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. 7-1.

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

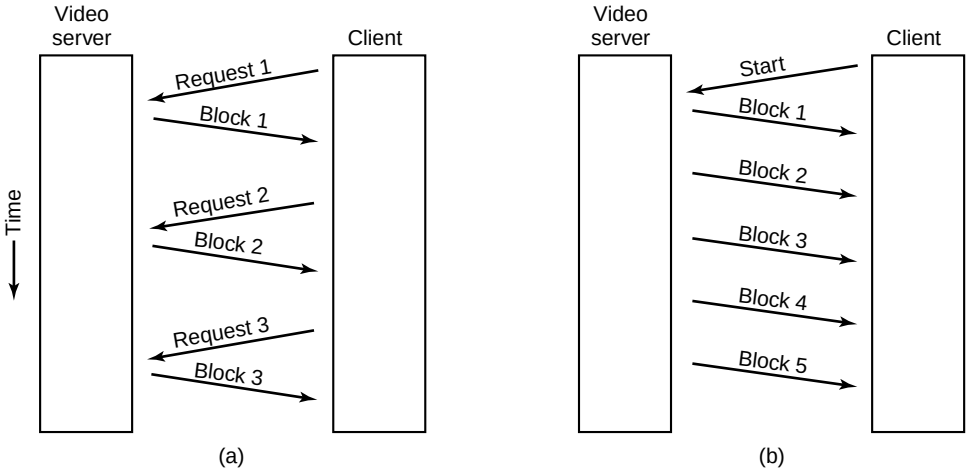

Figure 7-1. (a) A pull server. (b) A push server.

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. 7-0 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. 7-2, 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.

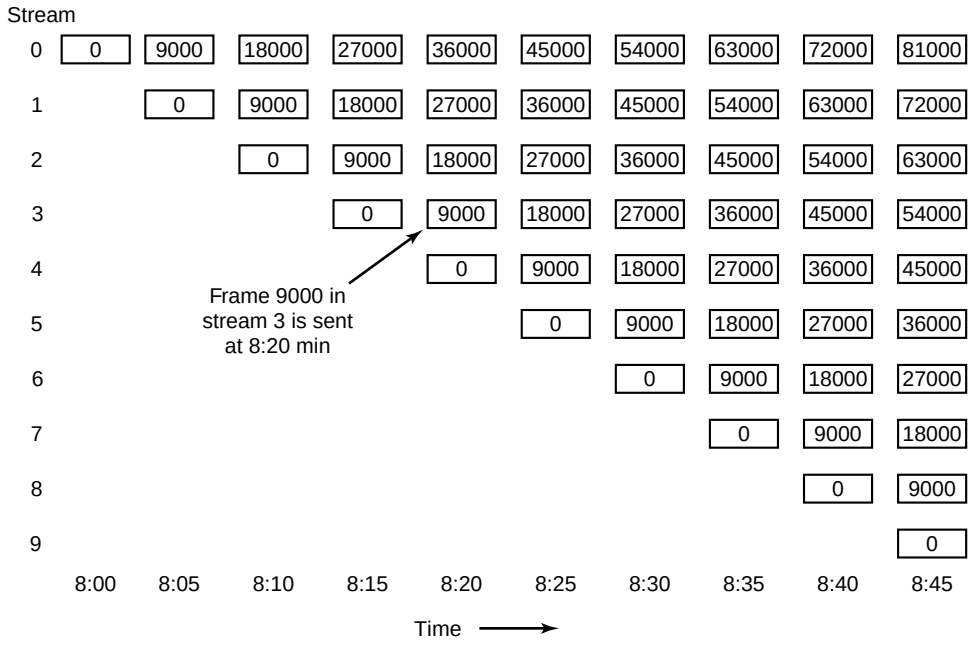

Figure 7-2. Near video on demand has a new stream starting at regular intervals, in this example every 5 minutes (9000 frames).

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 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. 7-2. 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. 7-3(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.

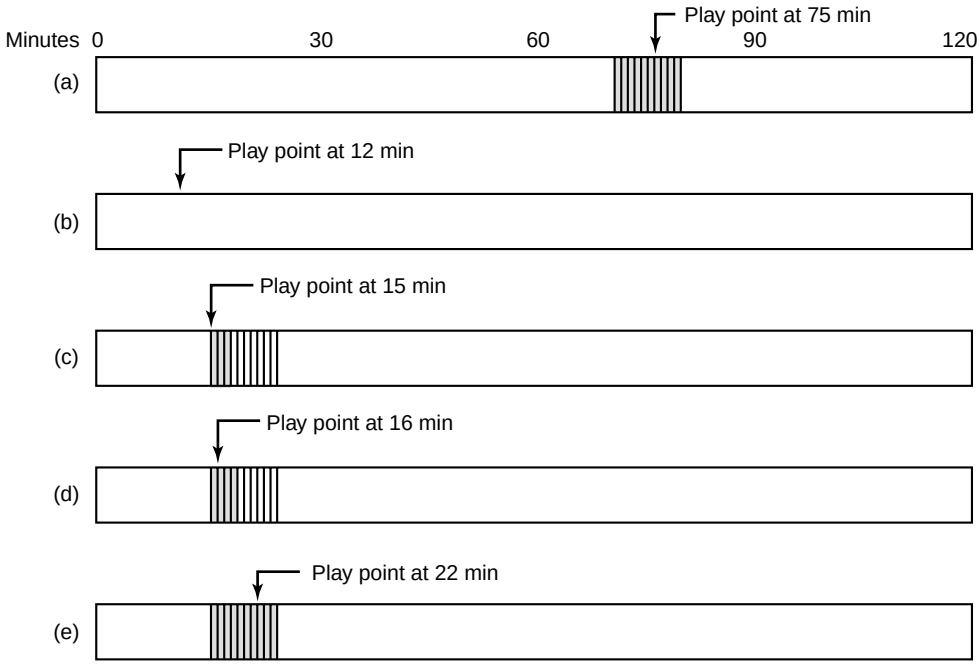

Figure 7-3. (a) Initial situation. (b) After a rewind to 12 min. (c) After waiting 3 min. (d) After starting to refill the buffer. (e) Buffer full.

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. 7-3(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. 7-3(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. 7-3(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.