTSC, 40 msec for PAL). At the start of each round, one disk request is generated on behalf of each user, as shown in Fig. 7-24.

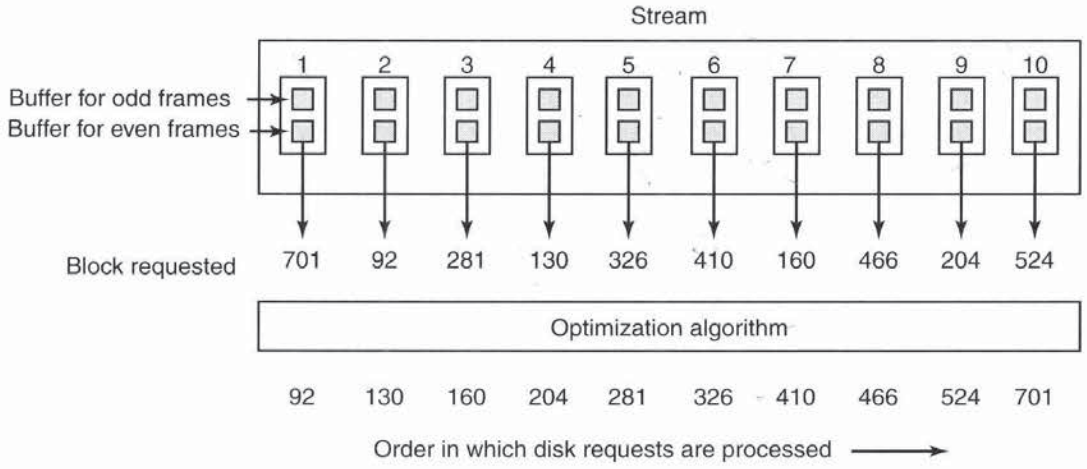


Figure 7-24. In one round, each movie asks for one frame.

After all the requests have come in at the start of the round, the disk knows what it has to do during that round. It also knows that no other requests will come in until these have been processed and the next round has begun. Consequently, it

can sort the requests in the optimal way, probably in cylinder order (although conceivably in sector order in some cases) and then process them in the optimal order. In Fig. 7-24, the requests are shown sorted in cylinder order.

At first glance, one might think that optimizing the disk in this way has no value because as long as the disk meets the deadline, it does not matter if it meets it with 1 msec to spare or 10 msec to spare. However, this conclusion is false. By optimizing seeks in this fashion, the average time to process each request is diminished, which means that the disk can handle more streams per round on the average. In other words, optimizing disk requests like this increases the number of movies the server can transmit simultaneously. Spare time at the end of the round can also be used to service any nonreal-time requests that may exist.

If a server has too many streams, once in a while when it is asked to fetch frames from distant parts of the disk and miss a deadline. But as long as missed deadlines are rare enough, they can be tolerated in return for handling more streams at once. Note that what matters is the number of streams being fetched. Having two or more clients per stream does not affect disk performance or scheduling.

To keep the flow of data out to the clients moving smoothly, double buffering is needed in the server. During round 1, one set of buffers is used, one buffer per stream. When the round is finished, the output process or processes are unblocked and told to transmit frame 1. At the same time, new requests come in for frame 2 of each movie (there might be a disk thread and an output thread for each movie). These requests must be satisfied using a second set of buffers, as the first ones are still busy. When round 3 starts, the first set of buffers are now free and can be reused to fetch frame 3.

We have assumed that there is one round per frame. This limitation is not strictly necessary. There could be two rounds per frame to reduce the amount of buffer space required, at the cost of twice as many disk operations. Similarly, two frames could be fetched from the disk per round (assuming pairs of frames are stored contiguously on the disk). This design cuts the number of disk operations in half, at the cost of doubling the amount of buffer space required. Depending on the relative availability, performance, and cost of memory versus disk I/O, the optimum strategy can be calculated and used.

7.8.2 Dynamic Disk Scheduling

In the example above, we made the assumption that all streams have the same resolution, frame rate, and other properties. Now let us drop this assumption. Different movies may now have different data rates, so it is not possible to have one round every 33.3 msec and fetch one frame for each stream. Requests come in to the disk more or less at random.

Each read request specifies which block is to be read and in addition at what time the block is needed, that is, the deadline. For simplicity, we will assume that

the actual service time for each request is the same (even though this is certainly not true). In this way we can subtract the fixed service time from each request to get the latest time the request can be initiated and still meet the deadline. This makes the model simpler because what the disk scheduler cares about is the deadline for scheduling the request.

When the system starts up, there are no disk requests pending. When the first request comes in, it is serviced immediately. While the first seek is taking place, other requests may come in, so when the first request is finished, the disk driver may have a choice of which request to process next. Some request is chosen and started. When that request is finished, there is again a set of possible requests: those that were not chosen the first time and the new arrivals that came in while the second request was being processed. In general, whenever a disk request completes, the driver has some set of requests pending from which it has to make a choice. The question is: "What algorithm does it use to select the next request to service?"

Two factors play a role in selecting the next disk request: deadlines and cylinders. From a performance point of view, keeping the requests sorted on cylinder and using the elevator algorithm minimizes total seek time, but may cause requests on outlying cylinders to miss their deadline. From a real-time point of view, sorting the requests on deadline and processing them in deadline order, earliest deadline first, minimizes the chance of missing deadlines, but increases total seek time.

These factors can be combined using the **scan-EDF algorithm** (Reddy and Wyllie, 1992). The basic idea of this algorithm is to collect requests whose deadlines are relatively close together into batches and process these in cylinder order. As an example, consider the situation of Fig. 7-25 at $t = 700$. The disk driver knows it has 11 requests pending for various deadlines and various cylinders. It could decide, for example, to treat the five requests with the earliest deadlines as a batch, sort them on cylinder number, and use the elevator algorithm to service these in cylinder order. The order would then be 110, 330, 440, 676, and 680. As long as every request is completed before its deadline, the requests can be safely rearranged to minimize the total seek time required.

When different streams have different data rates, a serious issue arises when a new customer shows up: should the customer be admitted. If admission of the customer will cause other streams to miss their deadlines frequently, the answer is probably no. There are two ways to calculate whether to admit the new customer or not. One way is to assume that each customer needs a certain amount of resources on the average, for example, disk bandwidth, memory buffers, CPU time, etc. If there is enough of each left for an average customer, the new one is admitted.

The other algorithm is more detailed. It takes a look at the specific movie the new customer wants and looks up the (precomputed) data rate for that movie, which differs for black and white versus color, cartoons versus filmed, and even

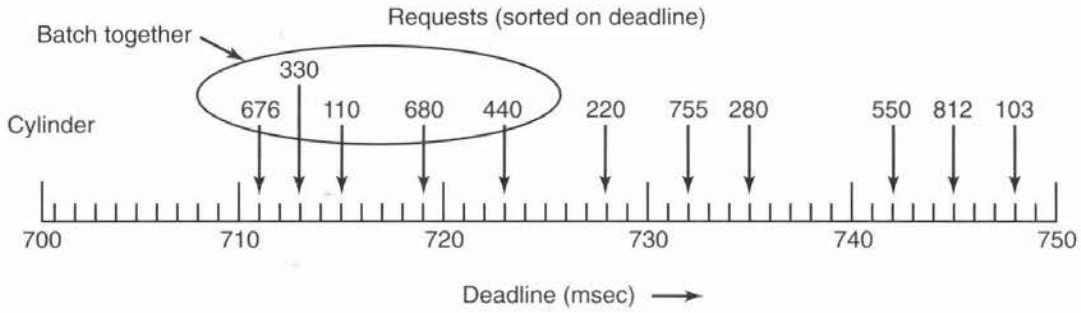


Figure 7-25. The scan-EDF algorithm uses deadlines and cylinder numbers for scheduling.

love stories versus war films. Love stories move slowly with long scenes and slow cross dissolves, all of which compress well whereas war films have many rapid cuts, and fast action, hence many I-frames and large P-frames. If the server has enough capacity for the specific film the new customer wants, then admission is granted; otherwise it is denied.

7.9 RESEARCH ON MULTIMEDIA

Multimedia is a hot topic these days, so there is a considerable amount of research about it. Much of this research is about the content, construction tools, and applications, all of which are beyond the scope of this book. However, some of it involves operating system structure, either writing a new multimedia operating system (Brandwein et al., 1994), or adding multimedia support to an existing operating system (Mercer, 1994). A related area is the design of multimedia servers (Bernhardt and Biersack, 1996; Heybey et al., 1996; Lougher et al., 1994; and Wong and Lee, 1997).

Some papers on multimedia are not about complete new systems, but about algorithms useful in multimedia systems. A popular topic has been real-time CPU scheduling for multimedia (Baker-Harvey, 1999; Bolosky et al., 1997; Dan et al., 1994; Goyal et al., 1996; Jones et al., 1997; Nieh and Lam, 1997; and Wu and Shu, 1996). Another topic that has been examined is disk scheduling for multimedia (Lee et al., 1997; Rompogiannakis et al., 1998; and Wang et al., 1999). File placement and load management on video servers are also important (Gafsi and Biersack, 1999; Shenoy and Vin, 1999; Shenoy et al., 1999; and Venkatasubramanian and Ramanathan, 1997) as is merging video streams to reduce bandwidth requirements (Eager et al., 1999).

In the text, we discussed how movie popularity affects placement on the video server. This topic is an ongoing area of research (Bisdikian and Patel, 1995; and

Griwodz et al., 1997). Finally, security and privacy in multimedia (e.g., in video-conferencing) are also subjects of research interest (Adams and Sasse, 1999; and Honeyman et. al, 1998)

7.10 SUMMARY

Multimedia is an up-and-coming use of computers. Due to the large sizes of multimedia files and their stringent real-time playback requirements, operating systems designed for text are not optimal for multimedia. Multimedia files consist of multiple, parallel tracks, usually one video and at least one audio and sometimes subtitle tracks as well. These must all be synchronized during playback.

Audio is recorded by sampling the volume periodically, usually 44,100 times/sec (for CD quality sound). Compression can be applied to the audio signal, giving a uniform compression rate of about 10x. Video compression uses both intraframe compression (JPEG) and interframe compression (MPEG). The latter represents P-frames as differences from the previous frame. B-frames can be based either on the previous frame or the next frame.

Multimedia needs real-time scheduling in order to meet its deadlines. Two algorithms are commonly used. The first is rate monotonic scheduling, which is a static preemptive algorithm that assigns fixed priorities to processes based on their periods. The second is earliest deadline first, which is a dynamic algorithm that always chooses the process with the closest deadline. EDF is more complicated, but it can achieve 100% utilization, something that RMS cannot achieve.

Multimedia file systems usually use a push model rather than a pull model. Once a stream is started, the bits come off the disk without further user requests. This approach is radically different from conventional operating systems, but is needed to meet the real-time requirements.

Files can be stored contiguously or not. In the latter case, the unit can be variable length (one block is one frame) or fixed length (one block is many frames). These approaches have different trade-offs.

File placement on the disk affects performance. When there are multiple files, the organ-pipe algorithm is sometimes used. Striping files across multiple disks, either wide or narrow, is common. Block and file caching strategies are also widely employed to improve performance.

PROBLEMS

1. What is the bit rate for uncompressed full-color XGA running at 25 frames/sec? Can a stream at this rate come off an UltraWide SCSI disk?

2. Can uncompressed black-and-white NTSC television be sent over fast Ethernet? If so, how many channels at once?

3. HDTV has twice the horizontal resolution of regular TV (1280 versus 640 pixels). Using information provided in the text, how much more bandwidth does it require than standard TV?

4. In Fig. 7-3, there are separate files for fast forward and fast reverse. If a video server is intended to support slow motion as well, is another file required for slow motion in the forward direction? What about in the backward direction?

5. A Compact Disc holds 74 min of music or 650 MB of data. Make an estimate of the compression factor used for music.

6. A sound signal is sampled using a signed 16-bit number (1 sign bit, 15 magnitude bits). What is the maximum quantization noise in percent? Is this a bigger problem for flute concertos or for rock and roll or is it the same for both? Explain your answer.

7. A recording studio is able to make a master digital recording using 20-bit sampling. The final distribution to listeners will use 16 bits. Suggest a way to reduce the effect of quantization noise, and discuss advantages and disadvantages of your scheme.

8. NTSC and PAL both use a 6-MHz broadcast channel, yet NTSC has 30 frames/sec whereas PAL has only 25 frames/sec. How is this possible? Does this mean that if both systems were to use the same color encoding scheme, NTSC would have inherently better quality than PAL? Explain your answer.

9. The DCT transformation uses an 8×8 block, yet the algorithm used for motion compensation uses 16×16. Does this difference cause problems, and if so, how are they solved in MPEG?

10. In Fig. 7-10 we saw how MPEG works with a stationary background and a moving actor. Suppose that an MPEG video is made from a scene in which the camera is mounted on a tripod and pans slowing from left to right at a speed such that no two consecutive frames are the same. Do all the frames have to be I-frames now? Why or why not?

11. Suppose that each of the three processes in Fig. 7-11 is accompanied by a process that supports an audio stream running with the same period as its video process, so audio buffers can be updated between video frames. All three of these audio processes are identical. How much CPU time is available for each burst of an audio process?

12. Two real-time processes are running on a computer. The first one runs every 25 msec for 10 msec. The second one runs every 40 msec for 15 msec. Will RMS always work for them?

13. The CPU of a video server has a utilization of 65%. How many movies can it show using RMS scheduling?

14. In Fig. 7-13, EDF keeps the CPU busy 100% of the time up to $t = 150$. It cannot keep the CPU busy indefinitely because there is only 975-msec work per second for it to do so. Extend the figure beyond 150 msec and determine when the CPU first goes idle with EDF.

15. A DVD can hold enough data for a full-length movie and the transfer rate is adequate to display a television-quality program. Why not just use a "farm" of many DVD drives as the data source for a video server?

16. The operators of a near video-on-demand system have discovered that people in a certain city are not willing to wait more than 6 minutes for a movie to start. How many parallel streams do they need for a 3-hour movie?

17. Consider a system using the scheme of Abram-Profeta and Shin in which the video server operator wishes customers to be able to search forward or backward for 1 min entirely locally. Assuming the video stream is MPEG-2 at 4 Mbps, how much buffer space must each customer have locally?

18. A video-on-demand system for HDTV uses the small block model of Fig. 7-18(a) with a 1-KB disk block. If the video resolution is 1280×720 and the data stream is 12 Mbps, how much disk space is wasted on internal fragmentation in a 2-hour movie using NTSC?

19. Consider the storage allocation scheme of Fig. 7-18(a) for NTSC and PAL. For a given disk block and movie size, does one of them suffer more internal fragmentation than the other? If so, which one is better and why?

20. Consider the two alternatives shown in Fig. 7-18. Does the shift toward HDTV favor either of these systems over the other? Discuss.

21. The near video-on-demand scheme of Chen and Thapar works best when each frame set is the same size. Suppose that a movie is being shown in 24 simultaneous streams and that one frame in 10 is an I-frame. Also assume that I-frames are 10 times larger than P-frames. B-frames are the same size as P-frames. What is the probability that a buffer equal to 4 I-frames and 20 P-frames will not be big enough? Do you think that such a buffer size is acceptable? To make the problem tractable, assume that frame types are randomly and independently distributed over the streams.

22. The end result of Fig. 7-16 is that the play point is not in the middle of the buffer any more. Devise a scheme to have at least 5 min behind the play point and 5 min ahead of it. Make any reasonable assumptions you have to, but state them explicitly.

23. The design of Fig. 7-17 requires that all language tracks be read on each frame. Suppose that the designers of a video server have to support a large number of languages, but do not want to devote so much RAM to buffers to hold each frame. What other alternatives are available, and what are the advantages and disadvantages of each one?

24. A small video server has eight movies. What does Zipf's law predict as the probabilities for the most popular movie, second most popular movie, and so on down to the least popular movie?

25. A 14-GB disk with 1000 cylinders is used to hold 1000 30-sec MPEG-2 video clips running at 4 Mbps. They are stored according to the organ-pipe algorithm. Assuming Zipf's law, what fraction of the time will the disk arm spend in the middle 10 cylinders?

26. Assuming that the relative demand for films *A*, *B*, *C*, and *D* is described by Zipf's law, what is the expected relative utilization of the four disks in Fig. 7-22 for the four striping methods shown?

27. Two video-on-demand customers started watching the same PAL movie 6 sec apart. If the system speeds up one stream and slows down the other to get them to merge, what percent speed up/down is needed to merge them in 3 min?

28. An MPEG-2 video server uses the round scheme of Fig. 7-24 for NTSC video. All the videos come off a single 10,800 rpm UltraWide SCSI disk with an average seek time of 3 msec. How many streams can be supported?

29. Repeat the previous problem, but now assume that scan-EDF reduces the average seek time by 20%. How many streams can now be supported?

30. Repeat the previous problem once more, but now assume that each frame is striped across four disks, with scan-EDF giving the 20% on each disk. How many streams can now be supported.

31. The text describes using a batch of five data requests to schedule the situation described in Fig. 7-25(a). If all requests take an equal amount of time, what is the maximum time per request allowable in this example?

32. Many of the bitmap images that are supplied for generating computer "wallpaper" use few colors and are easily compressed. A simple compression scheme is the following: choose a data value that does not appear in the input file, and use it as a flag. Read the file, byte by byte, looking for repeated byte values. Copy single values and bytes repeated up to three times directly to the output file. When a repeated string of 4 or more bytes is found, write to the output file a string of three bytes consisting of the flag byte, a byte indicating a count from 4 to 255, and the actual value found in the input file. Write a compression program using this algorithm, and a decompression program that can restore the original file. Extra credit: how can you deal with files that contain the flag byte in their data?

33. Computer animation is accomplished by displaying a sequence of slightly different images. Write a program to calculate the byte by byte difference between two uncompressed bitmap images of the same dimensions. The output will be the same size as the input files, of course. Use this difference file as input to the compression program of the previous problem, and compare the effectiveness of this approach with compression of individual images.

8

MULTIPLE PROCESSOR SYSTEMS

Since its inception, the computer industry has been driven by an endless quest for more and more computing power. The ENIAC could perform 300 operations per second, easily 1000 times faster than any calculator before it, yet people were not satisfied. We now have machines a million times faster than the ENIAC and still there is a demand for yet more horsepower. Astronomers are trying to make sense of the universe, biologists are trying to understand the implications of the human genome, and aeronautical engineers are interested in building safer and more efficient aircraft, and all want more CPU cycles. However much computing power there is, it is never enough.

In the past, the solution was always to make the clock run faster. Unfortunately, we are beginning to hit some fundamental limits on clock speed. According to Einstein's special theory of relativity, no electrical signal can propagate faster than the speed of light, which is about 30 cm/nsec in vacuum and about 20 cm/nsec in copper wire or optical fiber. This means that in a computer with a 10-GHz clock, the signals cannot travel more than 2 cm in total. For a 100-GHz computer the total path length is at most 2 mm. A 1-THz (1000 GHz) computer will have to be smaller than 100 microns, just to let the signal get from one end to the other and back once with a single clock cycle.

Making computers this small may be possible, but then we hit another fundamental problem: heat dissipation. The faster runs the computer, the more heat it generates, and the smaller the computer, the harder it is to get rid of this heat.

Already on high-end Pentium systems, the CPU cooler is bigger than the CPU itself. All in all, going from 1 MHz to 1 GHz simply required incrementally better engineering of the chip manufacturing process. Going from 1 GHz to 1 THz is going to require a radically different approach.

One approach to greater speed is through massively parallel computers. These machines consist of many CPUs, each of which runs at "normal" speed (whatever that may mean in a given year), but which collectively have far more computing power than a single CPU. Systems with 1000 CPUs are now commercially available. Systems with 1 million CPUs are likely to be built in the coming decade. While there are other potential approaches to greater speed, such as biological computers, in this chapter we will focus on systems with multiple conventional CPUs.

Highly parallel computers are often used for heavy number crunching. Problems such as predicting the weather, modeling airflow around an aircraft wing, simulating the world economy, or understanding drug-receptor interactions in the brain are all computationally intensive. Their solutions require long runs on many CPUs at once. The multiple processor systems discussed in this chapter are widely-used for these and similar problems in science and engineering, among other areas.

Another relevant development is the incredibly rapid growth of the Internet. It was originally designed as a prototype for a fault-tolerant military control system, then became popular among academic computer scientists, and has recently acquired many new uses. One of these is linking up thousands of computers all over the world to work together on large scientific problems. In a sense, a system consisting of 1000 computers spread all over the world is no different than one consisting of 1000 computers in a single room, although the delay and other technical characteristics are different. We will also consider these systems in this chapter.

Putting 1 million unrelated computers in a room is easy to do provided that you have enough money and a sufficiently large room. Spreading 1 million unrelated computers around the world is even easier since it finesses the second problem. The trouble comes in when you want them to communicate with one another to work together on a single problem. As a consequence, a great deal of work has been done on the interconnection technology, and different interconnect technologies have led to qualitatively different kinds of systems and different software organizations.

All communication between electronic (or optical) components ultimately comes down to sending messages—well-defined bit strings—between them. The differences are in the time scale, distance scale, and logical organization involved. At one extreme are the shared-memory multiprocessors, systems in which somewhere between two and about 1000 CPUs communicate via a shared memory. In this model, every CPU has equal access to the entire physical memory, and can read and write individual words using LOAD and STORE instructions. Accessing a

memory word usually takes 10–50 nsec. While this model, illustrated in Fig. 8-1(a), sounds simple, actually implementing it is far from simple and usually involves considerable message passing under the covers, as we will explain shortly.

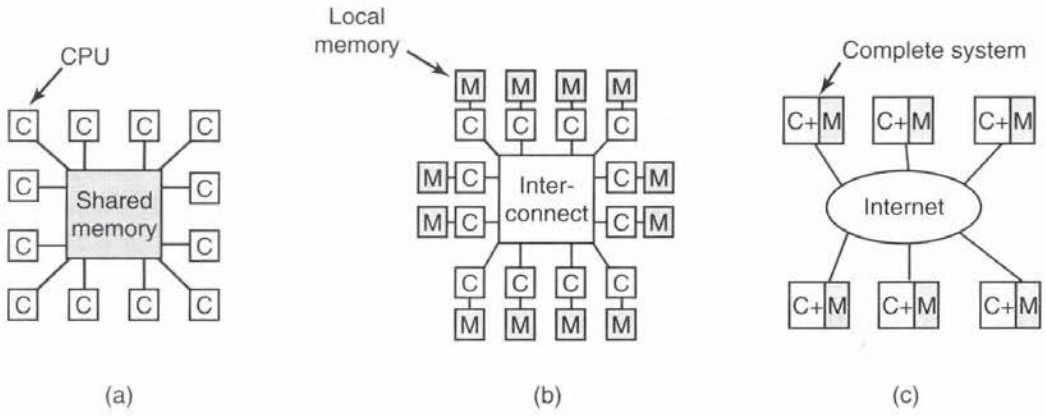


Figure 8-1. (a) A shared-memory multiprocessor. (b) A message-passing multicomputer. (c) A wide area distributed system.

Next comes the system of Fig. 8-1(b) in which a number of CPU-memory pairs are connected by some kind of high-speed interconnect. This kind of system is called a message-passing multicomputer. Each memory is local to a single CPU and can be accessed only by that CPU. The machines communicate by sending multiword messages over the interconnect. With a good interconnect, a short message can be sent in 10–50 μsec, but still far longer than the memory access time of Fig. 8-1(a). There is no shared global memory in this design. Multicomputers (i.e., message-passing systems) are much easier to build than (shared-memory) multiprocessors but they are harder to program. Thus each genre has its fans.

The third model, illustrated in Fig. 8-1(c), connects complete computer systems over a wide area network, such as the Internet, to form a **distributed system**. Each of these has its own memory, of course, and the systems communicate by message passing. The only real difference between Fig. 8-1(c) and Fig. 8-1(b) is that in the former, complete computers are used and message times are often 10–50 msec. This long delay forces these **loosely-coupled** systems to be used in different ways than the **tightly-coupled** systems of Fig. 8-1(b). The three types of systems differ in their delays by something like three orders of magnitude. That is the difference between a day and three years.

This chapter has three major sections, corresponding to the three models of Fig. 8-1. In each one, we start out with a brief introduction to the relevant hardware. Then we move on to the software, especially the operating system issues for that type of system. As we will see, in each case different issues are present.

8.1 MULTIPROCESSORS

A **shared-memory multiprocessor** (or just multiprocessor henceforth) is a computer system in which two or more CPUs share full access to a common RAM. A program running on any of the CPUs sees a normal (usually paged) virtual address space. The only unusual property this system has is that the CPU can write some value into a memory word and then read the word back and get a different value (because another CPU has changed it). When organized correctly, this property forms the basis of interprocessor communication: one CPU writes some data into memory and another one reads the data out.

For the most part, multiprocessor operating systems are just regular operating systems. They handle system calls, do memory management, provide a file system, and manage I/O devices. Nevertheless, there are some areas in which they have unique features. These include process synchronization, resource management, and scheduling. Below we will first take a brief look at multiprocessor hardware and then move on to these operating systems issues.

8.1.1 Multiprocessor Hardware

Although all multiprocessors have the property that every CPU can address all of memory, some multiprocessors have the additional property that every memory word can be read as fast as every other memory word. These machines are called **UMA** (**Uniform Memory Access**) multiprocessors. In contrast, **NUMA** (**Nonuniform Memory Access**) multiprocessors do not have this property. Why this difference exists will become clear later. We will first examine UMA multiprocessors and then move on to NUMA multiprocessors.

UMA Bus-Based SMP Architectures

The simplest multiprocessors are based on a single bus, as illustrated in Fig. 8-2(a). Two or more CPUs and one or more memory modules all use the same bus for communication. When a CPU wants to read a memory word, it first checks to see if the bus is busy. If the bus is idle, the CPU puts the address of the word it wants on the bus, asserts a few control signals, and waits until the memory puts the desired word on the bus.

If the bus is busy when a CPU wants to read or write memory, the CPU just waits until the bus becomes idle. Herein lies the problem with this design. With two or three CPUs, contention for the bus will be manageable; with 32 or 64 it will be unbearable. The system will be totally limited by the bandwidth of the bus, and most of the CPUs will be idle most of the time.

The solution to this problem is to add a cache to each CPU, as depicted in Fig. 8-2(b). The cache can be inside the CPU chip, next to the CPU chip, on the

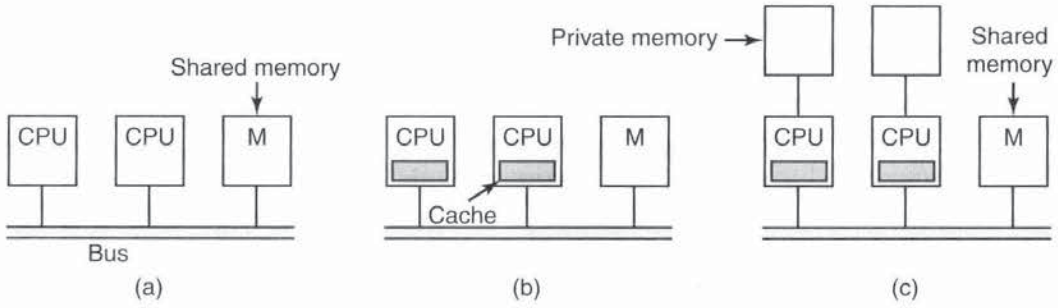


Figure 8-2. Three bus-based multiprocessors. (a) Without caching. (b) With caching. (c) With caching and private memories.

processor board, or some combination of all three. Since many reads can now be satisfied out of the local cache, there will be much less bus traffic, and the system can support more CPUs. In general, caching is not done on an individual word basis but on the basis of 32- or 64-byte blocks. When a word is referenced, its entire block is fetched into the cache of the CPU touching it.

Each cache block is marked as being either read-only (in which case it can be present in multiple caches at the same time), or as read-write (in which case it may not be present in any other caches). If a CPU attempts to write a word that is in one or more remote caches, the bus hardware detects the write and puts a signal on the bus informing all other caches of the write. If other caches have a "clean" copy, that is, an exact copy of what is in memory, they can just discard their copies and let the writer fetch the cache block from memory before modifying it. If some other cache has a "dirty" (i.e., modified) copy, it must either write it back to memory before the write can proceed or transfer it directly to the writer over the bus. Many cache transfer protocols exist.

Yet another possibility is the design of Fig. 8-2(c), in which each CPU has not only a cache, but also a local, private memory which it accesses over a dedicated (private) bus. To use this configuration optimally, the compiler should place all the program text, strings, constants and other read-only data, stacks, and local variables in the private memories. The shared memory is then only used for writable shared variables. In most cases, this careful placement will greatly reduce bus traffic, but it does require active cooperation from the compiler.

UMA Multiprocessors Using Crossbar Switches

Even with the best caching, the use of a single bus limits the size of a UMA multiprocessor to about 16 or 32 CPUs. To go beyond that, a different kind of interconnection network is needed. The simplest circuit for connecting n CPUs to k memories is the **crossbar switch**, shown in Fig. 8-3. Crossbar switches have been used for decades within telephone switching exchanges to connect a group of incoming lines to a set of outgoing lines in an arbitrary way.

At each intersection of a horizontal (incoming) and vertical (outgoing) line is a **crosspoint**. A crosspoint is a small switch that can be electrically opened or closed, depending on whether the horizontal and vertical lines are to be connected or not. In Fig. 8-3(a) we see three crosspoints closed simultaneously, allowing connections between the (CPU, memory) pairs (001, 000), (101, 101), and (110, 010) at the same time. Many other combinations are also possible. In fact, the number of combinations is equal to the number of different ways eight rooks can be safely placed on a chess board.

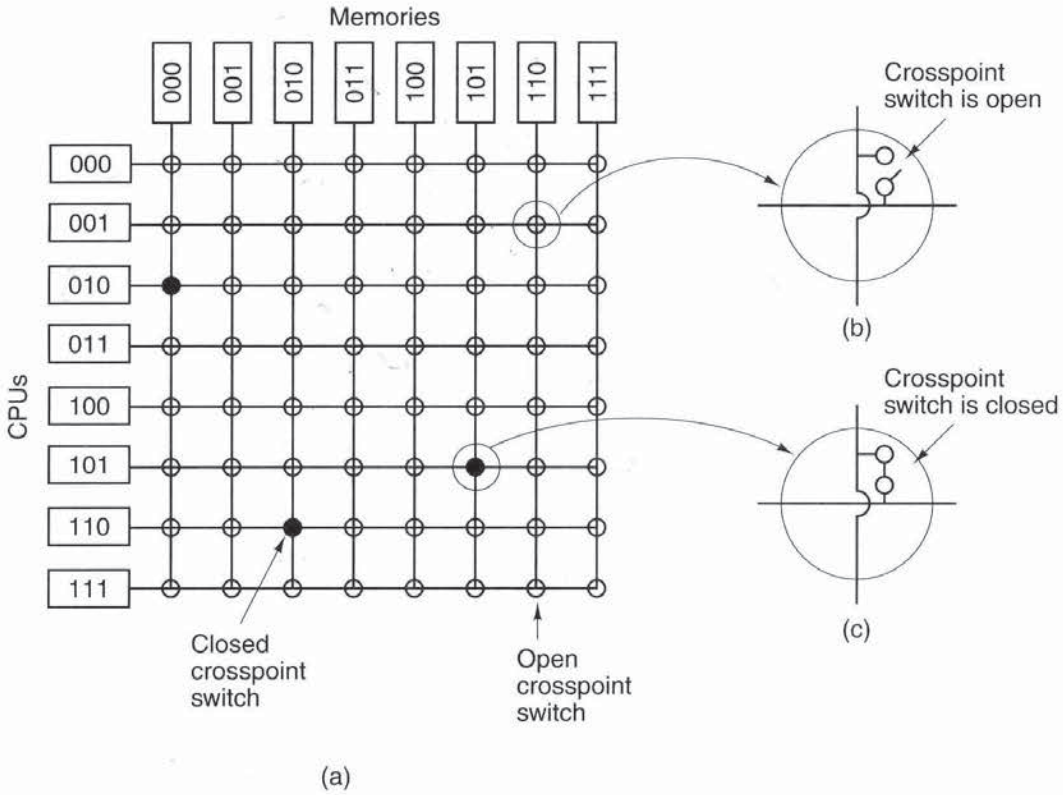


Figure 8-3. (a) An 8 × 8 crossbar switch. (b) An open crosspoint. (c) A closed crosspoint.

One of the nicest properties of the crossbar switch is that it is a **nonblocking network**, meaning that no CPU is ever denied the connection it needs because some crosspoint or line is already occupied (assuming the memory module itself is available). Furthermore, no advance planning is needed. Even if seven arbitrary connections are already set up, it is always possible to connect the remaining CPU to the remaining memory.

One of the worst properties of the crossbar switch is the fact that the number of crosspoints grows as n^2. With 1000 CPUs and 1000 memory modules we need a million crosspoints. Such a large crossbar switch is not feasible. Nevertheless, for medium-sized systems, a crossbar design is workable.

UMA Multiprocessors Using Multistage Switching Networks

A completely different multiprocessor design is based on the humble 2×2 switch shown in Fig. 8-4(a). This switch has two inputs and two outputs. Messages arriving on either input line can be switched to either output line. For our purposes, messages will contain up to four parts, as shown in Fig. 8-4(b). The *Module* field tells which memory to use. The *Address* specifies an address within a module. The *Opcode* gives the operation, such as READ or WRITE. Finally, the optional *Value* field may contain an operand, such as a 32-bit word to be written on a WRITE. The switch inspects the *Module* field and uses it to determine if the message should be sent on *X* or on *Y*.

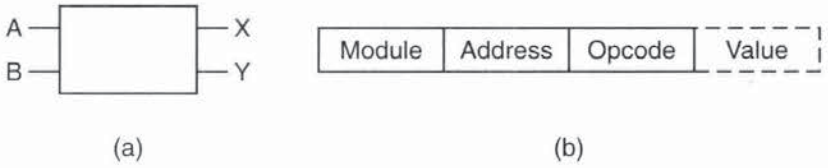


Figure 8-4. (a) A 2×2 switch. (b) A message format.

Our 2×2 switches can be arranged in many ways to build larger **multistage switching networks** (Adams et al., 1987; Bhuyan et al., 1989; and Kumar and Reddy, 1987). One possibility is the no-frills, economy class **omega network**, illustrated in Fig. 8-5. Here we have connected eight CPUs to eight memories using 12 switches. More generally, for n CPUs and n memories we would need $\log_2 n$ stages, with $n/2$ switches per stage, for a total of $(n/2)\log_2 n$ switches, which is a lot better than n^2 crosspoints, especially for large values of n.

The wiring pattern of the omega network is often called the **perfect shuffle**, since the mixing of the signals at each stage resembles a deck of cards being cut in half and then mixed card-for-card. To see how the omega network works, suppose that CPU 011 wants to read a word from memory module 110. The CPU sends a READ message to switch 1D containing 110 in the *Module* field. The switch takes the first (i.e., leftmost) bit of 110 and uses it for routing. A 0 routes to the upper output and a 1 routes to the lower one. Since this bit is a 1, the message is routed via the lower output to 2D.

All the second-stage switches, including 2D, use the second bit for routing. This, too, is a 1, so the message is now forwarded via the lower output to 3D. Here the third bit is tested and found to be a 0. Consequently, the message goes out on the upper output and arrives at memory 110, as desired. The path followed by this message is marked in Fig. 8-5 by the letter *a*.

As the message moves through the switching network, the bits at the left-hand end of the module number are no longer needed. They can be put to good use by recording the incoming line number there, so the reply can find its way back. For

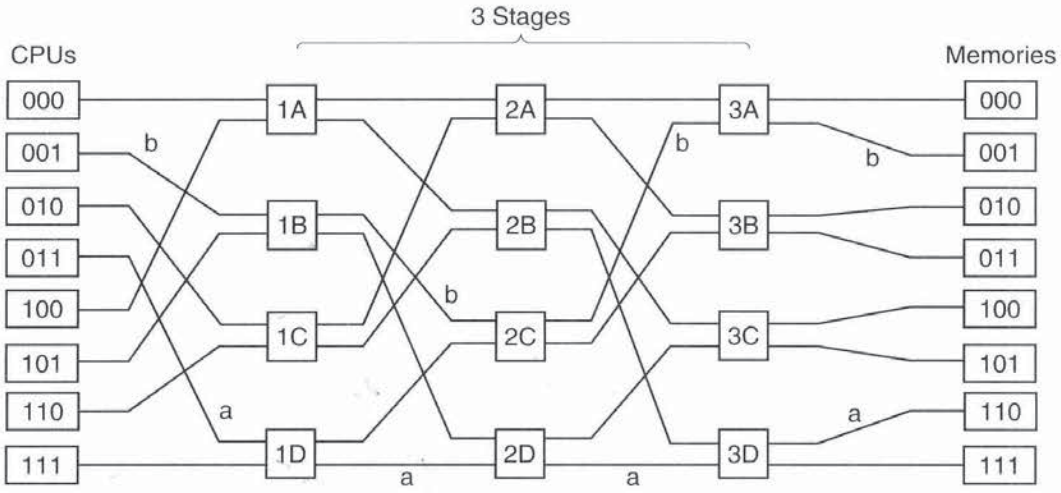


Figure 8-5. An omega switching network.

path *a*, the incoming lines are 0 (upper input to 1D), 1 (lower input to 2D), and 1 (lower input to 3D), respectively. The reply is routed back using 011, only reading it from right to left this time.

At the same time all this is going on, CPU 001 wants to write a word to memory module 001. An analogous process happens here, with the message routed via the upper, upper, and lower outputs, respectively, marked by the letter *b*. When it arrives, its *Module* field reads 001, representing the path it took. Since these two requests do not use any of the same switches, lines, or memory modules, they can proceed in parallel.

Now consider what would happen if CPU 000 simultaneously wanted to access memory module 000. Its request would come into conflict with CPU 001's request at switch 3A. One of them would have to wait. Unlike the crossbar switch, the omega network is a **blocking network**. Not every set of requests can be processed simultaneously. Conflicts can occur over the use of a wire or a switch, as well as between requests *to* memory and replies *from* memory.

It is clearly desirable to spread the memory references uniformly across the modules. One common technique is to use the low-order bits as the module number. Consider, for example, a byte-oriented address space for a computer that mostly accesses 32-bit words. The 2 low-order bits will usually be 00, but the next 3 bits will be uniformly distributed. By using these 3 bits as the module number, consecutively addressed words will be in consecutive modules. A memory system in which consecutive words are in different modules is said to be **interleaved**. Interleaved memories maximize parallelism because most memory references are to consecutive addresses. It is also possible to design switching networks that are nonblocking and which offer multiple paths from each CPU to each memory module, to spread the traffic better.

NUMA Multiprocessors

Single-bus UMA multiprocessors are generally limited to no more than a few dozen CPUs and crossbar or switched multiprocessors need a lot of (expensive) hardware and are not that much bigger. To get to more than 100 CPUs, something has to give. Usually, what gives is the idea that all memory modules have the same access time. This concession leads to the idea of NUMA multiprocessors, as mentioned above. Like their UMA cousins, they provide a single address space across all the CPUs, but unlike the UMA machines, access to local memory modules is faster than access to remote ones. Thus all UMA programs will run without change on NUMA machines, but the performance will be worse than on a UMA machine at the same clock speed.

NUMA machines have three key characteristics that all of them possess and which together distinguish them from other multiprocessors:

1. There is a single address space visible to all CPUs.
2. Access to remote memory is via LOAD and STORE instructions.
3. Access to remote memory is slower than access to local memory.

When the access time to remote memory is not hidden (because there is no caching), the system is called **NC-NUMA**. When coherent caches are present, the system is called **CC-NUMA (Cache-Coherent NUMA)**.

The most popular approach for building large CC-NUMA multiprocessors currently is the **directory-based multiprocessor**. The idea is to maintain a database telling where each cache line is and what its status is. When a cache line is referenced, the database is queried to find out where it is and whether it is clean or dirty (modified). Since this database must be queried on every instruction that references memory, it must be kept in extremely-fast special-purpose hardware that can respond in a fraction of a bus cycle.

To make the idea of a directory-based multiprocessor somewhat more concrete, let us consider as a simple (hypothetical) example, a 256-node system, each node consisting of one CPU and 16 MB of RAM connected to the CPU via a local bus. The total memory is 2^{32} bytes, divided up into 2^{26} cache lines of 64 bytes each. The memory is statically allocated among the nodes, with 0–16M in node 0, 16M–32M in node 1, and so on. The nodes are connected by an interconnection network, as shown in Fig. 8-6(a). Each node also holds the directory entries for the 2^{18} 64-byte cache lines comprising its 2^{24} byte memory. For the moment, we will assume that a line can be held in at most one cache.

To see how the directory works, let us trace a LOAD instruction from CPU 20 that references a cached line. First the CPU issuing the instruction presents it to its MMU, which translates it to a physical address, say, 0x24000108. The MMU splits this address into the three parts shown in Fig. 8-6(b). In decimal, the three

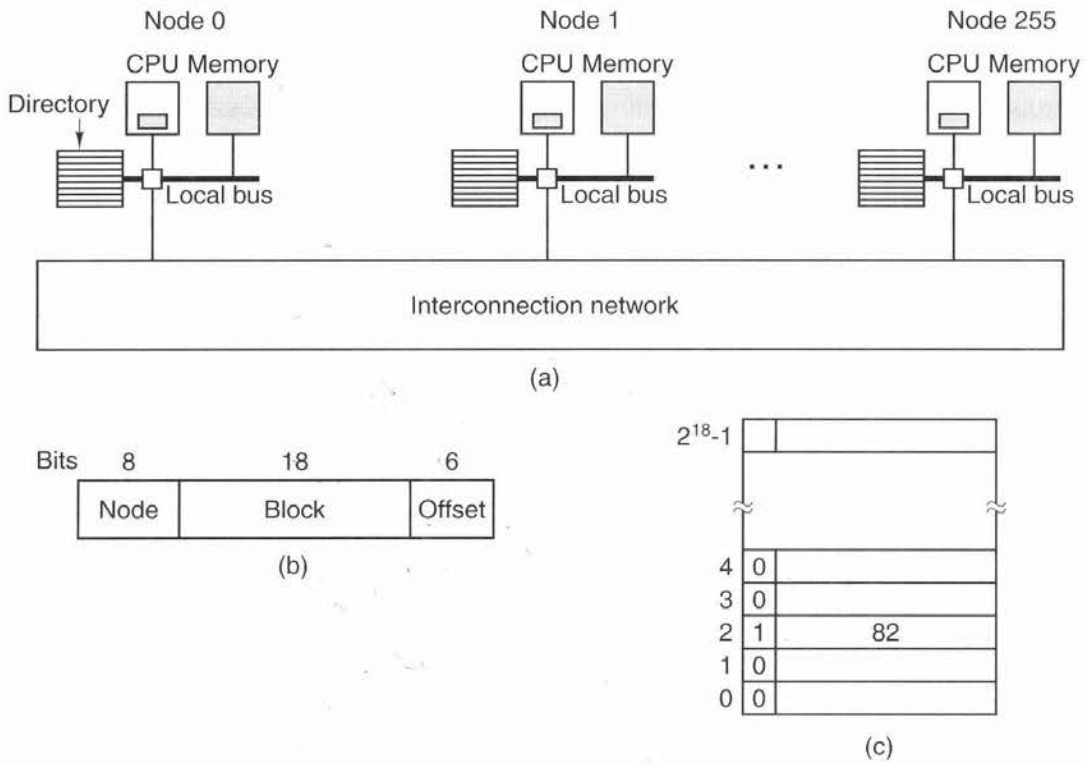


Figure 8-6. (a) A 256-node directory-based multiprocessor. (b) Division of a 32-bit memory address into fields. (c) The directory at node 36.

parts are node 36, line 4, and offset 8. The MMU sees that the memory word referenced is from node 36, not node 20, so it sends a request message through the interconnection network to the line's home node, 36, asking whether its line 4 is cached, and if so, where.

When the request arrives at node 36 over the interconnection network, it is routed to the directory hardware. The hardware indexes into its table of 2^{18} entries, one for each of its cache lines and extracts entry 4. From Fig. 8-6(c) we see that the line is not cached, so the hardware fetches line 4 from the local RAM, sends it back to node 20, and updates directory entry 4 to indicate that the line is now cached at node 20.

Now let us consider a second request, this time asking about node 36's line 2. From Fig. 8-6(c) we see that this line is cached at node 82. At this point the hardware could update directory entry 2 to say that the line is now at node 20 and then send a message to node 82 instructing it to pass the line to node 20 and invalidate its cache. Note that even a so-called "shared-memory multiprocessor" has a lot of message passing going on under the hood.

As a quick aside, let us calculate how much memory is being taken up by the directories. Each node has 16 MB of RAM and 2^{18} 9-bit entries to keep track of that RAM. Thus the directory overhead is about 9×2^{18} bits divided by 16 MB or

about 1.76 percent, which is generally acceptable (although it has to be high-speed memory, which increases its cost). Even with 32-byte cache lines the overhead would only be 4 percent. With 128-byte cache lines, it would be under 1 percent.

An obvious limitation of this design is that a line can be cached at only one node. To allow lines to be cached at multiple nodes, we would need some way of locating all of them, for example, to invalidate or update them on a write. Various options are possible to allow caching at several nodes at the same time, but a discussion of these is beyond the scope of this book.

8.1.2 Multiprocessor Operating System Types

Let us now turn from multiprocessor hardware to multiprocessor software, in particular, multiprocessor operating systems. Various organizations are possible. Below we will study three of them.

Each CPU Has Its Own Operating System

The simplest possible way to organize a multiprocessor operating system is to statically divide memory into as many partitions as there are CPUs and give each CPU its own private memory and its own private copy of the operating system. In effect, the n CPUs then operate as n independent computers. One obvious optimization is to allow all the CPUs to share the operating system code and make private copies of only the data, as shown in Fig. 8-7.

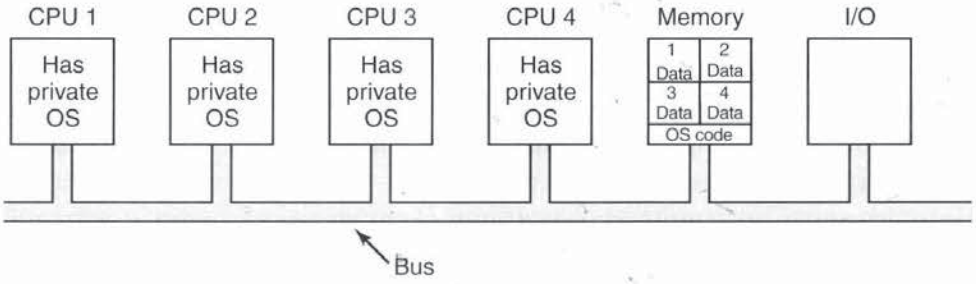


Figure 8-7. Partitioning multiprocessor memory among four CPUs, but sharing a single copy of the operating system code. The boxes marked Data are the operating system's private data for each CPU.

This scheme is still better than having n separate computers since it allows all the machines to share a set of disks and other I/O devices, and it also allows the memory to be shared flexibly. For example, if one day an unusually large program has to be run, one of the CPUs can be allocated an extra large portion of memory for the duration of that program. In addition, processes can efficiently communicate with one another by having, say a producer be able to write data into memory and have a consumer fetch it from the place the producer wrote it.

Still, from an operating systems' perspective, having each CPU have its own operating system is as primitive as it gets.

It is worth explicitly mentioning four aspects of this design that may not be obvious. First, when a process makes a system call, the system call is caught and handled on its own CPU using the data structures in that operating system's tables.

Second, since each operating system has its own tables, it also has its own set of processes that it schedules by itself. There is no sharing of processes. If a user logs into CPU 1, all of his processes run on CPU 1. As a consequence, it can happen that CPU 1 is idle while CPU 2 is loaded with work.

Third, there is no sharing of pages. It can happen that CPU 1 has pages to spare while CPU 2 is paging continuously. There is no way for CPU 2 to borrow some pages from CPU 1 since the memory allocation is fixed.

Fourth, and worst, if the operating system maintains a buffer cache of recently used disk blocks, each operating system does this independently of the other ones. Thus it can happen that a certain disk block is present and dirty in multiple buffer caches at the same time, leading to inconsistent results. The only way to avoid this problem is to eliminate the buffer caches. Doing so is not hard, but it hurts performance considerably.

Master-Slave Multiprocessors

For these reasons, this model is rarely used any more, although it was used in the early days of multiprocessors, when the goal was to port existing operating systems to some new multiprocessor as fast as possible. A second model is shown in Fig. 8-8. Here, one copy of the operating system and its tables are present on CPU 1 and not on any of the others. All system calls are redirected to CPU 1 for processing there. CPU 1 may also run user processes if there is CPU time left over. This model is called **master-slave** since CPU 1 is the master and all the others are slaves.

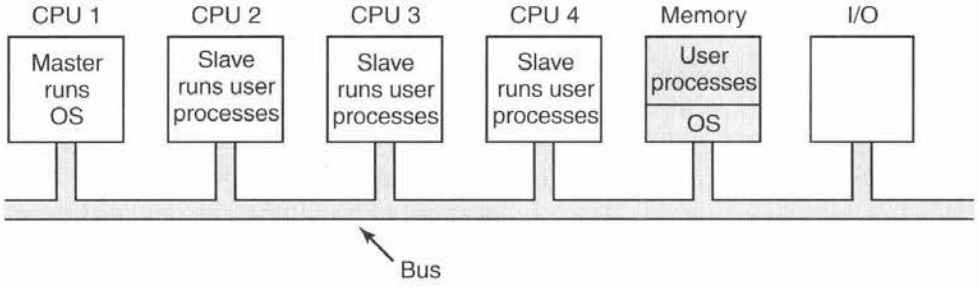


Figure 8-8. A master-slave multiprocessor model.

The master-slave model solves most of the problems of the first model. There is a single data structure (e.g., one list or a set of prioritized lists) that keeps track

of ready processes. When a CPU goes idle, it asks the operating system for a process to run and it is assigned one. Thus it can never happen that one CPU is idle while another is overloaded. Similarly, pages can be allocated among all the processes dynamically and there is only one buffer cache, so inconsistencies never occur.

The problem with this model is that with many CPUs, the master will become a bottleneck. After all, it must handle all system calls from all CPUs. If, say, 10% of all time is spent handling system calls, then 10 CPUs will pretty much saturate the master, and with 20 CPUs it will be completely overloaded. Thus this model is simple and workable for small multiprocessors, but for large ones it fails.

Symmetric Multiprocessors

Our third model, the **SMP** (**Symmetric MultiProcessor**), eliminates this asymmetry. There is one copy of the operating system in memory, but any CPU can run it. When a system call is made, the CPU on which the system call was made traps to the kernel and processes the system call. The SMP model is illustrated in Fig. 8-9.

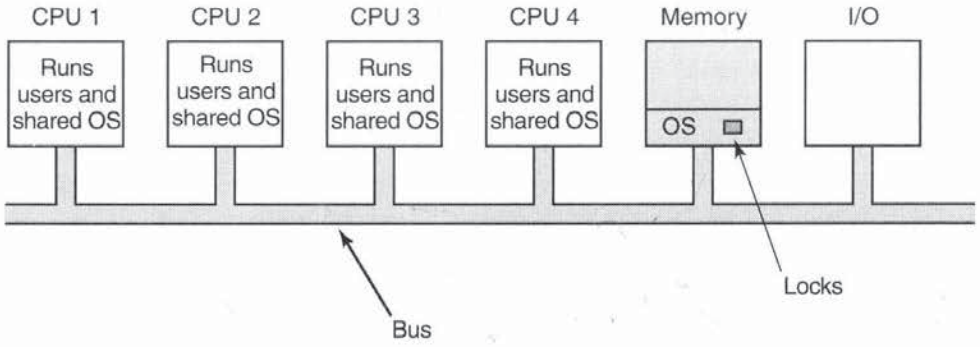


Figure 8-9. The SMP multiprocessor model.

This model balances processes and memory dynamically, since there is only one set of operating system tables. It also eliminates the master CPU bottleneck, since there is no master, but it introduces its own problems. In particular, if two or more CPUs are running operating system code at the same time, disaster will result. Imagine two CPUs simultaneously picking the same process to run or claiming the same free memory page. The simplest way around these problems is to associate a mutex (i.e., lock) with the operating system, making the whole system one big critical region. When a CPU wants to run operating system code, it must first acquire the mutex. If the mutex is locked, it just waits. In this way, any CPU can run the operating system, but only one at a time.

This model works, but is almost as bad as the master-slave model. Again, suppose that 10% of all run time is spent inside the operating system. With 20 CPUs, there will be long queues of CPUs waiting to get in. Fortunately, it is easy

to improve. Many parts of the operating system are independent of one another. For example, there is no problem with one CPU running the scheduler while another CPU is handling a file system call and a third one is processing a page fault.

This observation leads to splitting the operating system up into independent critical regions that do not interact with one another. Each critical region is protected by its own mutex, so only one CPU at a time can execute it. In this way, far more parallelism can be achieved. However, it may well happen that some tables, such as the process table, are used by multiple critical regions. For example, the process table is needed for scheduling, but also for the fork system call and also for signal handling. Each table that may be used by multiple critical regions needs its own mutex. In this way, each critical region can be executed by only one CPU at a time and each critical table can be accessed by only one CPU at a time.

Most modern multiprocessors use this arrangement. The hard part about writing the operating system for such a machine is not that the actual code is so different from a regular operating system. It is not. The hard part is splitting it into critical regions that can be executed concurrently by different CPUs without interfering with one another, not even in subtle, indirect ways. In addition, every table used by two or more critical regions must be separately protected by a mutex and all code using the table must use the mutex correctly.

Furthermore, great care must be taken to avoid deadlocks. If two critical regions both need table *A* and table *B*, and one of them claims *A* first and the other claims *B* first, sooner or later a deadlock will occur and nobody will know why. In theory, all the tables could be assigned integer values and all the critical regions could be required to acquire tables in increasing order. This strategy avoids deadlocks, but it requires the programmer to think very carefully which tables each critical region needs to make the requests in the right order.

As the code evolves over time, a critical region may need a new table it did not previously need. If the programmer is new and does not understand the full logic of the system, then the temptation will be to just grab the mutex on the table at the point it is needed and release it when it is no longer needed. However reasonable this may appear, it may lead to deadlocks, which the user will perceive as the system freezing. Getting it right is not easy and keeping it right over a period of years in the face of changing programmers is very difficult.

8.1.3 Multiprocessor Synchronization

The CPUs in a multiprocessor frequently need to synchronize. We just saw the case in which kernel critical regions and tables have to be protected by mutexes. Let us now take a close look at how this synchronization actually works in a multiprocessor. It is far from trivial, as we will soon see.

To start with, proper synchronization primitives are really needed. If a process on a uniprocessor makes a system call that requires accessing some critical kernel table, the kernel code can just disable interrupts before touching the table. It can then do its work knowing that it will be able to finish without any other process sneaking in and touching the table before it is finished. On a multiprocessor, disabling interrupts affects only the CPU doing the disable. Other CPUs continue to run and can still touch the critical table. As a consequence, a proper mutex protocol must be used and respected by all CPUs to guarantee that mutual exclusion works.

The heart of any practical mutex protocol is an instruction that allows a memory word to be inspected and set in one indivisible operation. We saw how TSL (Test and Set Lock) was used in Fig. 2-22 to implement critical regions. As we discussed earlier, what this instruction does is read out a memory word and store it in a register. Simultaneously, it writes a 1 (or some other nonzero value) into the memory word. Of course, it takes two separate bus cycles to perform the memory read and memory write. On a uniprocessor, as long as the instruction cannot be broken off halfway, TSL always works as expected.

Now think about what could happen on a multiprocessor. In Fig. 8-10 we see the worst case timing, in which memory word 1000, being used as a lock is initially 0. In step 1, CPU 1 reads out the word and gets a 0. In step 2, before CPU 1 has a chance to rewrite the word to 1, CPU 2 gets in and also reads the word out as a 0. In step 3, CPU 1 writes a 1 into the word. In step 4, CPU 2 also writes a 1 into the word. Both CPUs got a 0 back from the TSL instruction, so both of them now have access to the critical region and the mutual exclusion fails.

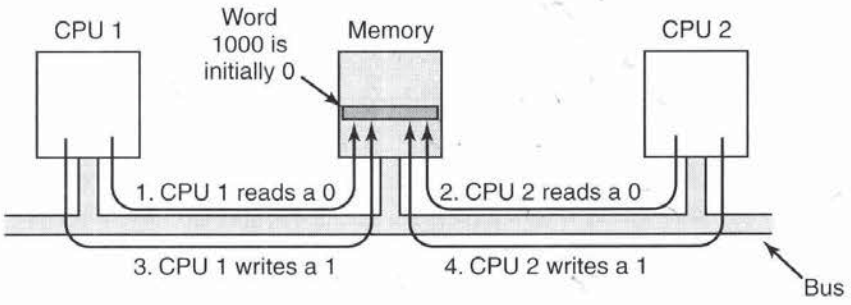


Figure 8-10. The TSL instruction can fail if the bus cannot be locked. These four steps show a sequence of events where the failure is demonstrated.

To prevent this problem, the TSL instruction must first lock the bus, preventing other CPUs from accessing it, then do both memory accesses, then unlock the bus. Typically, locking the bus is done by requesting the bus using the usual bus request protocol, then asserting (i.e., setting to a logical 1) some special bus line until *both* cycles have been completed. As long as this special line is being asserted, no other CPU will be granted bus access. This instruction can only be

implemented on a bus that has the necessary lines and (hardware) protocol for using them. Modern buses have these facilities, but on earlier ones that did not, it was not possible to implement TSL correctly. This is why Peterson's protocol was invented, to synchronize entirely in software (Peterson, 1981).

If TSL is correctly implemented and used, it guarantees that mutual exclusion can be made to work. However, this mutual exclusion method uses a **spin lock** because the requesting CPU just sits in a tight loop testing the lock as fast as it can. Not only does it completely waste the time of the requesting CPU (or CPUs), but it may also put a massive load on the bus or memory, seriously slowing down all other CPUs trying to do their normal work.

At first glance, it might appear that the presence of caching should eliminate the problem of bus contention, but it does not. In theory, once the requesting CPU has read the lock word, it should get a copy in its cache. As long as no other CPU attempts to use the lock, the requesting CPU should be able to run out of its cache. When the CPU owning the lock writes a 1 to it to release it, the cache protocol automatically invalidates all copies of it in remote caches requiring the correct value to be fetched again.

The problem is that caches operate in blocks of 32 or 64 bytes. Usually, the words surrounding the lock are needed by the CPU holding the lock. Since the TSL instruction is a write (because it modifies the lock), it needs exclusive access to the cache block containing the lock. Therefore every TSL invalidates the block in the lock holder's cache and fetches a private, exclusive copy for the requesting CPU. As soon as the lock holder touches a word adjacent to the lock, the cache block is moved to its machine. Consequently, the entire cache block containing the lock is constantly being shuttled between the lock owner and the lock requester, generating even more bus traffic than individual reads on the lock word would have.

If we could get rid of all the TSL-induced writes on the requesting side, we could reduce cache thrashing appreciably. This goal can be accomplished by having the requesting CPU first do a pure read to see if the lock is free. Only if the lock appears to be free does it do a TSL to actually acquire it. The result of this small change is that most of the polls are now reads instead of writes. If the CPU holding the lock is only reading the variables in the same cache block, they can each have a copy of the cache block in shared read-only mode, eliminating all the cache block transfers. When the lock is finally freed, the owner does a write, which requires exclusive access, thus invalidating all the other copies in remote caches. On the next read by the requesting CPU, the cache block will be reloaded. Note that if two or more CPUs are contending for the same lock, it can happen that both see that it is free simultaneously, and both do a TSL simultaneously to acquire it. Only one of these will succeed, so there is no race condition here because the real acquisition is done by the TSL instruction, and this instruction is atomic. Seeing that the lock is free and then trying to grab it immediately with a CX u TSL does not guarantee that you get it. Someone else might win.

Another way to reduce bus traffic is to use the Ethernet binary exponential backoff algorithm (Anderson, 1990). Instead of continuously polling, as in Fig. 2-22, a delay loop can be inserted between polls. Initially the delay is one instruction. If the lock is still busy, the delay is doubled to two instructions, then four instructions and so on up to some maximum. A low maximum gives fast response when the lock is released, but wastes more bus cycles on cache thrashing. A high maximum reduces cache thrashing at the expense of not noticing that the lock is free so quickly. Binary exponential backoff can be used with or without the pure reads preceding the TSL instruction.

An even better idea is to give each CPU wishing to acquire the mutex its own private lock variable to test, as illustrated in Fig. 8-11 (Mellor-Crummey and Scott, 1991). The variable should reside in an otherwise unused cache block to avoid conflicts. The algorithm works by having a CPU that fails to acquire the lock allocate a lock variable and attach itself to the end of a list of CPUs waiting for the lock. When the current lock holder exits the critical region, it frees the private lock that the first CPU on the list is testing (in its own cache). This CPU then enters the critical region. When it is done, it frees the lock its successor is using, and so on. Although the protocol is somewhat complicated (to avoid having two CPUs attach themselves to the end of the list simultaneously), it is efficient and starvation free. For all the details, readers should consult the paper.

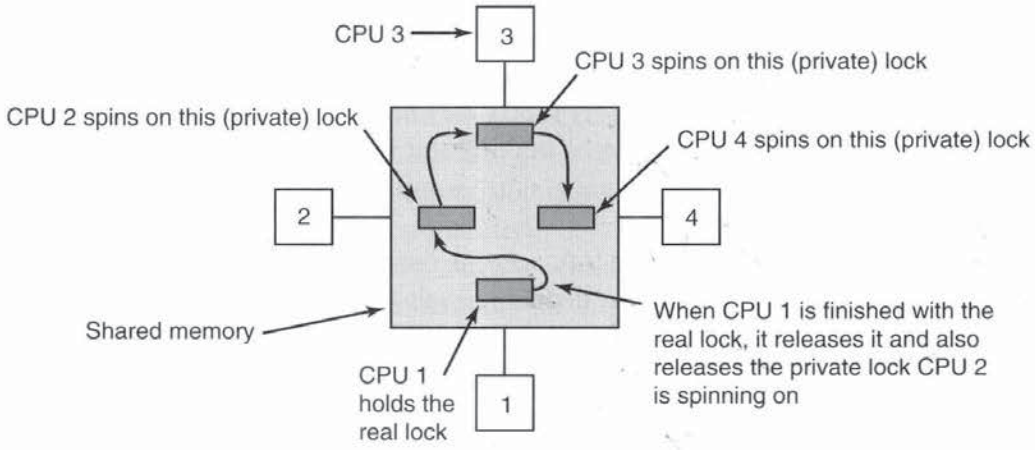


Figure 8-11. Use of multiple locks to avoid cache thrashing.

Spinning versus Switching

So far we have assumed that a CPU needing a locked mutex just waits for it, either by polling continuously, polling intermittently, or attaching itself to a list of waiting CPUs. In some cases, there is no real alternative for the requesting CPU to just waiting. For example, suppose that some CPU is idle and needs to access the shared ready list to pick a process to run. If the ready list is locked, the CPU

cannot just decide to suspend what it is doing and run another process, because doing that would require access to the ready list. It *must* wait until it can acquire the ready list.

However, in other cases, there is a choice. For example, if some thread on a CPU needs to access the file system buffer cache and that is currently locked, the CPU can decide to switch to a different thread instead of waiting. The issue of whether to spin or whether to do a thread switch has been a matter of much research, some of which will be discussed below. Note that this issue does not occur on a uniprocessor because spinning does not make much sense when there is no other CPU to release the lock. If a thread tries to acquire a lock and fails, it is always blocked to give the lock owner a chance to run and release the lock.

Assuming that spinning and doing a thread switch are both feasible options, the trade-off is as follows. Spinning wastes CPU cycles directly. Testing a lock repeatedly is not productive work. Switching, however, also wastes CPU cycles, since the current thread's state must be saved, the lock on the ready list must be acquired, a thread must be selected, its state must be loaded, and it must be started. Furthermore, the CPU cache will contain all the wrong blocks, so many expensive cache misses will occur as the new thread starts running. TLB faults are also likely. Eventually, a switch back to the original thread must take place, with more cache misses following it. The cycles spent doing these two context switches plus all the cache misses are wasted.

If it is known that mutexes are generally held for, say, 50 μsec and it takes 1 msec to switch from the current thread and 1 msec to switch back later, it is more efficient just to spin on the mutex. On the other hand, if the average mutex is held for 10 msec, it is worth the trouble of making the two context switches. The trouble is that critical regions can vary considerably in their duration, so which approach is better?

One design is to always spin. A second design is to always switch. But a third design is to make a separate decision each time a locked mutex is encountered. At the time the decision has to be made, it is not known whether it is better to spin or switch, but for any given system, it is possible to make a trace of all activity and analyze it later offline. Then it can be said in retrospect which decision was the best one and how much time was wasted in the best case. This hindsight algorithm then becomes a benchmark against which feasible algorithms can be measured.

This problem has been studied by researchers (Karlin et al., 1989; Karlin et al., 1991; and Ousterhout, 1982). Most work uses a model in which a thread failing to acquire a mutex spins for some period of time. If this threshold is exceeded, it switches. In some cases the threshold is fixed, typically the known overhead for switching to another thread and then switching back. In other cases it is dynamic, depending on the observed history of the mutex being waited on.

The best results are achieved when the system keeps track of the last few observed spin times and assumes that this one will be similar to the previous ones.

For example, assuming a 1-msec context switch time again, a thread would spin for a maximum of 2 msec, but observe how long it actually spun. If it fails to acquire a lock and sees that on the previous three runs it waited an average of 200 μsec, it should spin for 2 msec before switching. However, it if sees that it spun for the full 2 msec on each of the previous attempts, it should switch immediately and not spin at all. More details can be found in (Karlin et al., 1991).

8.1.4 Multiprocessor Scheduling

On a uniprocessor, scheduling is one dimensional. The only question that must be answered (repeatedly) is: "Which process should be run next?" On a multiprocessor, scheduling is two dimensional. The scheduler has to decide which process to run and which CPU to run it on. This extra dimension greatly complicates scheduling on multiprocessors.

Another complicating factor is that in some systems, all the processes are unrelated whereas in others they come in groups. An example of the former situation is a timesharing system in which independent users start up independent processes. The processes are unrelated and each one can be scheduled without regard to the other ones.

An example of the latter situation occurs regularly in program development environments. Large systems often consist of some number of header files containing macros, type definitions, and variable declarations that are used by the actual code files. When a header file is changed, all the code files that include it must be recompiled. The program *make* is commonly used to manage development. When *make* is invoked, it starts the compilation of only those code files that must be recompiled on account of changes to the header or code files. Object files that are still valid are not regenerated.

The original version of *make* did its work sequentially, but newer versions designed for multiprocessors can start up all the compilations at once. If 10 compilations are needed, it does not make sense to schedule 9 of them quickly and leave the last one until much later since the user will not perceive the work as completed until the last one finishes. In this case it makes sense to regard the processes as a group and to take that into account when scheduling them.

Timesharing

Let us first address the case of scheduling independent processes; later we will consider how to schedule related processes. The simplest scheduling algorithm for dealing with unrelated processes (or threads) is to have a single system-wide data structure for ready processes, possibly just a list, but more likely a set of lists for processes at different priorities as depicted in Fig. 8-12(a). Here the 16 CPUs are all currently busy, and a prioritized set of 14 processes are waiting to run. The first CPU to finish its current work (or have its process block) is CPU 4,