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

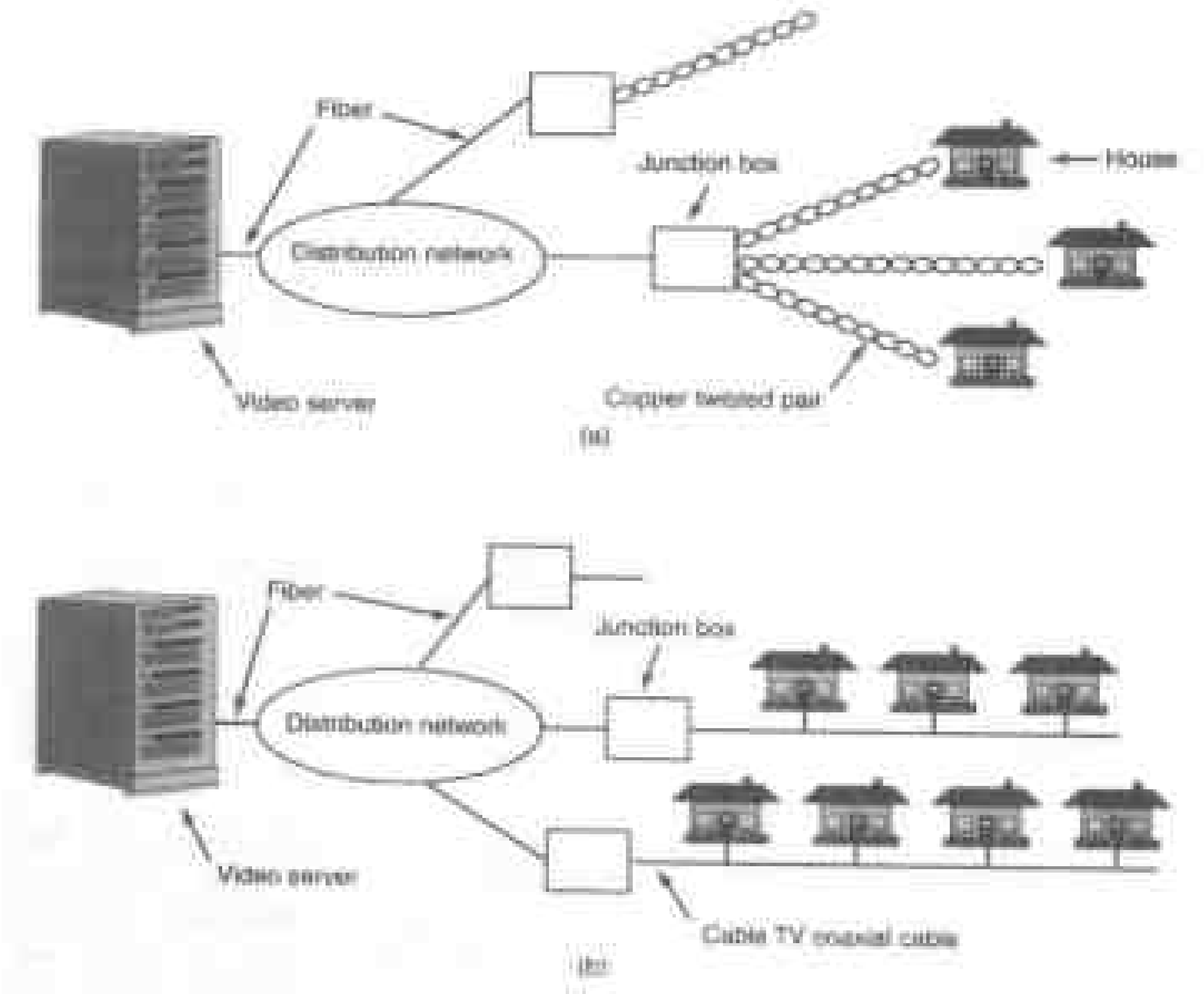

Figure 7-1. Video on demand using different local distribution technologies. (a) ADSL. (b) Cable TV.

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.

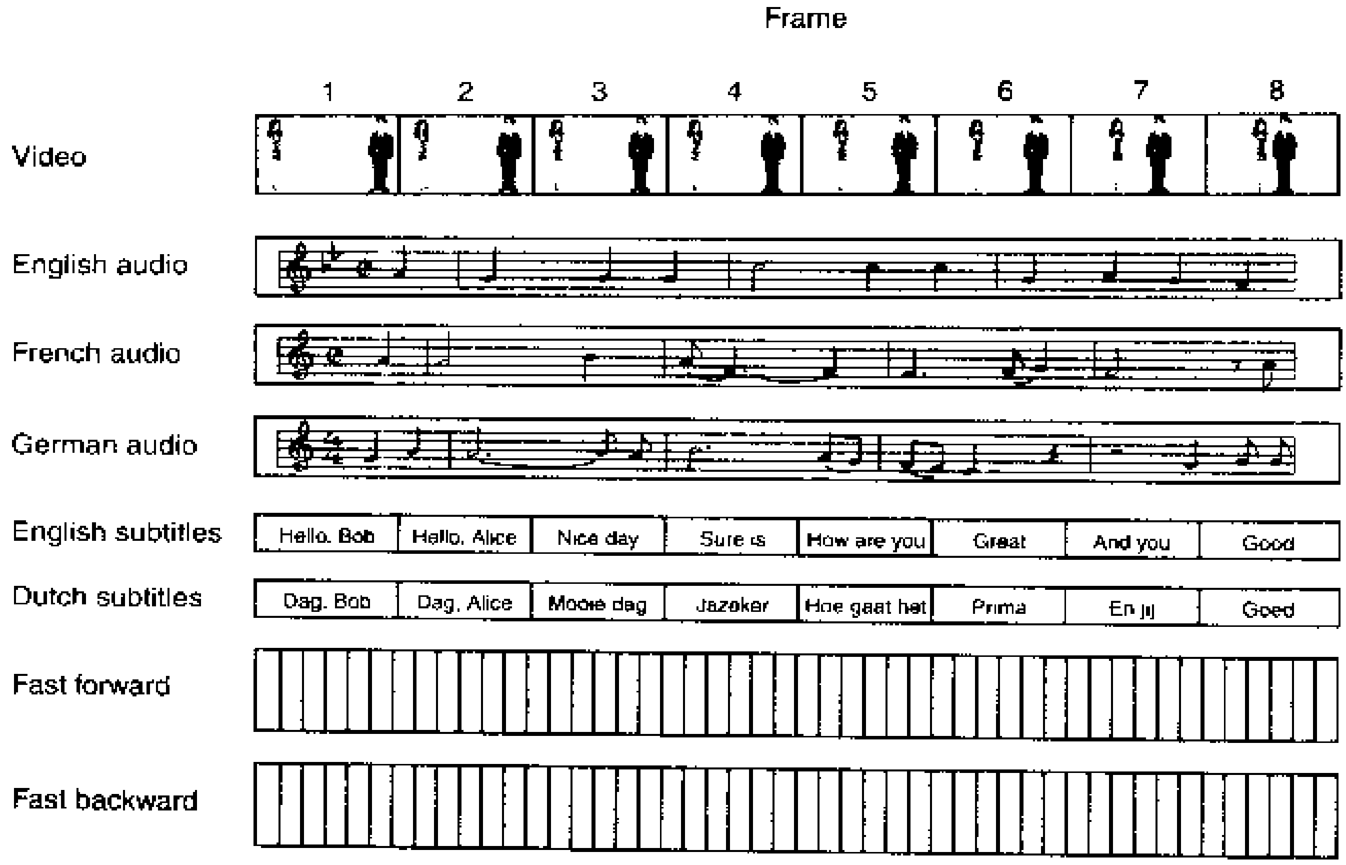

Figure 7-3. A movie may consist of several files.

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

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

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.

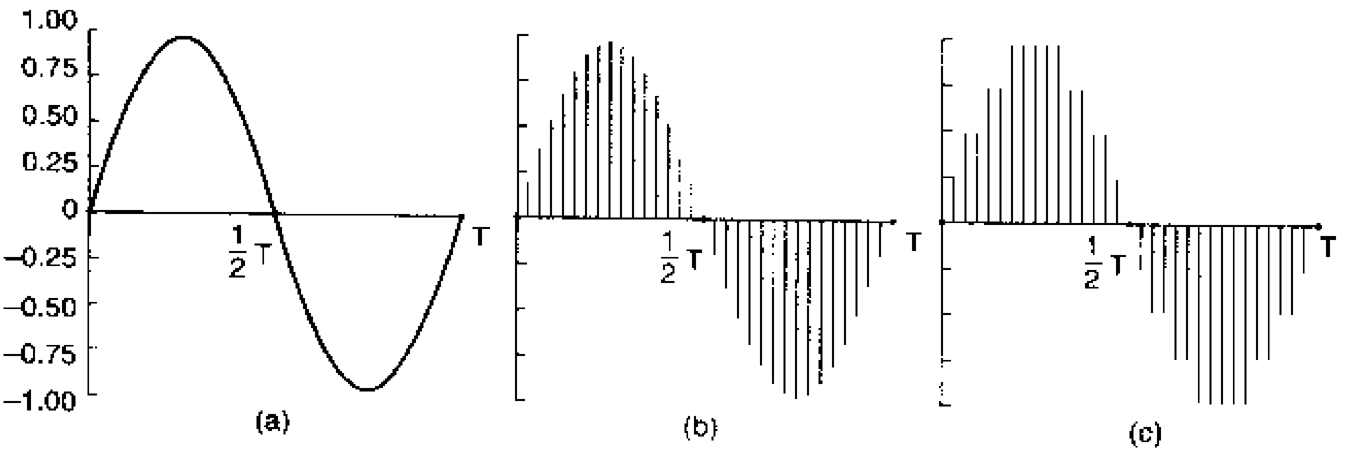

Figure 7-4. (a) A sine wave. (b) Sampling the sine wave. (c) Quantizing the samples to 4 bits.

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

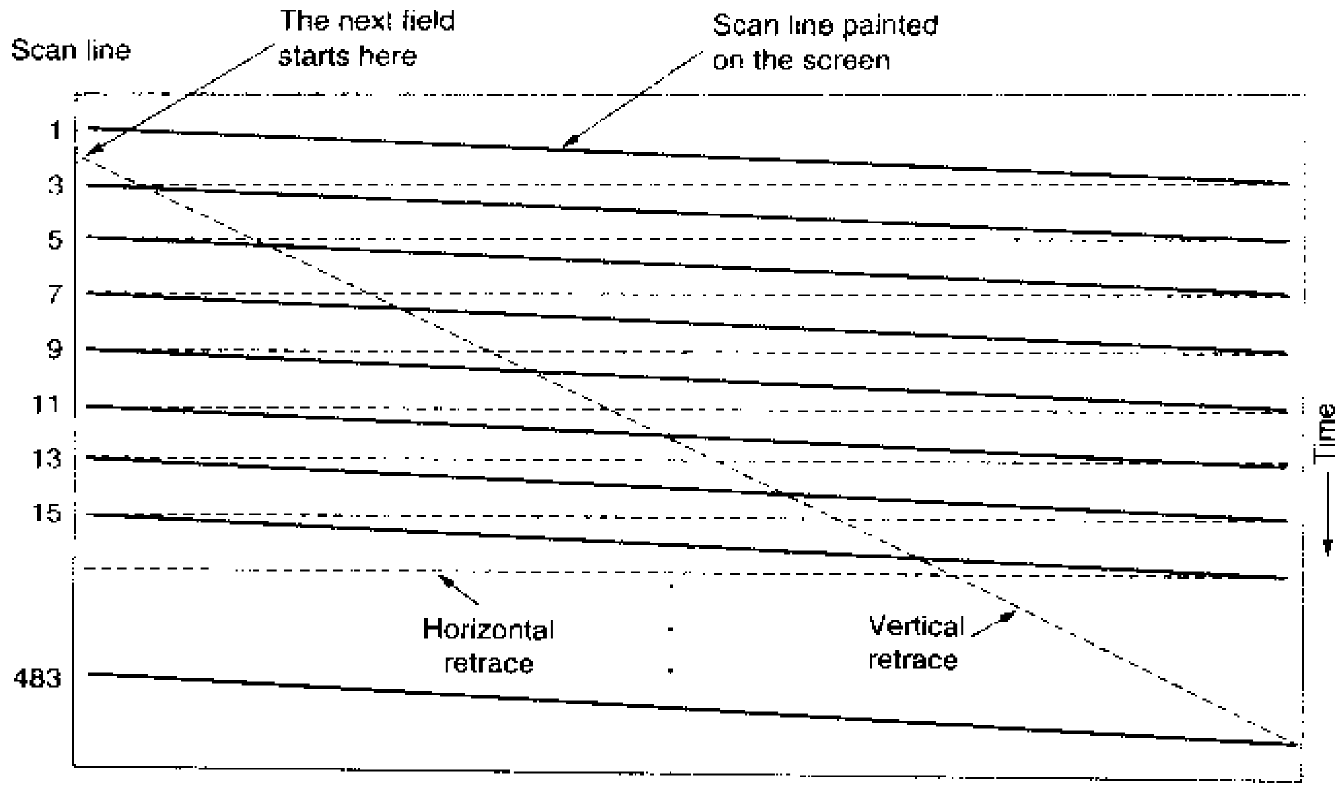

Figure 7-5. The scanning pattern used for NTSC video and television.

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

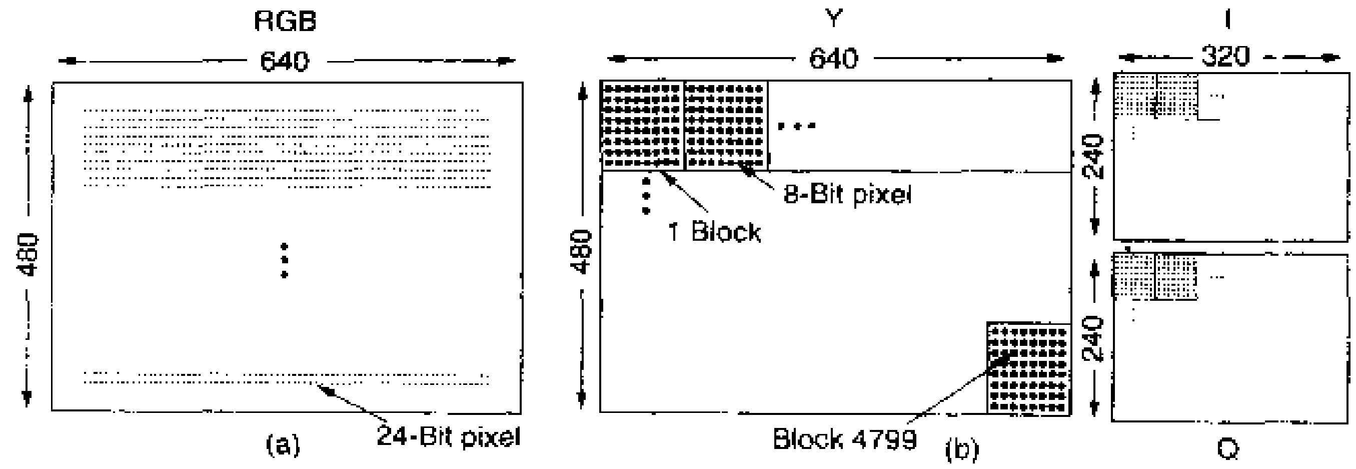

Figure 7-6. (a) RGB input data. (b) After block preparation.

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).

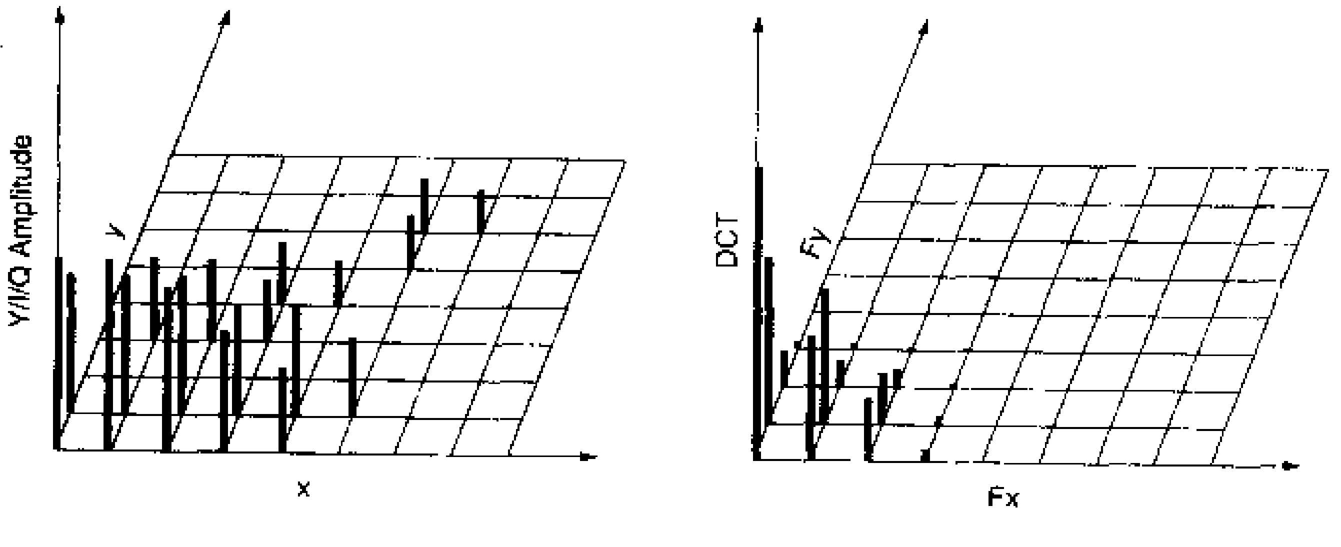

Figure 7-7. (a) One block of the *Y* matrix. (b) The DCT coefficients.

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.

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

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.

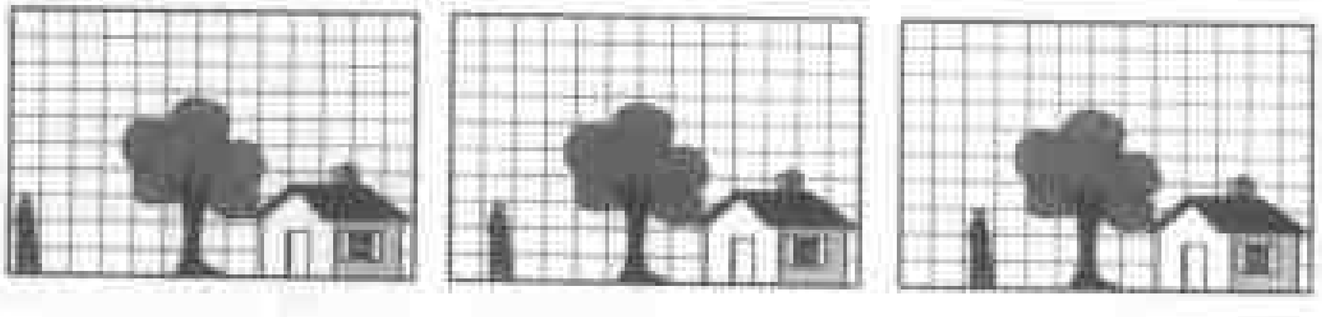

Figure 7-10. Three consecutive video frames.

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.

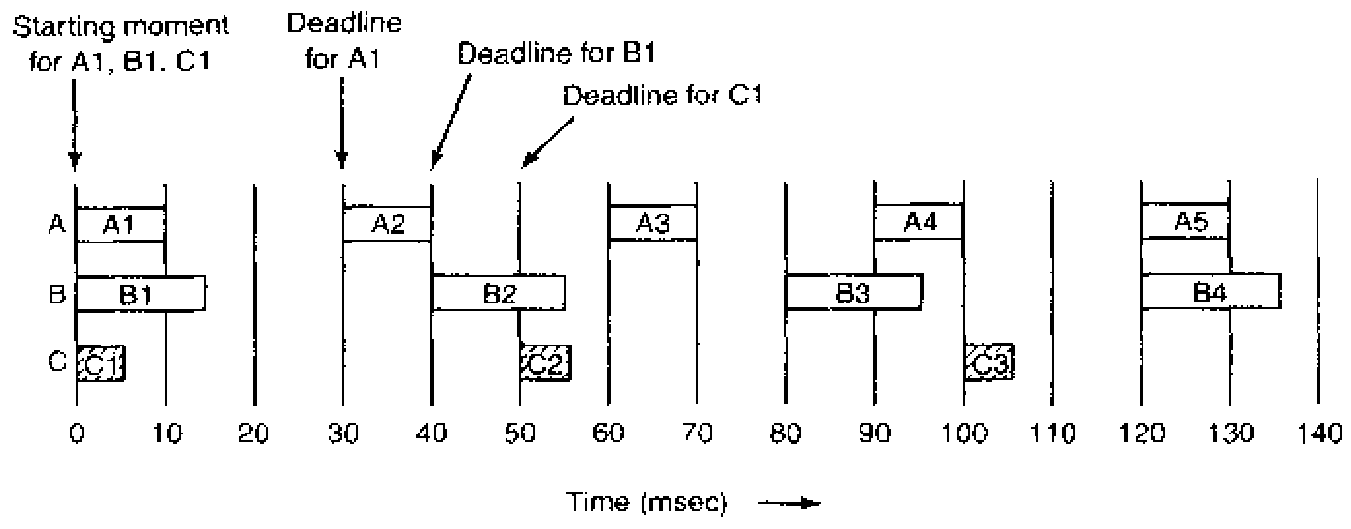

Figure 7-11. Three periodic processes, each displaying a movie. The frame rates and processing requirements per frame are different for each movie.

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

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

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 $t = 70$.

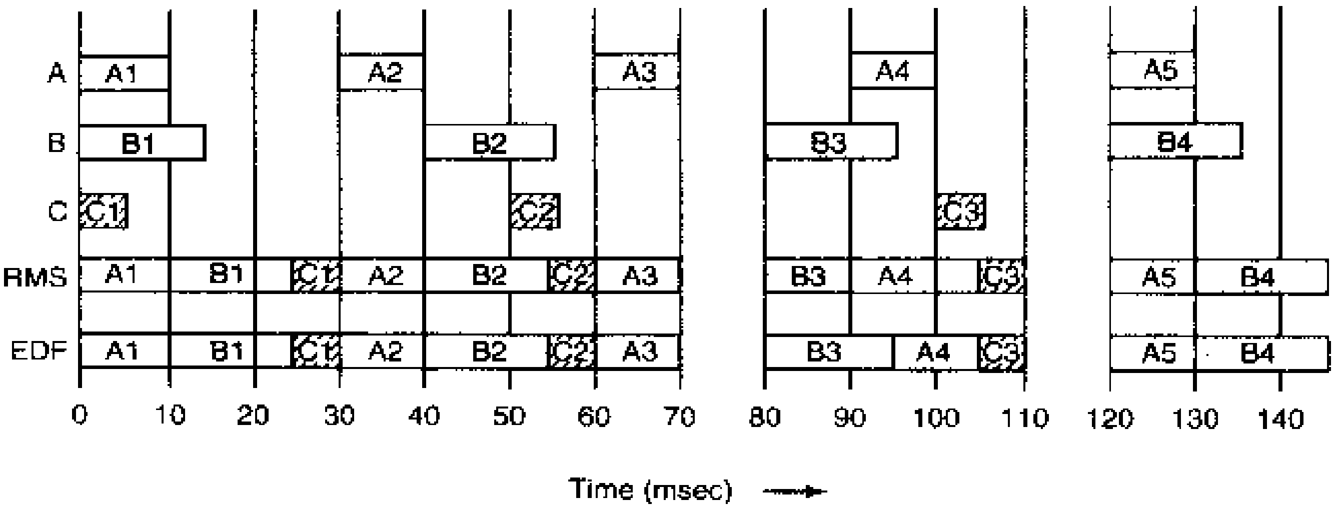

Figure 7-12. An example of RMS and EDF real-time scheduling.

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.

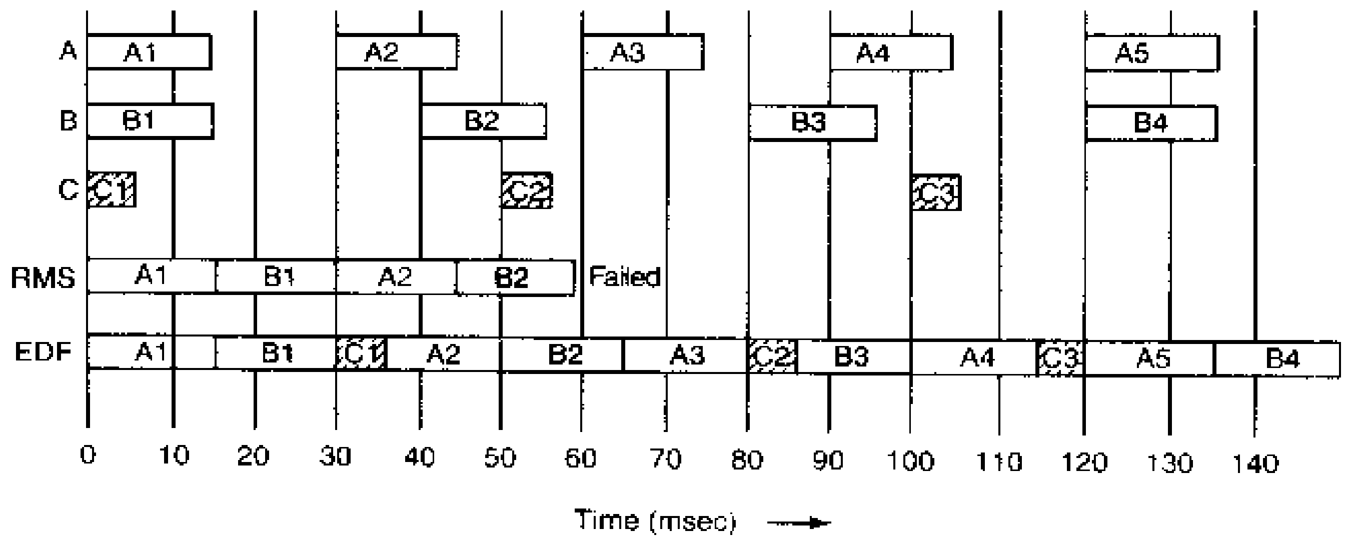

Figure 7-13. Another example of real-time scheduling with RMS and EDF.

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

$$\sum_{i=1}^{m} \frac{C_i}{P_i} \le m(2^{1/m} - 1)$$

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 $m \to \infty$, 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.

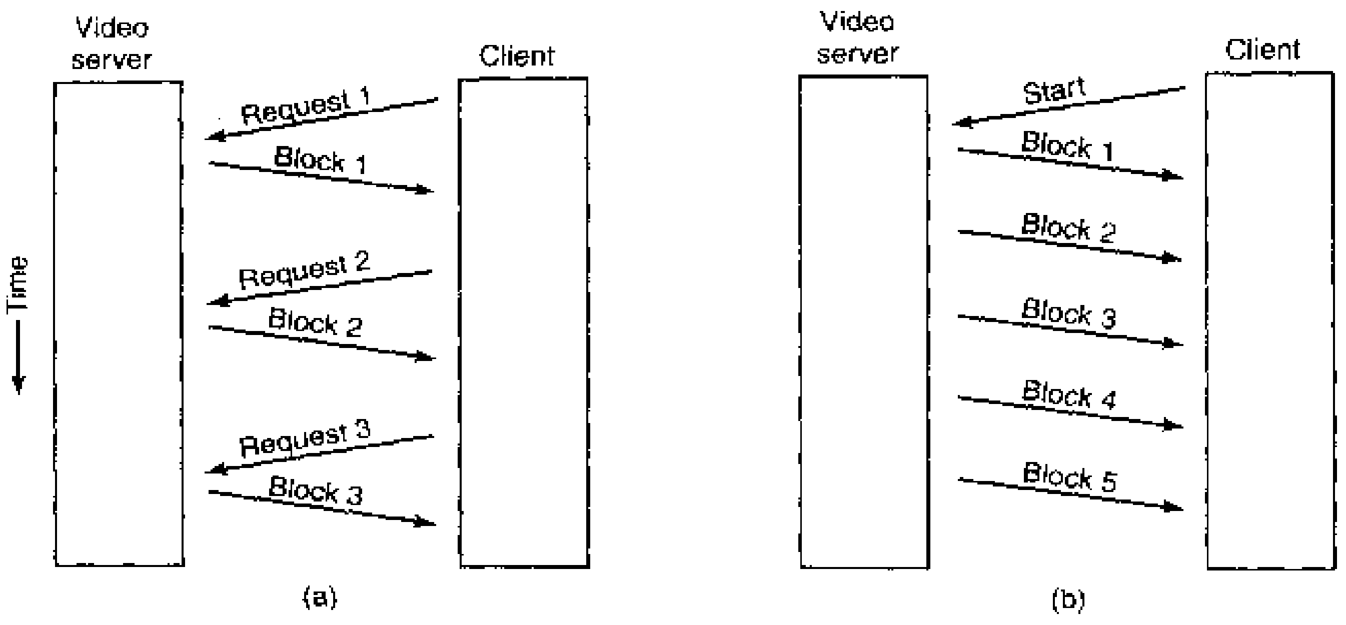

Figure 7-14. (a) A pull server. (b) A push server.

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

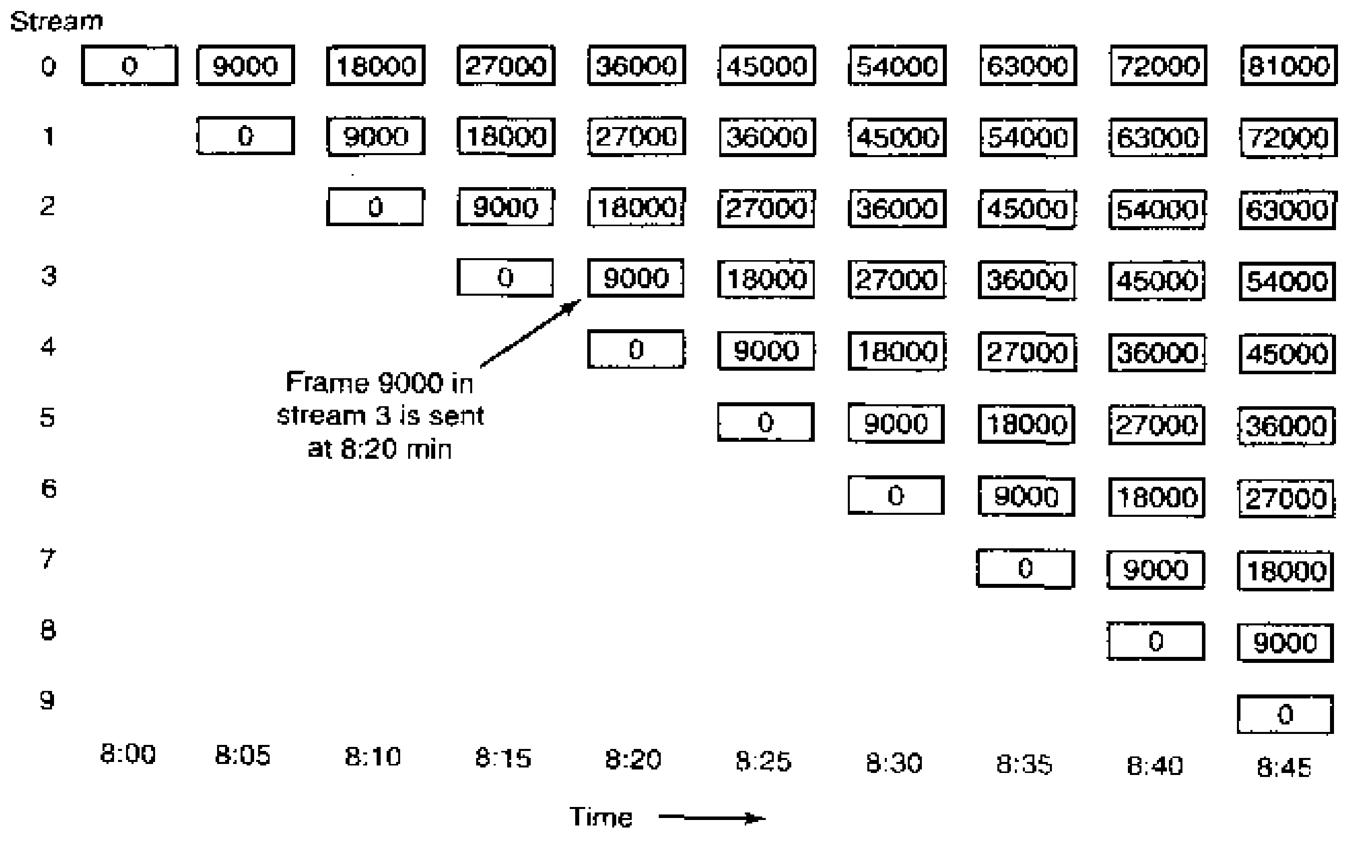

Figure 7-15. Near video on demand has a new stream starting at regular intervals, in this example every 5 minutes (9000 frames).

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.

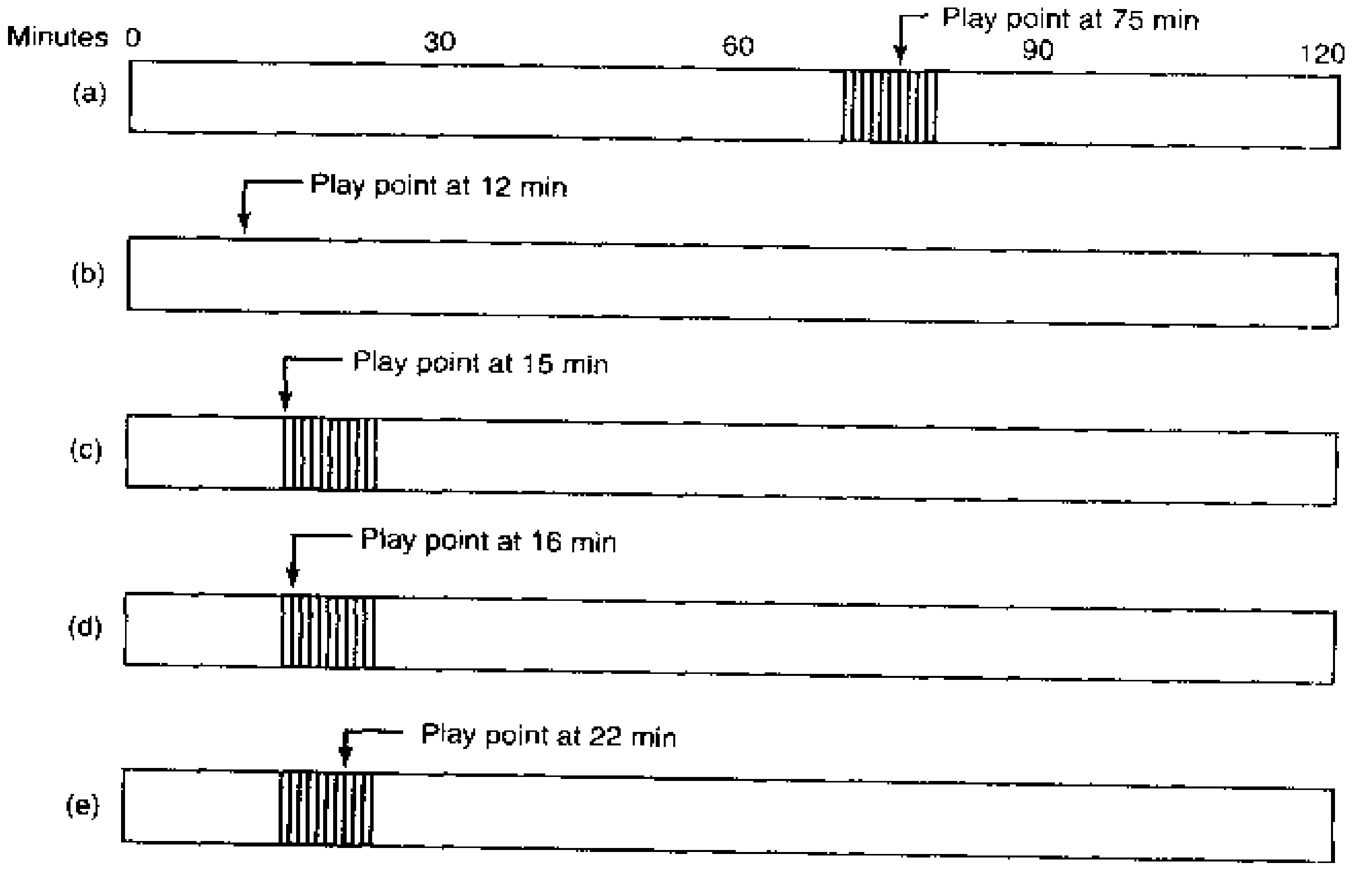

Figure 7-16. (a) Initial situation. (b) After a rewind to 12 min. (c) After waiting 3 min. (d) After starting to refill the buffer. (e) Buffer full.

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.

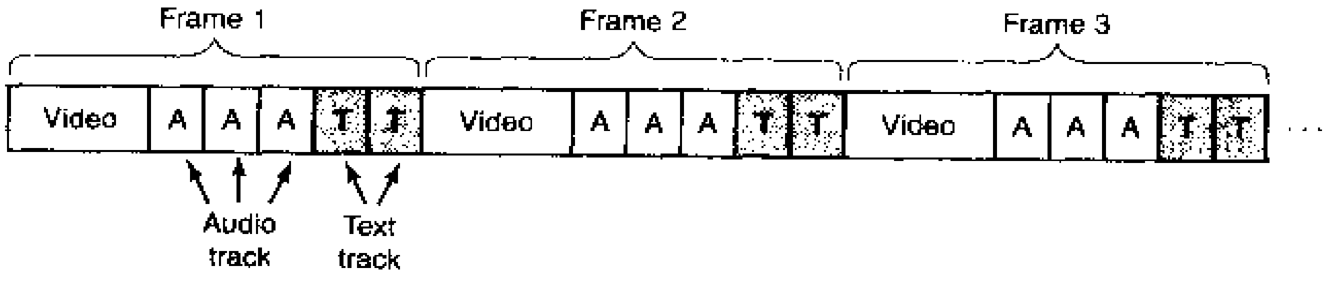

Figure 7-17. Interleaving video, audio, and text in a single contiguous file per movie.

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.

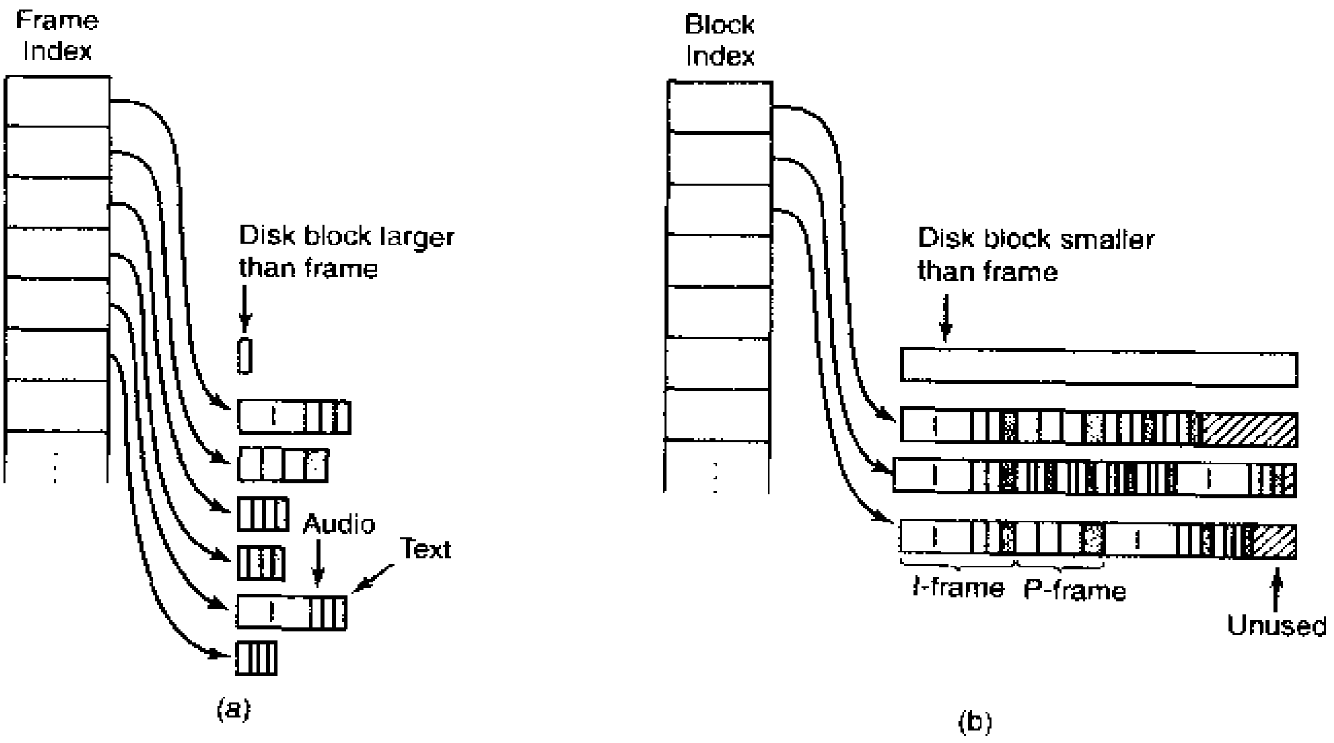

Figure 7-18. Noncontiguous movie storage. (a) Small disk blocks. (b) Large disk blocks.

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.

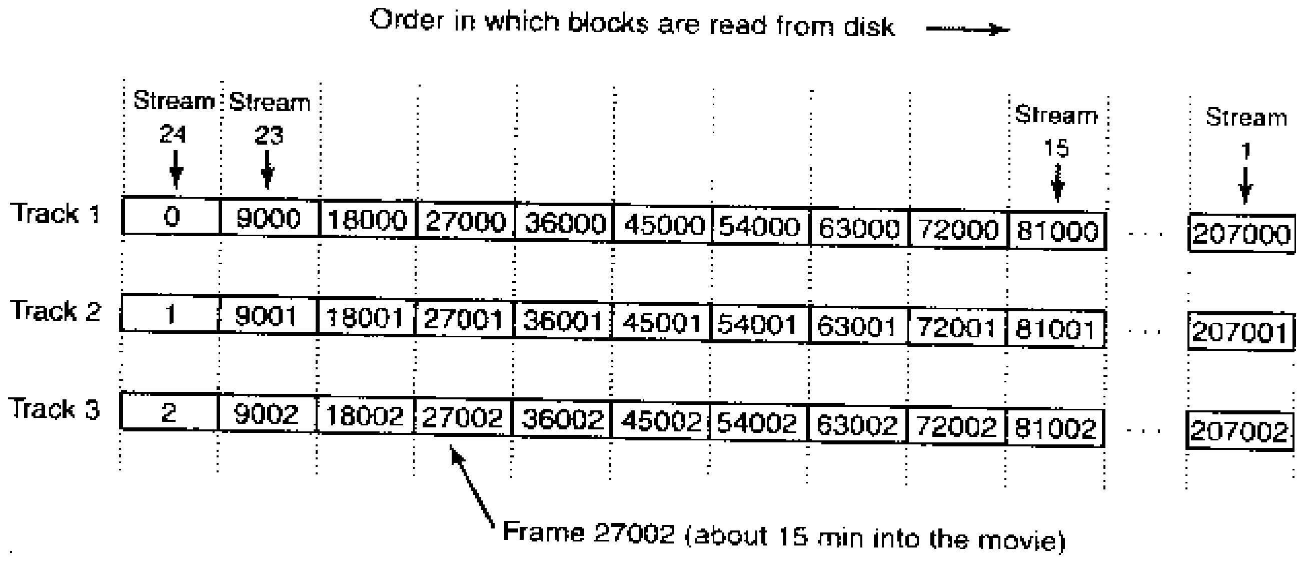

Figure 7-19. Optimal frame placement for near video on demand.

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

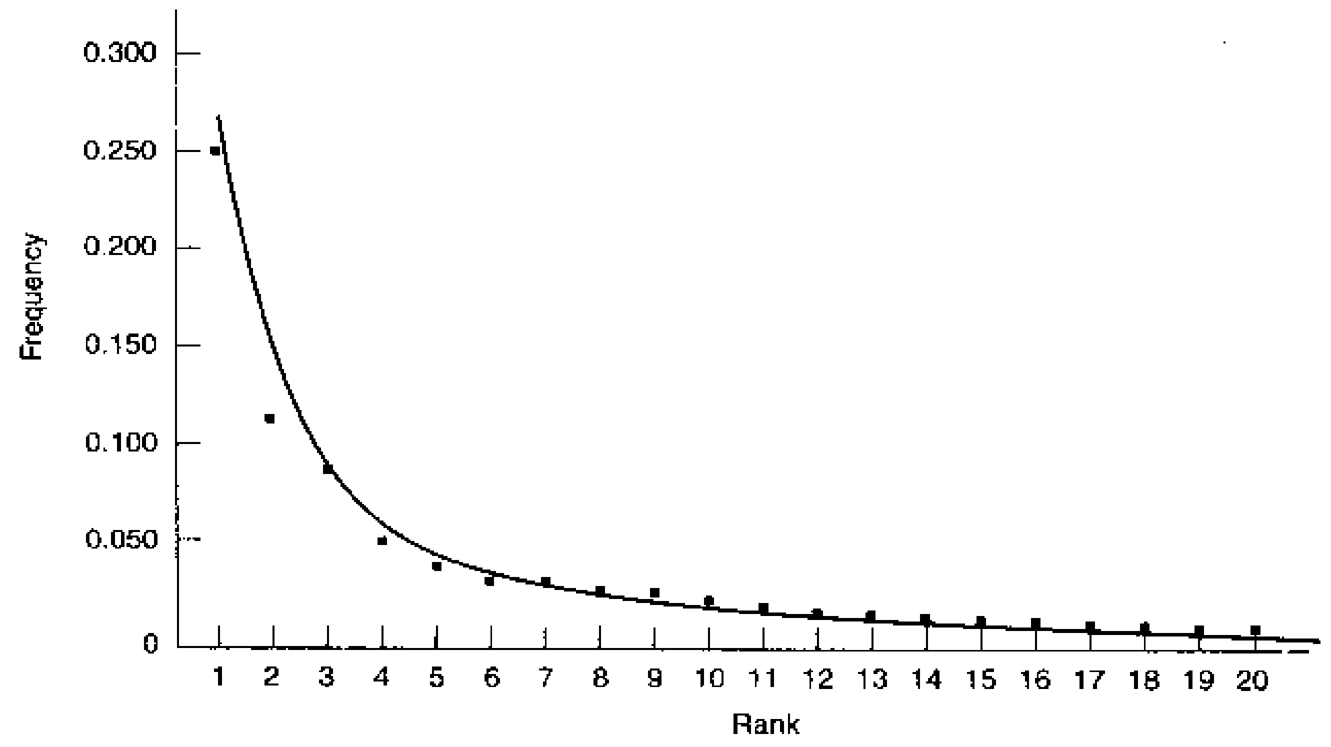

Figure 7-20. The curve gives Zipf's law for $N = 20$. The squares represent the populations of the 20 largest cities in the U.S., sorted on rank order (New York is 1, Los Angeles is 2, Chicago is 3, etc.).

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.

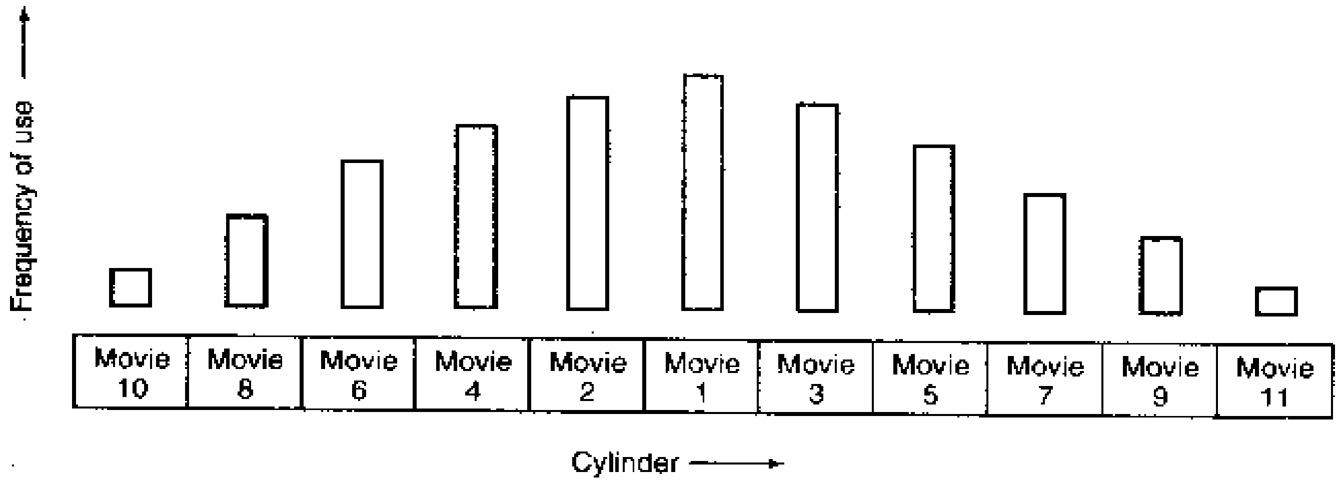

Figure 7-21. The organ-pipe distribution of files on a video server

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.

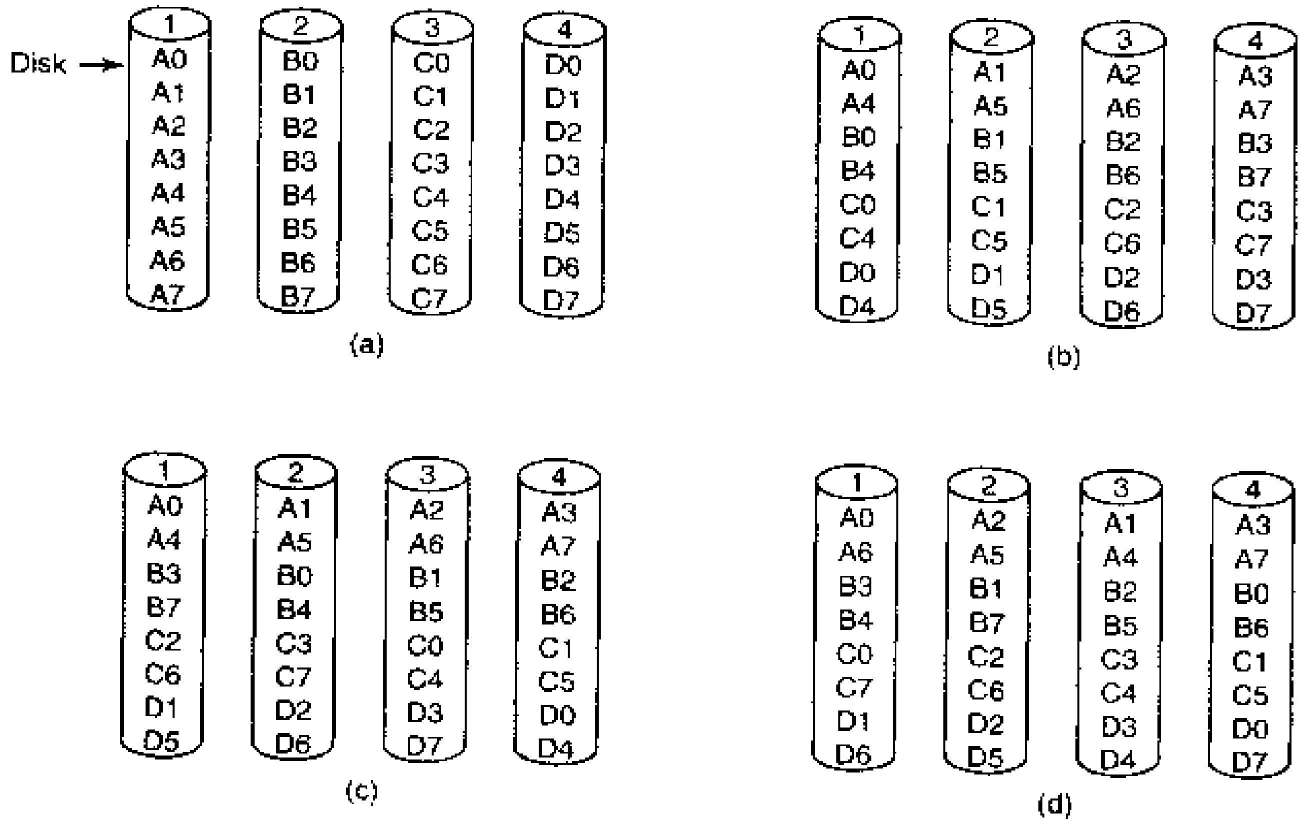
Disk
1 A0 A1 A2 A3 A4 A5 A6 A7
2 B0 B1 B2 B3 B4 B5 B6 B7
3 C0 C1 C2 C3 C4 C5 C6 C7
4 D0 D1 D2 D3 D4 D5 D6 D7
(a)
1 A0 A4 B0 B4 C0 C4 D0 D4
2 A1 A5 B1 B5 C1 C5 D1 D5
3 A2 A6 B2 B6 C2 C6 D2 D6
4 A3 A7 B3 B7 C3 C7 D3 D7
(b)
1 A0 A4 B3 B7 C2 C6 D1 D5
2 A1 A5 B0 B4 C3 C7 D2 D6
3 A2 A6 B1 B5 C0 C4 D3 D7
4 A3 A7 B2 B6 C1 C5 D0 D4
(c)
1 A0 A6 B3 B4 C0 C7 D1 D6
2 A2 A5 B1 B7 C2 C6 D2 D5
3 A1 A4 B2 B5 C3 C4 D3 D4
4 A3 A7 B0 B6 C1 C5 D0 D7
(d)

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.

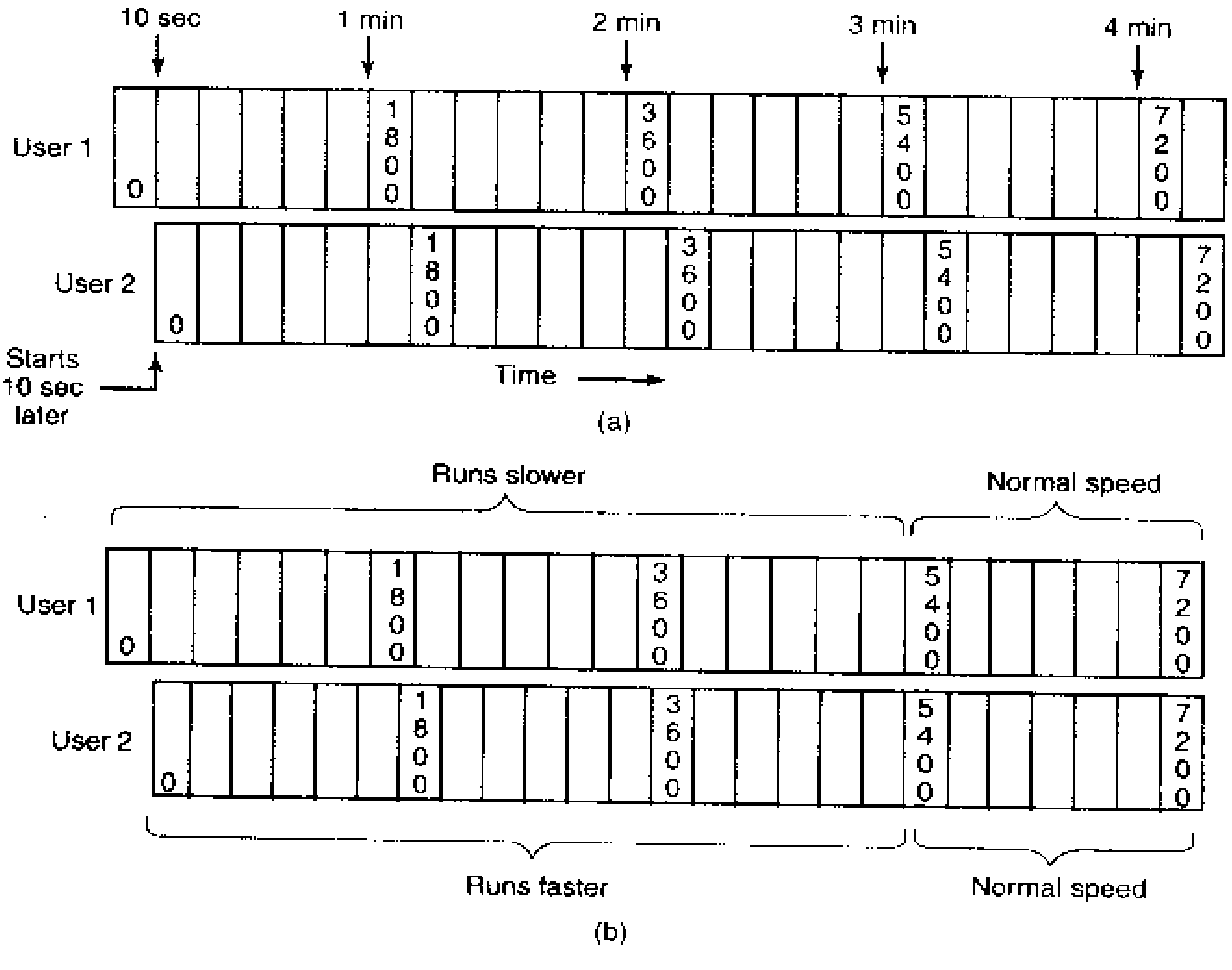

Figure 7-23. (a) Two users watching the same movie 10 sec out of sync. (b) Merging the two streams into one.

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.

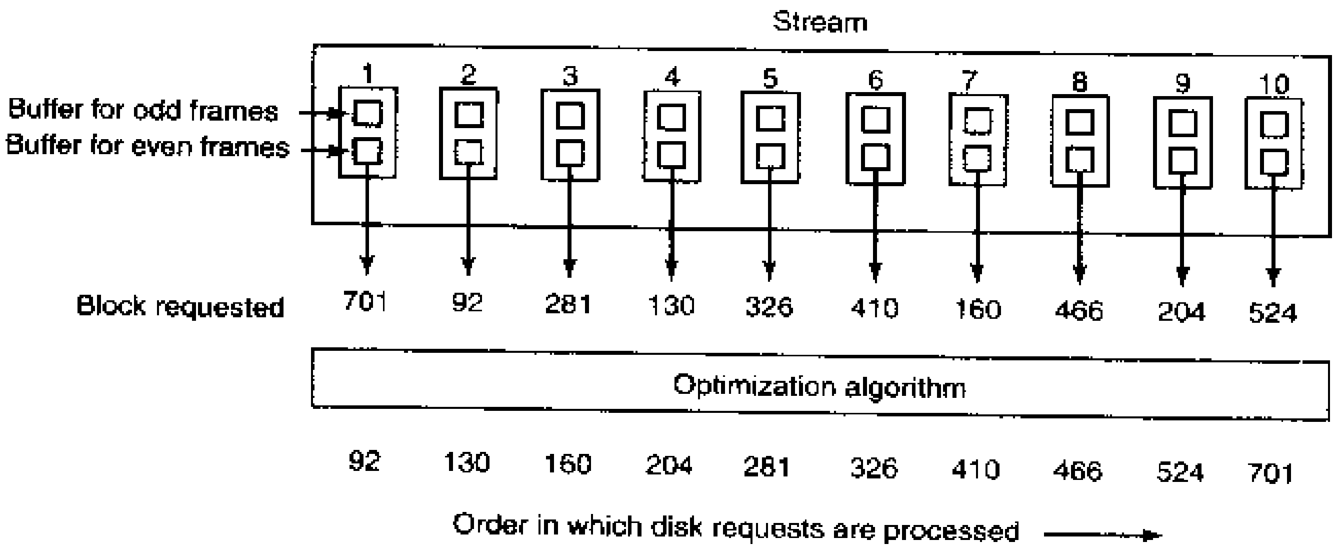

Figure 7-24. In one round, each movie asks for one frame.

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

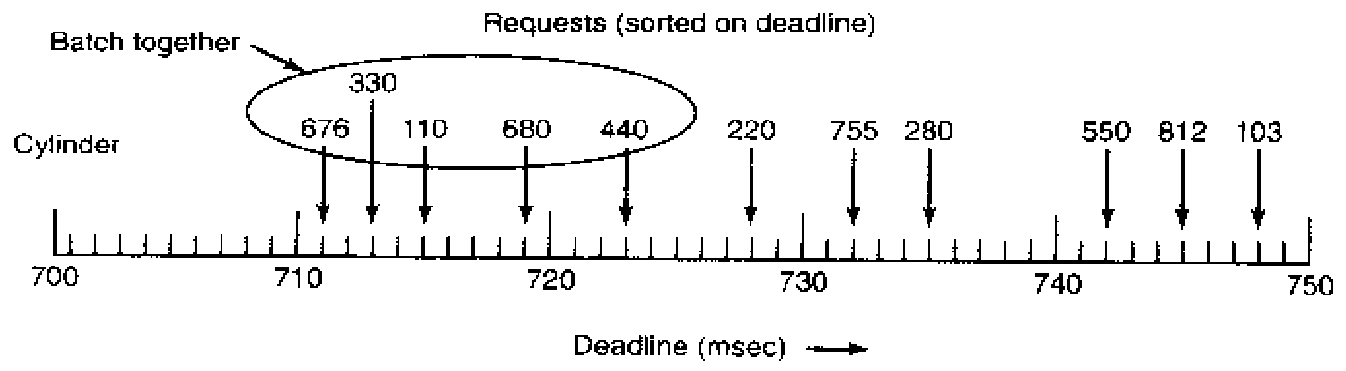

Figure 7-25. The scan-EDF algorithm uses deadlines and cylinder numbers for scheduling.

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 videoconferencing) 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.