Multimedia Synchronization and UNIX —or— If Multimedia Support is the Problem, Is UNIX the Solution?

Dick C.A. Bulterman
Guido van Rossum
Dik Winter
CWI: Centrum voor Wiskunde en Informatica
Amsterdam, The Netherlands

Abstract

This paper considers the role of UNIX in supporting multimedia applications. In particular, we consider the ability of the UNIX operating system (in general) and the UNIX I/O system (in particular) to support the synchronization of a number of high-bandwidth data sets that must be combined to support generalized multimedia systems. The paper is divided into three main sections. The first section reviews the requirements and characteristics that are inherent to multimedia applications. The second section reviews the facilities provided by UNIX and the UNIX I/O model. The third section contrasts the needs of multimedia and the abilities of UNIX to support these needs, with special attention paid to UNIX's problem aspects. We close by sketching an approach we are studying to solve the multimedia processing problem: the use of a distributed operating system to provide a separate data and processing management layer for multimedia information.

Keywords: Multimedia, UNIX kernel organization, I/O systems, distributed operating systems.

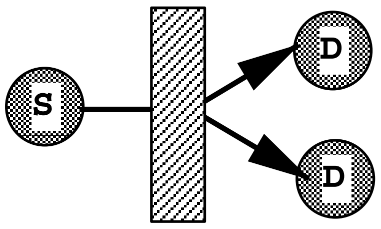
1. Introduction

If a definition for ''Multimedia'' were to exist in an ultra-modern dictionary of computer jargon, the entry might read as follows:

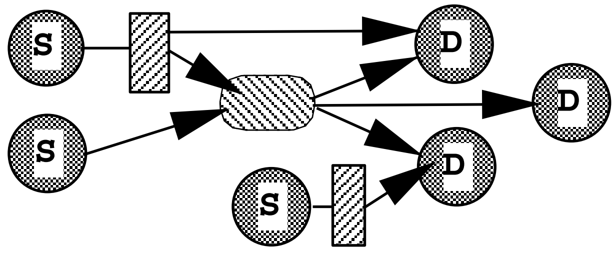
> **mul-ti-me-di-a** <*buzzword; adj.*>: A property of applications software that allows for the mixed use of several (chiefly output) media—such as sound, video, text, image and graphic data—in a manner that makes an unsuspecting user think that something extraordinary is happening on an otherwise conventional computing system.
> *Examples*: multimedia mail, multimedia documents, multimedia research.
> **See also**: *Clothes, Emperor's New*.

While serious multimedia researchers (such as the authors) would take exception to the tone of this definition, few would argue against the description of multimedia systems as consisting of applications software that uses mixed forms of media as a basis for presenting information. Unfortunately, even this definition has a serious flaw: it places an emphasis on *applications*; this obscures the crucial role that other layers in a computing system play in providing multimedia support.

Support for multimedia data manipulation can exist at three levels: in the applications code that allows the user to access and control the flow of information, in the operating systems code through the use of device drivers and scheduling software, or at the hardware controller level. Activity at the controller level is currently constrained to providing or accepting data under control of the other layers in the system. The partitioning of activities between the operating system and the applications layer depends on the complexity of the system being supported. (See figure 1.) In simple multimedia systems (fig.1a), data consists of raw information that must be moved from one place to another at a specified rate. This is FAX-style multimedia: the information itself is uninterpreted, making content-based manipulation of the data impossible. In this case, the application layer may request that the operating system start a transfer, with most subsequent device actions controlled by a (set of) device drivers. In less simple forms of multimedia systems (fig. 1b), each of the various data paths may consist of structured information that can be manipulated individually or as a whole, and in which the *meaning* of the information may influence its processing. While the characteristics of multimedia systems are as different as the number of separate applications systems, we can generally conclude that the more manipulation of information that is required, the higher the degree of applications support that will be necessary in the system. (See [1] for a discussion of multimedia and workstations.)



(a)



(b)

Figure 1: Two examples of multimedia data processing.

At first glance, multimedia support seems to be nothing more than providing UNIX I/O drivers for a set of new device controllers for a variety of media types. The most fundamental difference between multimedia systems and other types of input/output transformations of data, however, is the fact that multimedia data is inherently multi-dimensional. This means multimedia information consists of a number of *related* components that must be gathered and/or scattered to/from a variety of devices, under a set of implicit or explicit synchronization constraints. This synchronization needs to be supported by a system layer that has access to the underlying hardware (for integrating device I/O) and is accessible to the application code (which ultimately controls the logical information flow). The synchronization also needs to be supported by the general system-wide process scheduling mechanism to ensure that all of the components at all levels are active at the desired times.

In this paper, we consider the role of the UNIX kernel in providing multimedia support. Our purpose is to consider the needs of multimedia systems and to contrast these against the facilities provided in UNIX to support these needs. While several aspects of multimedia data manipulation are considered, we focus primarily on the synchronization of independent data paths (such as separate audio and image paths) within a multimedia application. We do this because, unlike application-specific user programs or media-specific hardware controllers, it is the operating system that currently provides the only programmable place for providing the efficient synchronization primitives that both the hardware and applications software need to allow complex multimedia data interaction.

We start our consideration of the relationship between multimedia support and operating systems by discussing two classes of multimedia data models: multimedia data location models and multimedia data synchronization models. We then review the location and synchronization facilities provided by the UNIX I/O model. We then compare the needs and expressive capabilities of both UNIX and generic multimedia systems to see if UNIX is ''multimedia ready''. We conclude by offering an alternative approach that we are studying as part of the CWI/Multimedia project [2,3] to better support broad classes of multimedia applications. Note that while our general observations on the utility of UNIX for long-term multimedia support are not positive, we hope to provide more than an exercise in UNIX-bashing; instead, we hope to highlight a set of problems that we feel will increase in importance as the use of multimedia expands.

2. Two Classes of Models for Data Interaction in Multimedia Systems

The broadest notion of the term multimedia is simply the use of several different types of data formats to encode and/or present information. In considering methods of supporting multimedia, two issues immediately arise: first, it is important to know where the multimedia data comes from (as well as knowing where it will need to be sent), and second, it is important to consider which types of synchronization is required to ensure that the multiple data streams interact in the desired manner. The first issue deals with data location models; these models determine the sources and destinations of data streams.[1] The second issue deals with the relationships among data streams; these relationships can be the property of an application or of the data itself. Both of these issues are considered in the following sections.

2.1. Multimedia Data Location Models

The location of multimedia information determines the amount of operating systems support that are required to gather scattered data for possible processing, and to pass that data on to a set of output devices. There are four general models that can describe the sources and destinations for multimedia information. These are reviewed in figure 2.

- *Local Single Source:* this location model has all data originating at a single source. An example is a CD-ROM that contains sound, text and picture information. Information is fetched in blocks from the source device and then routed (by either the device controller, the operating system kernel or the application) to one or more output devices. The primary attraction of the single source model is that all synchronization among input media is the responsibility of the source material designer. This media is typically interleaved into a single sequential stream that is fetched using conventional system calls.

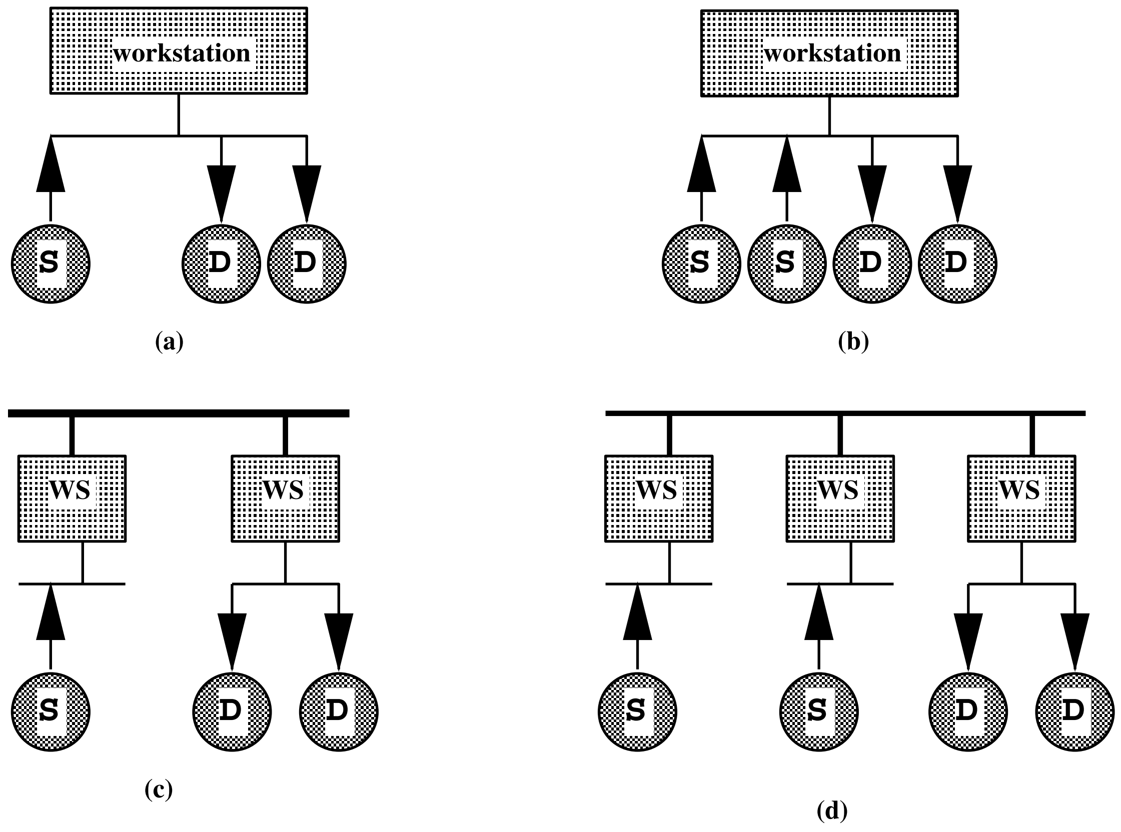


Figure 2: Location of multimedia data: (a) local single source, (b) local multiple source, (c) single distributed source, (d) multiple distributed source.

[1] Note that *streams* is used here in terms of a collection of information, not in terms of a particular device driver implementation technique.

- *Local Multiple Sources:* this model appears similar to that defined above, except that source data is scattered across several devices. (We again assume that output data goes to several devices as well.) An example of this type of interaction is combining voice annotation with images in an electronic slide-show. The principal difference between single source and multiple source data is the need for some sort of external synchronization between the data streams. The location of this synchronization (either in user code or in the kernel) will depend on the performance needs of the application, with a general trade-off existing between better performance (more towards the kernel) and greater flexibility (more towards the user).
- *Distributed Single Source:* in this model, we assume that a single source of information exists that is located on a remote workstation. The single-source nature of data means that no multi-stream synchronization is necessary. The difference between local and distributed models is that some account needs to be made for the transfer delays between source and destination. These delays may result in portions of the composite material arriving at non-constant rates or out of order. Control over the data is spread over at least two kernels (the sending and receiving) as well as several protocol layers and a user layer.
- *Distributed Multiple Sources:* this is the most general model of data location. Information may be gathered from many sources on many workstations, and destinations may also be spread over several places. This model is the most interesting because it combines aspects of synchronization problems with transfer delays and raw information scheduling. While current network technologies limit short-term practical applications of this model, we expect that this model will become much more viable within five years time.

The central problem in supporting multimedia data is, then, the degree to which processing layers need to exist between the source of information and its destination(s). In general, the more flexibility required (whether in terms of number of streams or amount of post-fetch processing) the greater the delay. For simple operating systems (such as single-user, single CPU environments), the delays can be easily predicted for a given application. For multi-processing workstations, the coordination and scheduling tasks become much greater. In both cases, the level of complexity is directly related to the amount of synchronization processing required by an application.

2.2. Multimedia Data Synchronization Models

Regardless of the location of data, the data itself can contain synchronization information (that is, it can be self-synchronizing) or it may require synchronization through an external mechanism. Synchronization concerns cover a broad spectrum. In this section, we consider four aspects that affect the partitioning of tasks among the application software, the OS and the device controller(s). These are: the basic type of relationship among data streams, the scope of synchronization information, the determination of the controlling party in a synchronization relationship, and issues regarding the precision of synchronization required.

- *Synchronization Classes:* there are two basic classes of synchronization within a multimedia framework: *serial* synchronization and *parallel* synchronization. Serial synchronization requirements determine the rate at which events must occur within a single data stream; this includes the rate at which sound information is processed, or video information is fetched, etc. Parallel synchronization requirements determine the relative scheduling of separate synchronization streams. In most non-trivial multimedia applications, each stream will have a serial synchronization requirement and a parallel relationship with other streams. Note that a special case of serial synchronization can be defined as *composite* or *embedded* synchronization; in this case, each serial block of data contains information for parallel output streams. In this case, the parallel synchronization among blocks is embedded in a serial stream.
- *Synchronization Scope:* the second distinction is between point and continuous synchronization. Point synchronization requires only that a single point of one block coincides with a single point of another. Continuous synchronization requires close synchronization of two events of longer duration. In general, point synchronization can be managed by the applications layer while continuous synchronization will need to be managed by a device controller or a high-performance, low-overhead portion of the operating system.
- *Synchronization Masters:* the third distinction regards the controlling entity in a (set of) stream(s). Sometimes we have two channels that are equally important, but sometimes one channel is the

‘‘master’’ and the other the ‘‘slave’’. It is also possible that an external clock plays the role of the master, either for all of the streams or for a subset of time-critical ones.

- *Synchronization Precision:* Finally, there are levels of precision. Stereo sound channels must be synchronized very closely (within 1 to 0.1 millisecond), because perception of the stereo effect is based on minimal phase differences. A lip-synchronous sound track to go with a video movie requires a precision of 10 to 100 milliseconds. Subtitles only require a 0.1 to 1 second of imprecision. Sometimes even longer deviations are acceptable (background music, slides). Note that in all cases the *cumulative* difference between the channels is what matters, not the speed difference.

In general, the synchronization problems make multimedia systems difficult (and interesting). Since each type of medium has its own characteristics, the level of support for a combination of media is a challenging design issue. Most vendors of current commercial equipment use embedded synchronization that is mapped onto a serial stream of data. As a result, they need to consider only point-type synchronization scope with a single master device. The precision is determined by the characteristics of the input source and the system load; most of the synchronization precision is supported by managing interrupt contention between the input and output devices. While this approach can lead to dramatic results, it is not sufficient if the user is to be given more control over the data being processed or if information needs to be combined from several sources (either locally or from distributed points in a network).

3. I/O Processing and the UNIX Kernel

This section will review the standard UNIX I/O model. We start with the layers of logical control that are possible within a UNIX environment to support processing of multiple data streams. We then describe the interaction of these layers when supporting UNIX I/O. Our purpose is to consider generic UNIX facilities rather than the particulars of any one UNIX implementation.

3.1. UNIX Processing Layers

Activity within a UNIX environment can be divided over five general layers in a system: the *thread*[2] layer, the *process* layer, the *kernel top-half* layer, the *kernel bottom-half* (or *interrupt*) layer, and the *device controller* layer. These layers are illustrated in figure 3. An applications program typically runs on the thread level in user mode. Upon issuing a system call, the processor first switches in to system (or privileged)

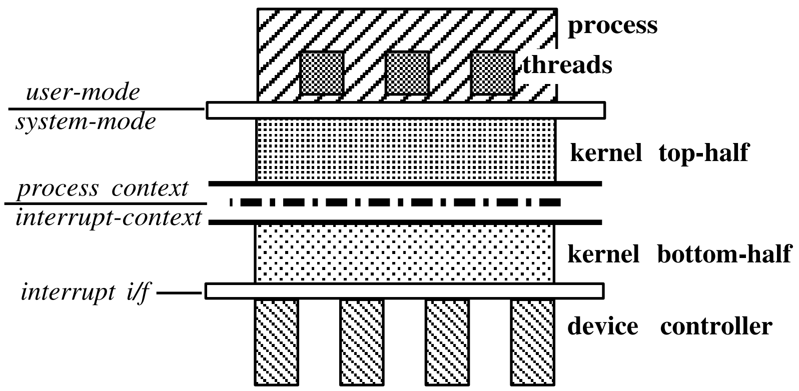


Figure 3: Elements of the UNIX processing hierarchy.

[2] Note that in systems that do not support a user threads package, a process can be considered to be an entity with a single thread of control.

mode, then acts on the request and then switches back to user mode. All of this processing typically occurs in the context of the process active at the time of the system call. Occasionally, a device will need to interrupt the running process (which is either in user mode or system mode) to service some aspect of a transfer request; such a change is know as a context switch. A typical UNIX kernel will act as the scheduler for a series of user processes, each of which will compete for time on the CPU. Within the process, one or more of its threads will be selected to execute in an implementation specific manner. (Recall that processes are entities that own resources but which do not themselves execute outside the context of their threads.) The kernel may execute its top-half code in response to a system call by the active thread. It may also execute its bottom-half code in response to an interrupt from one of the external devices (such as disks, terminals or the network) or from its system clock. Device drivers run partially as kernel top-half code and partially as kernel bottom-half code. Finally, a great deal of processing may take place within a device controller. Such processing is initiated and ultimately controlled by kernel device drivers, although much of the detailed processing happens in parallel to other processing done on the system. (Note: our partitioning of layers is based on an I/O view of UNIX processing; it is clear that other partitionings are available, but the combination we have shown is most relevant to our multimedia discussion.)

Each of the processing layers has its advantages in managing I/O transfers. The thread level provides an application program with full control over the processing of data. Unfortunately, threads represent the weakest link in a processing chain, since they may be suspended at any time when a higher-priority thread becomes available. Processing within the kernel top-half, on the other hand, is immune from suspension unless the operation suspends itself waiting for I/O completion (or resource availability) or unless an interrupt occurs. Of course, the kernel top-half code is typically not user-modifyable, making it inflexible for information processing (as opposed to information fetching, dispatching or sending). Code that executes at interrupt time has the highest probability of being allocated the CPU quickly; this can allow a fast response to time-critical events, although the amount of processing that can be done on data is usually limited by the need to serve other interrupts quickly as well and a general desire not to ‘‘hog’’ the CPU in interrupt mode. The best performing processing layer may well be the device controller layer. Here, activity can take place in parallel with other devices and the general CPU processing. The disadvantage of controller-layer processing is the limited information that such controllers have on other events taking place in the system. The consequences of this limited information are reflected in a cyclic shifting of processing responsibility that has taken place between the kernel and the device controller in recent years. SCSI disks, for example, use embedded controllers that relieve the kernel of a great deal of overhead processing. (This is usually thought of as a positive development.) Network controllers, on the other hand, have seen a shift from on-controller processing back to in-kernel processing. This is because of the inflexibility of protocol versions that are ‘‘baked’’ into hardware and the difficulty of routing information among different controllers without kernel intervention.

3.2. UNIX I/O Models

The basic I/O facility that UNIX offers to its users is that of a byte-oriented transfer mechanism based on variants of the *read* and *write* system calls. When a user issues the

read(fd, buffer, n)

system call, for example, the thread is effectively suspended until upto *n* bytes of data are transferred from device *fd* into the buffer named (in this case) *buffer*. This transfer is initiated at the thread level, then processed by the kernel top-half code, then by the device controller, then by the kernel interrupt code, then by the kernel top-half code and then by the thread code. The flow of control may block within the kernel top-half (after initiating the transfer but before the transfer is completed); at this point, the thread is descheduled pending completion of the transfer. Once the transfer-completion interrupt is received from the device, the thread is made reschedulable, although it does not run again until it is the highest-priority thread in the system. If the transfer is completed in one I/O request, the thread is usually able to continue as soon as the kernel top-half code completes (that is, as soon as the system call completes). If the operation is a read/write cycle, such as:

read(fd, buffer, n)
process the data here
write(fd2, buffer, m)

then the output data path is similar to that described above (except, of course, in the other direction!).

The movement of data across the various processing layers can be optimized by reducing the number of layers used in each transfer. Unfortunately, if the data needs to be processed in some application dependent way, the options available for speed-up are limited. The general methods of controlling the flow of information within a UNIX environment are shown in figure 4; each of the four general models are discussed in the following paragraphs. (Note that for purposes of simplification, both kernel top-half and kernel bottom-half processing are regarded as simply ‘‘kernel’’ processing in the following discussion.)

- *Controller Managed I/O:* In this class of I/O, one controller sends information directly to another controller. The kernel, a controlling thread and process data structures are typically not involved once the transfer has started. (Internal resource contention is typically handled by the system hardware.) Controller managed I/O is illustrated in figure 4.a.
- *Kernel Managed I/O:* In this class, the device controllers transfer information to the kernel, which then dispatches data to other controllers. The controlling threads and process data structures are typically not directly involved in the transfer. One example of kernel managed I/O is an in-kernel implementation of the IP networking protocol: a packet may arrive from one network device and processed by the kernel IP code, which may then route it to a separate network device. If this is done at interrupt mode, then the transfer can occur quite quickly. Of course, if too much network traffic is forwarded by this kernel, then over-all performance of the system will decrease. Kernel managed I/O is illustrated in figure 4.b.

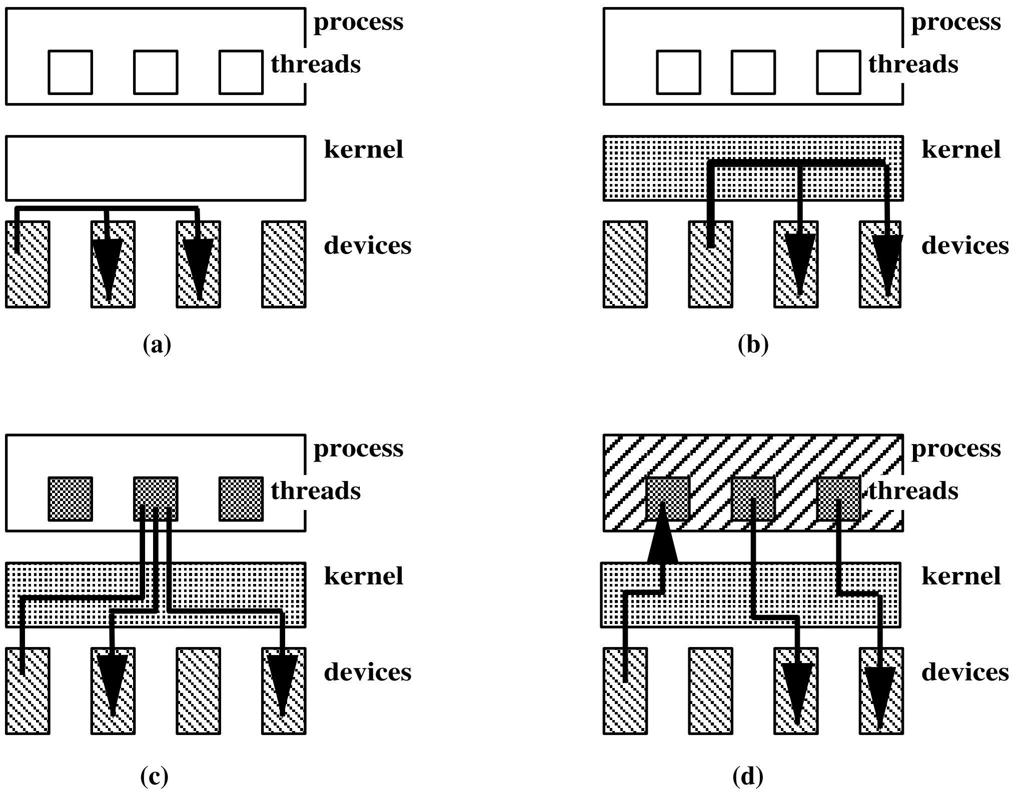


Figure 4: Interactions within UNIX for supporting I/O: (a) controller managed, (b) kernel managed, (c) thread managed, (d) process managed.

- *Thread Managed I/O:* In this class, a controlling thread is used to manage the data transfer. Information is passed from a device controller through the kernel to the thread. The kernel schedules the incoming data as well as the thread; the thread schedules the outgoing data. A single thread may schedule several transfers. The principal advantage of this technique is that it allows data that has arrived at the thread to be processed in an application-dependent manner before it is sent out to an output device. The disadvantage of this technique is that it may take a relatively long amount of time before this processing is completed—with the wait interval almost always being non-deterministic. (This, as we will see, can have very severe consequences for synchronization processing.) Thread managed I/O is illustrated in figure 4.c.
- *Process Managed I/O:* Although activity in a process can only take place in a thread, we use this class to define multi-threaded activity. Here, UNIX synchronization (such as locking and semaphore activity) is used to merge several data streams. Process managed I/O is illustrated in figure 4.d.

If large amounts of data need to be processed quickly, controller-controller I/O is typically the only useful choice available to an applications programmer. If, on the other hand, two input or output streams need to be coordinated based on their content, then thread-based (or process-based) processing may be the only option available. This is because controllers have only limited abilities to process data and because the kernel is typically a black-box that cannot be changed by the applications programmer. We consider the ramifications to multimedia I/O of this situation in the next section.

4. The Impact of the UNIX I/O Subsystem on Multimedia Interaction

The previous sections have considered the needs of multimedia applications and the general types of I/O that can be supported by a UNIX kernel. In this section, we combine these topics to consider UNIX's impact on multimedia (and vice-versa). We begin our discussion with a comparison of data location models and then discuss synchronization topics.

4.1. Location Models

In section 2.1, we defined four location models for multimedia data: *local single source (LSS)*, *local multiple source (LMS)*, *distributed single source (DSS)*, and *distributed multiple source (DMS)*. Given the fact that we are considering multimedia data, we assume that the information that is received from each source will need to be processed in some manner, either by routing a data stream into a number of sub-streams (one for each medium) or by moving composite data from one location to another. Our definitions has explicitly excluded consideration of output locations since the problems encountered here simply mirror those encountered on input. We can measure the effectiveness of the UNIX I/O system by considering the impact of the multimedia location models for each of the types of control considered above:

- *Controller Managed I/O:* in spite of its attractive performance characteristics, controller managed I/O can only play a minor role in multimedia processing. To be sure, information can quickly be transferred from one location to another in a system (such as from a laser disk to video memory), but the relative lack of control given to a user for this type of transfer—which is typically limited to start/stop/rewind/search—will ultimately limit its appeal. In terms of our models of location, controller managed I/O may be useful for LSS data that is self synchronizing, but as soon as any management of separate streams is required, the limited knowledge of the controller will restrict its usefulness.
- *Kernel Managed I/O:* for reasons of performance, kernel managed I/O can potentially play a dominant role in providing multimedia support. One example of this is the use of interrupt context processing to provide a high-speed manner of controlling and routing of incoming data. Another example is the use of multiplexing device drivers to coordinate the activity of a number of I/O streams. This model is especially useful for LMS data and (to a reasonable degree) with DSS and DMS data. Unfortunately, kernel managed I/O has a number of severe limitations, the most restrictive of which is that few application program builders have the ability (or desire) to write new device drivers to cope with in-kernel I/O processing. This is especially true for applications that need to share I/O devices with other applications; in this case, driver modules simply cannot be unlinked and relinked efficiently enough to provide the flexibility required by several applications.
- *Thread Managed and Process Managed I/O:* application-based interaction in a multi-threaded model provides the most general form of support for all types of multimedia data processing. The application programmer can dispatch as many threads as is necessary to handle each type of data. Unfortunately,

there is a catch to this flexibility: performance. The non-deterministic scheduling characteristics of UNIX systems make them unreliable at the thread level for collecting and processing information. To understand the limitations of processing at the thread level, consider that it takes about 40 microseconds for a 20-mHz processor to switch from user-mode to kernel-mode in executing a system call. (This is raw system call overhead; processing time is extra.) This means that even if we provide a set of device drivers with a great deal of memory to buffer incoming and/or outgoing data, an application still loses nearly 100 microseconds just in changing the modes necessary to initiate a data transfer between an input and an output device. Since each multimedia transfer will typically cause at least three systems calls for trivial I/O (one for fetching composite data and two for writing it out to two devices), this overhead can be substantial. Add to this the perilous scheduling situation that all threads must endure and the fact that at the thread level only limited resource management facilities exist in UNIX (such as memory locking or explicit control over kernel buffer management), then the situation at the thread level is not particularly encouraging.

The conclusion that can be drawn from this discussion is that the level that offers the best performance in the processing hierarchy (the controller level) is least useful in the general case, while the level that offers the most flexibility may not be suitable for the high-performance needs of multimedia applications. In reaching this conclusion, we have concentrated on the LSS and LMS models. The situation is even worse for the DSS and DMS models, since here multiple thread layers and multiple kernel layers (and, of course, multiple controller layers) must be transited by data as it moves from one machine to the other. We return to this point in section 5.

4.2. Synchronization Models

While the discussion above focused on the abstract gathering, processing and scattering of multimedia data, in this section we focus on the particular problems that arise in a UNIX environment for handling synchronization processing. In considering the degree to which each of the UNIX processing layers can contribute (or hinder) synchronization of parallel multimedia streams, one immediate problem with which we are confronted is the UNIX scheduler. The priority-based scheduling mechanism offered by most kernel implementations is inflexible in responding to short-term constraints that can occur while synchronizing multiple data streams. This is a consequence of basic UNIX design constraints; even so-called real-time scheduling classes within recent implementations of UNIX do not provide a user with a great deal of dynamic scheduling control to respond to transient critical conditions. Although the scheduler could conceivably be changed, most users will not be able to do this easily. This means that the synchronization of, say, lip movements to sound data will be very difficult to guarantee unless application-specific processing is included as part of the interrupt service mechanism. As was mentioned above, however, it is unrealistic to expect that a device driver (such as, say, an audio controller) will have the ability to respond to broad synchronization requirements from a variety of applications programs. Instead, we can expect that the audio controller will simply try to push out audio information at a specified rate with only minimal regard for processing in other output streams.

The problems associated with scheduling delays present themselves most clearly when considering the synchronization problems associated with multiple multimedia streams. The facilities provided by UNIX for allowing the close synchronization of even two streams are limited by the processing granularity provided to an application layer. If we assume that a 20mHz CPU will allow approximately 25,000 system calls per second under ideal situations, then it is clear that synchronizing two input streams and then sending them to two output devices has a theoretical upper limit of an average of 6,250 samples per second. In reality, there will also be controller overhead, interrupt overhead, device driver overhead and scheduling overhead, so that the actual number of samples will probably be considerably less. If we assume that we wish to synchronize two high-quality audio streams with each other (requiring no other processing than simply collecting samples and sending them out), then the highest quality we could achieve would be approximately 6kHz sampling—which is less than 15% of that available with compact discs. If we wanted to do any processing in addition to the collecting and routing of samples, the situation would only be worse. In the case of non-local data fetches, fetch delay associated with a network would need to be added to the processing delays that would accumulate at each of the contributing or consuming hosts. (This example is obviously contrived: in an actual implementation, buffering would take place within kernel processing and actual sampling would be controlled by an external clock; the purpose of the example is to illustrate that even the theoretical limits of applications control granularity do not offer a tremendous amount of processing latitude to the applications designer.)

In terms of our detailed list of synchronization types, we can make the following observations about the ability of UNIX to support multimedia processing:

- *Synchronization classes:* UNIX can do reasonably well in supporting serial synchronization of data if the sampling rates are sufficiently low to not cause a burden on the system. The block-oriented fetching of data can significantly increase the number of samples processed by an application, although the limited scheduling control of each thread will not ensure the constancy required by high-bandwidth devices. For parallel synchronization, the prospects are less promising: the sequential nature of UNIX I/O will result in either a loss of data resolution or in a limit on the number of parallel tracks that can be processed. One reason for this is the form of the generic I/O system call; all I/O is done on a single file descriptor at a time, with separate file descriptors requiring separate system calls. It may be possible to build multiplexing drivers to combine I/O on a number of file descriptors, but this will not offer a general solution to most applications builders. Another possibility may be the development of multi-file I/O system calls (with particular synchronization semantics defined in the system call argument list), but even *if* these were to become accepted by the growing list of standards organizations, most languages would be unable to cope with the notions of parallel I/O accesses. For the time being, the best one can hope to do is to provide either an applications-based multi-threaded scheduling solution to parallel stream synchronization (with all of the performance limitations discussed above) or to rely on smarter controllers to by-pass the CPU altogether.
- *Synchronization scope:* Of the two types of synchronization scope defined above, point synchronization can be relatively well-managed by the thread level, but continuous synchronization can only be managed if the input and output data rates are sufficiently low. Once again, the scope of the synchronization is not only restricted by the implementation concerns of the UNIX I/O system, but also by the ability of applications code to flexibly access data at a low-enough layer in the system.
- *Synchronization masters:* the easiest way to support synchronization within a UNIX environment is to have a master clock regulate the gathering of samples and the dispatching of samples to various output devices. In order for such a clock to function, it will need to be able to influence processing in a number of threads in the same way that real-time clock can influence the scheduling of various real-time processes. (The problems are, of course, not simply similar, they are identical.) Unfortunately, the level of real-time support in UNIX systems has never been particularly good. As for peer-level synchronization, the problems with guaranteed scheduling time under UNIX once again limit the amount of coordinated processing that can be realistically accomplished.
- *Synchronization precision:* depending on the level of precision, processing can be implemented at any of the five layers in the UNIX hierarchy. If stereo channels need to be synchronized, then it can only occur at the controller or interrupt level (unless the data need only be resynchronized at a much lower rate). If, on the other hand, subtitles need to be added to a running video sequence, then this can easily be done at the thread level.

The general dilemma of processing multimedia data remains that those applications requiring the most processing support are probably the least likely to get it in a general UNIX environment. This is not really surprising: manufacturers of high-performance output devices (such as graphics controllers or even disk subsystems) have long realized that the only way to really improve over-all system performance is to migrate this processing out of the UNIX subsystems. Unfortunately, doing so is difficult for multimedia applications, since the type of processing required over a number of input and output streams is usually beyond the scope of the implementation of any one special-purpose I/O processor. In the next section we discuss a general approach that we are investigating for providing both good performance and reasonable flexibility to multimedia applications.

5. Conclusions and an Alternative Approach

The discussions in the preceding sections can lead us to two general conclusions:

(1) The fastest processing layers in the UNIX hierarchy are the device controller and the interrupt layers; these layers enjoy high-priority scheduling and can be invoked with relatively little overhead. In terms of efficient multimedia processing, it can be argued that once you reach either the kernel top-half code or the thread/process layers, it is probably too difficult to provide efficient and deterministic multimedia processing.

(2) It can be assumed that for all but the most trivial types of fetch-and-deposit multimedia operations, it is both desirable and necessary to provide a layer of applications support to manage the interactions among the various incoming and outgoing data streams. (Recall that the entire reason for having computerized multimedia systems is the measure of control a user can have over the sequencing and presentation of pieces of data.) This type of processing is ''easily'' done at the thread/process layers, it is possible (but often impractical) at the device driver layer, it is improbable at the interrupt layer and it is usually totally unavailable at the controller layer.

The net effect of these conclusions is that it is desirable to supply a new programmable layer in the UNIX hierarchy that combine the performance benefits of the existing lower layers with the flexibility of the existing upper layers. In providing this layer, it is probably not useful to simply steal cycles from the CPU—doing this is, in effect, only replacing the existing UNIX scheduler with a semi-real-time one. If we assume that all of the normal services available to a user must continue in addition to multimedia processing, then some form of co-processing will be required to satisfy both the UNIX user and the multimedia application. In closing this paper, we provide a brief description of an approach being studied at CWI for providing multimedia applications support. This approach, which is based on a distributed I/O and processing architecture, is a generalization of existing approaches for offering high-performance graphics and computation processing on a workstation: the special-purpose co-processor.

Figure 5 illustrates the placement of a *multimedia co-processor* (MmCP) as a component of a workstation architecture. The MmCP is assumed to be a programmable device that can be cross-loaded from the master processor. It is assumed that the MmCP can execute arbitrarily complex processing sequences, and that it will have access to all or a part of the workstation's memory. As with arithmetic co-processors, a simple interface should exist to control information flow from the UNIX processor. Unlike normal co-processors, however, we assume that the MmCP will be driven by a distributed operating system that will provide communications support between its hosting workstation and other workstations in a network environment. This distributed support (Fig. 5b) will provide for coordination among the various sources and destinations in the DSS and DMS models discussed above.

In the previous sections, not much direct mention has been made of the DSS and DMS models. This is, in part, due to their relative scarcity in current multimedia systems. It is clear to us, however, that there is a great need for coordinated data transfer among various agents in a networked multimedia system. This coordination may consist of bandwidth reservation algorithms for efficient network use or intelligent algorithms for information transfer. An example of the latter type of algorithm may be a transport-style communications layer that knows to bias its service towards one type of media—such as audio—at the expense of others—such as video—if bandwidth become limited during a transmission. If one were to try this in a

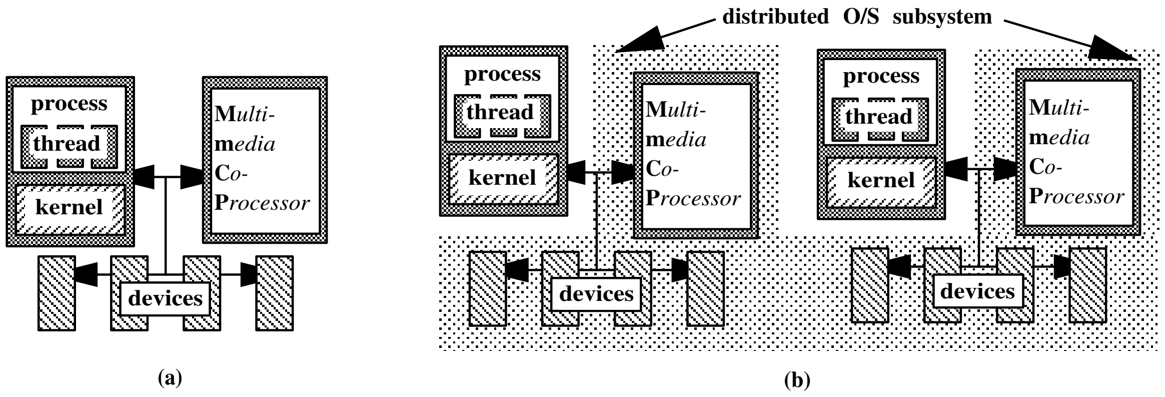


Figure 5: The Multimedia Co-Processor (MmCP): (a) local architecture, (b) as a distributed operating system.

typical UNIX kernel, then the process and mode switching time may well be longer than the adaptive period of transmission delay!

Our work is currently centered around evaluating the use of the Amoeba operating system as the basis for an MmCP [4,5]. Amoeba has two main advantages in our research: first, it has excellent communications characteristics that appear to make it suitable for light-weight protocol development; second, it is a mature but relatively unused system—meaning that it is still an open, experimental system (unencumbered by hundreds of users or thousands of standardization committee members). It should be pointed out that we are investigating *basing* our work on Amoeba, but that we do not intend to replace Amoeba. Also, unlike other operating systems research projects [6,7], we are not intending to develop a ''micro-kernel'' as such (that is, a kernel with core services for use in controlling activity on a workstation), but rather something which could be called a ''nano-kernel'': a kernel that handles a particular subset of services that can be allocated to one or more users of on a general workstation. (Figure 6.) In this sense, our work is aimed at replacing the partitioned intelligence in device controllers with a layer of shared intelligence at a super-controller level. This has the advantages of providing a full (and standard) UNIX environment plus a programmable interface layer for high-performance support.

Although we feel that a strong case can be made for the development of the MmCP (either based on Amoeba or otherwise defined), we are only starting a detailed investigation of the resource and functional partitioning requirements needed to support general multimedia systems. This research is driven by two observations:

- First, it should be clear from the sections above that the general motivation for a programmable, high-performance processing layer exists. It may be argued that this need will reduce with faster processors, although we feel that such processors will only stimulate the requirements for even higher processing rather than satisfying it.
- A second development that encourages our work is the rapid development of multi-processor workstation architectures. Although many of these systems are little more than trade-press rumors, several systems (such as the SGI PowerSeries) already provide moderate-cost multiprocessor workstations coupled with a wide array of input and output subsystems. There is no inherent reason why these systems can not simultaneously support multiple operating systems (one for the MmCP and one the remaining processors). An initial port of Amoeba to a Silicon Graphics platform (in our case, a 4D25) has been successfully completed as a proof-of-concept project.

We view these reasons as providing a basis for further work at the applications, kernel interface and architecture layers of the workstation.

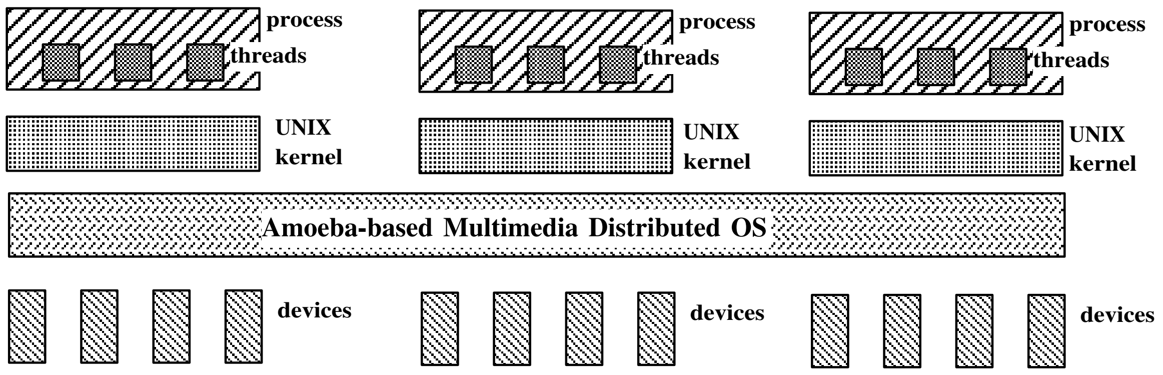


Figure 6: Multimedia support using an embedded distributed operating system.

6. Summary

We have attempted to argue that the conventional UNIX environment for workstation computing—as useful as it is for many applications—may not be idealy suited for high-performance multimedia computing. Although some of the factors that constrain UNIX are technology dependent, much of this problem lies with fundamental design issues that were a part of the original uniprocessor, sequential serial I/O model developed for UNIX in the 1970's. The approach of the multimedia co-processor that we have presented here is an attempt to overcome many of these problems without sacrificing the positive aspects of a uniform UNIX interface.

Our work is being done as part of the CWI/Multimedia research project, an interdisciplinary effort to study various related aspects of the multimedia problem. Please note that organizations and commercial firms mentioned in this paper have been selected as examples of architectures and approaches to supporting workstation computing. No attempt has been made to survey all relevant manufacturers of multimedia workstations and no particular endorsement is intended or implied for those companies referenced.

7. References

[1] *SIGGRAPH '89 Panel Proceedings,* ''The Multi-Media Workstation,'' Computer Graphics, Vol. 23, No. 5 (Dec 1989), pp 93-109.

[2] *Bulterman, D.C.A.,* ''The CWI van Gogh Multimedia Research Project: Goals and Objectives,'' CWI Report CST-90.1004, 1990.

[3] *Bulterman, van Rossum and van Liere,* ''A Structure for Transportable, Dynamic Multimedia Documents,'' Proceedings of the Summer 1991 Usenix Conference (Jun 1991), pp 137-156.

[4] *Mullender, van Rossum, Tanenbaum, van Renesse and van Staveren,* ''Amoeba: A Distributed Operating System for the 1990s,'' IEEE Computer Magazine, Vol. 23, No. 5 (May 1990), pp 44-53.

[5] *van Renesse, van Staveren and Tanenbaum,* ''Performance of the World's Fastest Distributed Operating System,'' Operating Systems Review, Vol. 22, No. 4 (Oct 1988), pp 25-34.

[6] *Accetta, Baron, Bolosky, Golub, Rashid, Young and Tevanian,* ''Mach: A New Kernel Foundation for UNIX Development,'' Proceedings of the Summer 1986 Usenix Conference (Jul 1986).

[7] *Dale and Goldstein,* ''Realizing the Full Potential of Mach,'' OSF Internal Paper, Open Software Foundation, Cambridge MA (Oct 1990).