Multimedia Synchronization and UNIX

Dick C.A. Bulterman
Robert van Liere
CWI: Centrum voor Wiskunde en Informatica
Amsterdam, The Netherlands

Abstract

One of the most important emerging developments for improving the user/computer interface has been the addition of multimedia facilities to high-performance workstations. Although the mention of multimedia I/O often conjures up visions of moving images, talking text and electronic music, multimedia I/O is not synonymous with interface bells and whistles. Instead, multimedia should be synonymous with the *synchronization* of bells and whistles so that application programs can integrate data from a broad spectrum of independent sources (including those with strict timing requirements). This paper considers the role of the operating system (in general) and UNIX (in particular) in supporting multimedia synchronization. The first section reviews the requirements and characteristics that are inherent to the problem of synchronizing a number of otherwise autonomous data sets. We then consider the ability of UNIX to support decentralized data and complex data synchronization requirements. While our conclusions on the viability of UNIX for supporting generalized multimedia are not optimistic, we offer an approach to solving some of the synchronization problems of multimedia I/O without losing the benefits of a standard UNIX environment. The basis of our approach is to integrate a distributed operating system kernel as a *multimedia co-processor*. This co-processor is a programmable device that can implement synchronization relationships in a manner that decouples I/O management from (user) process support. The principal benefit of this approach is that it integrates the potential of distributed I/O support with the standardization provided by a ''real'' UNIX kernel.

1. Introduction

One way of measuring progress in computer architectures is to study the evolution of the user/computer I/O interface. Ten years ago, the departmental minicomputer provided a dumb terminal and the occasional intelligent peripheral input or output device as the standard for user/computer interaction. Five years ago, the microprocessor-based personal workstation replaced the dumb terminal with a keyboard, a mouse and a high resolution bit-mapped display, as well as a network connection for accessing files and remote I/O devices. Currently, RISC-based workstations embody the standard of modern I/O support, adding 8-bit color displays, local disks, large main memories and a local I/O bus to the user's desktop. During this ten-year period, many of the implementation details of the user/computer interface have changed, providing faster generation and presentation of information through vastly increased processor speed, improved realism through special-purpose output technology, and (to a lessor extent) improved information precision through the use of enhanced data input facilities. Throughout this period, however, the fundamental user/computer I/O model has remained unchanged: information is first collected from one or more disjoint input sources, then transformed and filtered by a controlling application, and then passed on to one or more disjoint output destinations. The selection of I/O devices has also remained relatively static during the past decade, with input coming primarily from text- and pointer-based devices and output going to text- and picture-based devices.

While the expressive nature of the user/computer interface has seen little development during the past decade, it appears that its next evolutionary step *will* provide a fundamental change in the types of information that can be processed. The essence of this change is the introduction of temporal relationships among data sets. Consider that new workstations already provide the user with device-level access to DAT-quality stereo sound, 24-bit color images and live video input and output facilities. Taken alone, the independent manipulation of each separate information medium does not provide anything particularly new. Taken together, however, the integrated manipulation of this ‘‘multimedia’’ information provides the potential for dynamically creating new, user-directed composite data sets. (A review of the characteristics that can be expected from future multimedia workstations is provided in [1].) Unfortunately, the technology aspects of the user/computer interface have evolved more rapidly than the operating systems and programming languages interfaces for supporting dynamic temporal data relationships. As a result, it is clear that the essential problem of multimedia is not simply providing I/O support for sound and video. Rather, the essential problem of multimedia is that support must be provided for *synchronizing* otherwise autonomous data transfers within and across workstations.

To appreciate the scope of the multimedia support problem, consider that the development of programming languages, application-programmer interfaces, operating systems and device controllers have all been based on a model of user/computer interaction that treats I/O activity as a series of unrelated data movement operations on a set of independent (device) files. The UNIX[1] operating system is a good example of how I/O is currently managed. Here, each I/O request is passed to a device driver layer within the UNIX kernel; this layer is responsible for scheduling (possibly buffered) device transfers in such a way that the application in presented with a uniform sequential I/O model. The driver usually does not interpret any of the the data it is moving, leaving the coordination of multiple I/O operations within a process to the application layer. Since each driver is designed to manage activity within one device queue, low-level coordination of multiple resources is left to a first-come, first-serve contention strategy or to individual characteristics of the underlying hardware architecture. As a result of the UNIX I/O model, there is no way for an application to request coordinated input from multiple devices as an operating system primitive action, there is no way for the operating to conveniently coordinate separately specified I/O actions, and there is no way for a device controller to know how to effectively schedule itself in cooperation with other I/O devices. The situation becomes even more complicated when the sources and/or destinations of information are located on different workstations in a network. Here, no amount of *ad hoc* operating systems hacking will provide a comprehensive model for the synchronization of distributed time-based data.

In this paper, we consider the impact of supporting multimedia synchronization in a UNIX environment. (We use UNIX as a model for study because of it is representative of the types of operating environments that a multimedia user can expect to encounter.) We begin with a study of the scope of the problem by discussing two classes of multimedia data models: multimedia data location models and multimedia data synchronization models. We then consider these models in terms of the processing hierarchy provided by UNIX to see if UNIX is ‘‘multimedia ready’’. In so doing, we concentrate on current workstation-based user/computer I/O in local and distributed environments. We conclude by offering an alternative approach that we are studying as part of the CWI/Multimedia project [2,3] to better support distributed multimedia applications; this approach seeks to combine the benefits of providing a standard UNIX user interface model for common applications use with a co-equal distributed I/O subsystem that can be scheduled to implement multimedia data synchronization across a network of workstations.

---

[1] UNIX is a registered trademark of AT&T/USL in the United States and other countries.

2. Two Models of Data Interaction in Multimedia Systems

The broadest notion of multimedia encompasses the integration of arbitrary spatial and temporal data sets to encode and/or present information. In defining general support for this integration, it is important to model two aspects of information interaction. The first model defines the allowable data location relationships that exist among information sources and destinations; the second model defines the allowable synchronization relationships that exist among data sets. Both of these models 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 is required to gather and route scattered data. Figure 1 reviews four general models that can describe the sources and destinations for multimedia information.

a) *Local single source.* This model has all data originating at a single source and routed to several destinations. An example is a CD-ROM that contains sound, text and picture information. This model provides the least synchronization complexity because all intra-sample synchronization is the responsibility of the source material designer. Inter-sample synchronization is implemented by fetching input blocks at a predefined rate and then routing them (by either the device controller, the operating system kernel or the application) to one or more output devices.

b) *Local multiple sources.* This model is similar to (a), except that source data is scattered across several devices. An example of this type of interaction is combining voice annotation with images in an electronic slide-show. The principal difference between single and multiple source data is the need for external (to the data) synchronization among the data streams.

c) *Distributed single source.* In this model, we assume that a single source of information exists that is located on a remote location relative to the workstation managing the user/computer interface. The single-source nature of data means that no multi-stream synchronization is necessary but that the remote location of the data will require compensation for transfer delays.

d) *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. Synchronization problems include I/O scheduling across a set of workstation, transfer delays across several connections and processing delays at one or more sites.

The central issue in supporting the various location models is determining where synchronization is implemented in a system's processing hierarchy. (The hierarchy is: application code, operating systems code, device controller.) In the local single source model, the choice depends on the input sampling rate of the data. In the local multiple source model, the required synchronization can be placed in either user application code or within the operating system kernel; synchronization at the user level will yield more flexibility while synchronization at the kernel level will yield better performance. In the distributed single source model, the transfer-based synchronization can be accomplished by buffering data at the source or the destination; note that whatever option is used, data control is distributed over at least two kernels—the sending and receiving—as well as several protocol layers and at least one user layer. The distributed multiple source mode is the most interesting because it combines aspects of synchronization problems with transfer delays and raw information scheduling. The impact of data location on the processing hierarchy for UNIX systems will be discussed in detail in section 3, after we first consider exactly what we mean by synchronization.

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. The illustration in figure 2 can be used to get an intuitive feel for the general synchronization relationships that can exist among multimedia data streams. In this picture, we define five information streams that interact to provide a composite multimedia story. Each stream is made up of a number of blocks of information, with the timing of the presentation of each block dependent in some way on the presentation of information in other blocks. The details of the inter- and intra-stream relationships are beyond the scope of this paper. (Interested readers are referred to [3].) What is important is notion that 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.

a) *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.

b) *Synchronization scope.* We can distinguish 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.

c) *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.

d) *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 diversity in individual device characteristics makes the level of support for a combination of media 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. The Impact of the UNIX I/O Subsystem on Multimedia Interaction

The previous section has characterized aspects of multimedia information. In this section, we consider the impact of multimedia information on UNIX (and vice-versa). We begin with a discussion of data location models and then consider synchronization topics.

3.1. Location Models

As illustrated in figure 3, processing within a typical UNIX system can take place at the following levels: within a physical I/O controller, within the operating system kernel, and within an application thread and/or process. Processing that occurs within an I/O controller is done in parallel with other activity in the system. Processing within the kernel is done in *system mode*, either in *process context* (that is, as part of the low-level support activity for a particular process) or in *interrupt context* (which is time that is not directly associated with the support of a particular process's threads). Processing within at the thread/process level is always done in *user mode* in the context of a particular process.

If controller-based I/O processing is supported, then information can be fetched, synchronized and output transparently within a particular device controller. (This assumes that the device controller can manage all the required data sources and destinations.) Operating system kernel I/O processing includes the conventional activity of device drivers, both in the interrupt and the process contexts. Thread/process I/O processing includes all user-level I/O synchronization. In general, controller I/O is immune from interrupt overhead. Kernel I/O will be immune from preemption (that is, a process switch resulting from the scheduler's determination that a higher priority process became ready to run), but it will typically not be immune from interrupts. Thread/process I/O is subject to both preemption and interrupts.

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 potentially attractive performance characteristics, controller-managed I/O can only play a minor role in general multimedia processing. The single benefit of this type of I/O control is that information can be quickly fetched, synchronized and routed without interference from other system activity. Unfortunately, since the entire point of providing multimedia services is to give the user the ability to integrate separate information streams, some measure of ''interference'' beyond start/stop/rewind/search will nearly always be required. In terms of our models of location, controller-managed I/O may be useful for local single source data that is self-synchronizing, but as soon as any management of separate streams is required, the limited scope 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 high-speed control 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 local multiple source data and (to a lessor degree) with distributed 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 option of writing 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. Note that for distributed data, even in-kernel manipulation of data will be limited by the fact that there is no cross-kernel buffer sharing possbile among separate UNIX systems.

- *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 cost for 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.

Our consideration of the impact of data location on performance with the UNIX has concentrated on local data location models. The situation is even worse for distributed data, since here multiple thread layers and multiple kernel layers and multiple controllers must be transited by data as it moves from one machine to the other. We return to this point in section 4.

3.2. 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. The primary obstacle here 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; 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 have this as an option.

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.

3.3. Is UNIX Multimedia Ready?

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

(1) The fastest and most responsive layers in the UNIX I/O 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 processing efficiency, it can be argued that once you reach either the normal kernel or 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 synchronization 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 the next section, 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.

4. An Alternative Approach to Providing Multimedia Synchronization Support

Supporting multimedia synchronization on a local workstation requires a balance between I/O data rates, system scheduling and user interaction control. In a local environment (such as in the local single source and local multiple source models), it is possible that a combination of clever implementation techniques, creative kernel hacking and low user expectations can produce interesting multimedia results. (The wealth of personal computer multimedia applications prove this to be true.) Unfortunately, for many users, clever kernel hacking is not an option; they expect to run standard applications on standard systems that exhibit standard (if not always exciting) functionality. Even if it were possible to modify local UNIX implementations to improve I/O performance, the result can be the isolation of data and user/computer functionality just at a time when data sharing across applications in a networked environment is at the heart of modern computer evolution.

Decentralized data sharing requires a level of communication and coordination that surpasses that which can be cleverly added to a single kernel. 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 typical UNIX kernel, then the process and mode switching time may well be longer than the adaptive period of transmission delay!

In order to address the twin issues of decentralized data sharing and local UNIX standardization, we have been investigating the development of alternative workstation support for multimedia synchronization. Figure 4 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 UNIX 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 separate distributed operating system that will provide communications support between its hosting workstation and other workstations in a network environment. This distributed support (Fig. 4b) will provide for coordination among the various sources and destinations in the local and distributed location models discussed in section 2.

The development of the MmCP is driven by the following three observations:

(1) 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 be satisfied at the thread/process layer by faster processors, although we feel that such processors will only stimulate the requirements for even higher processing rather than satisfying it.

(2) The considerable standardization effort currently being undertaken for UNIX (and UNIX-like) systems demonstrates the fundamental importance of providing a compatible user interface for running a variety of application programs. While a totally new operating system could be developed with a better scheduler, a lighter-weight I/O system and a more flexible user/system interface, such a system would inevitably need to be loaded down with a UNIX support layer to gain general acceptance. Other approaches, such as assuming that UNIX will go on a functionality diet and be transformed into a lean-and-mean multimedia operating system, have no basis is recent history; in fact, all recent versions of UNIX have gotten functionally fatter as a part of the standardization process.

(3) A parallel development that encourages our work is the rapid development of multiprocessor workstation architectures. Although many of these systems are little more than trade-press rumors, several systems 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 or more for the MmCP and the rest for UNIX processing.)

The MmCP provides a reasonable balance between keeping the benefit of nearly fifteen years of UNIX development and providing new, high performance services for tasks such as synchronization.

If we examine the location models in section two, we can see that the MmCP will probably achieve the greatest benefit for distributed data. Its impact on controlling local I/O can also be substantial, however. In fact, one way of viewing the MmCP is as a very intelligent (and programmable) device controller rather than a second operating system. As far as synchronization support is concerned, mechanisms will be necessary to provide the required primitives for application program support, but these primitives can be developed in a much more flexible environment than that of the UNIX kernel.

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 mature but relatively unused—meaning that it is still an open, experimental system that is 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, not adding multimedia functionality to Amoeba. It is expected that various tasks that can be performed by the UNIX processors need not necessarily be duplicated in the MmCP. 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 5.) 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.

5. 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.

6. Acknowledgments

Members of the CWI/Multimedia group (notably Guido van Rossum, Dik Winter and Jack Jansen) have played important role in the development of the ideas in this paper. In particular, Guido van Rossum contributed the synchronization class definitions in section 2.2. An early version of this paper was presented at the 1991 EurOpen Autumn technical conference in Budapest, Hungary.

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).

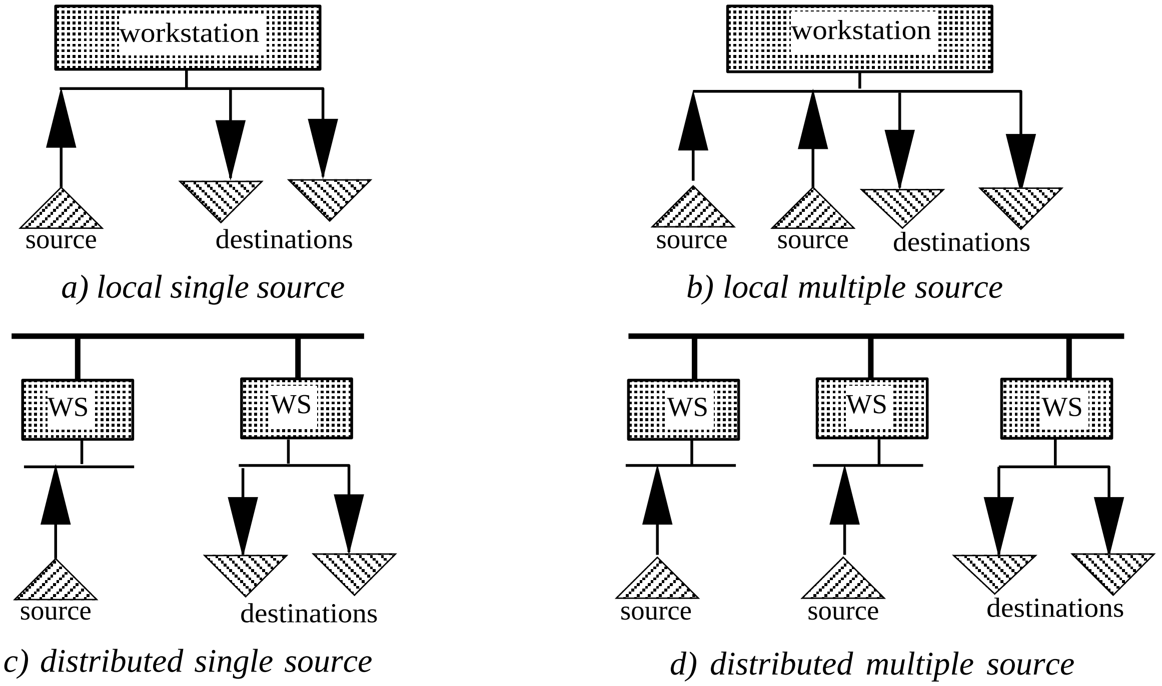

Figure 1: Four location models for multimedia data.

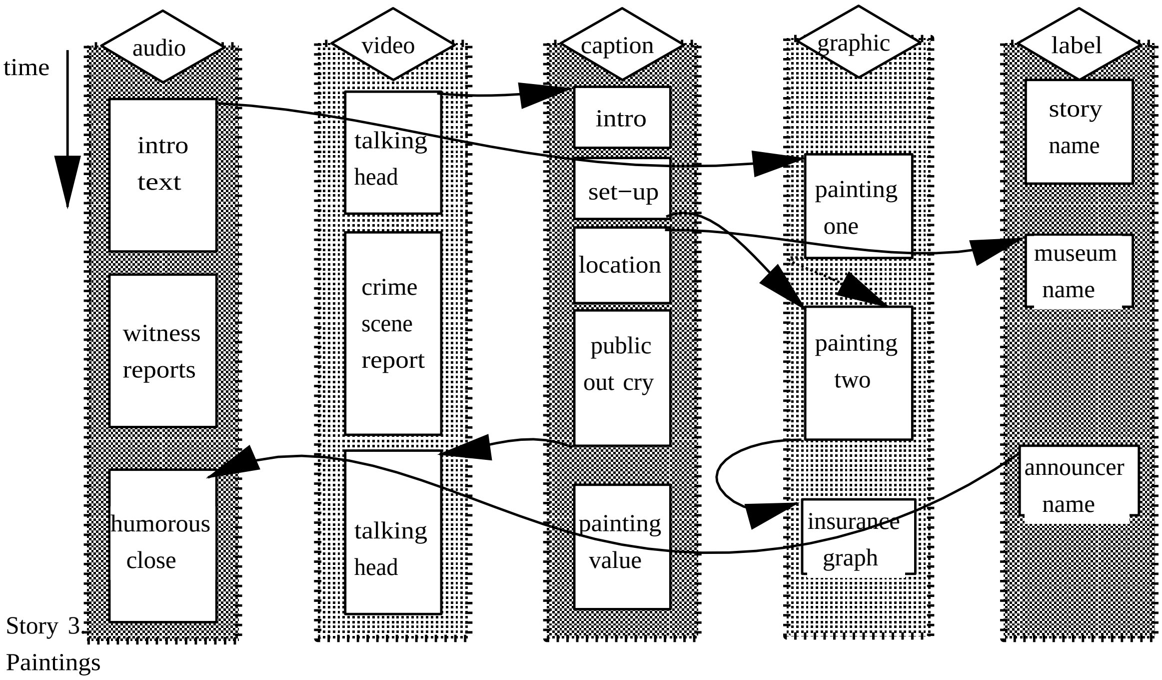

Figure 2: Synchronization in a composite multimedia document [3].

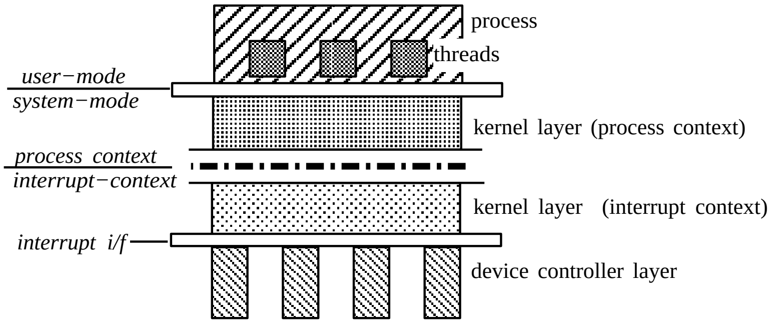

Figure 3: Elements of the UNIX processing hierarchy.

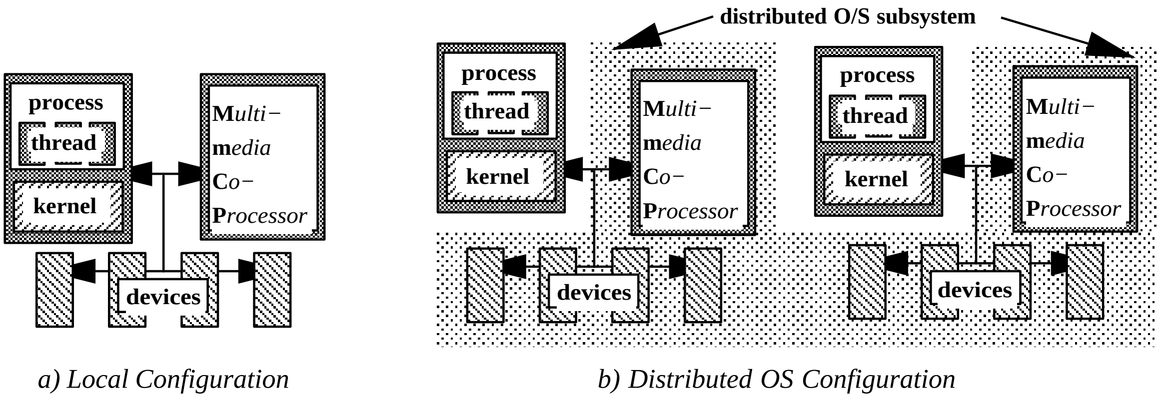

Figure 4: The Multimedia Co-Processor (MmCP).

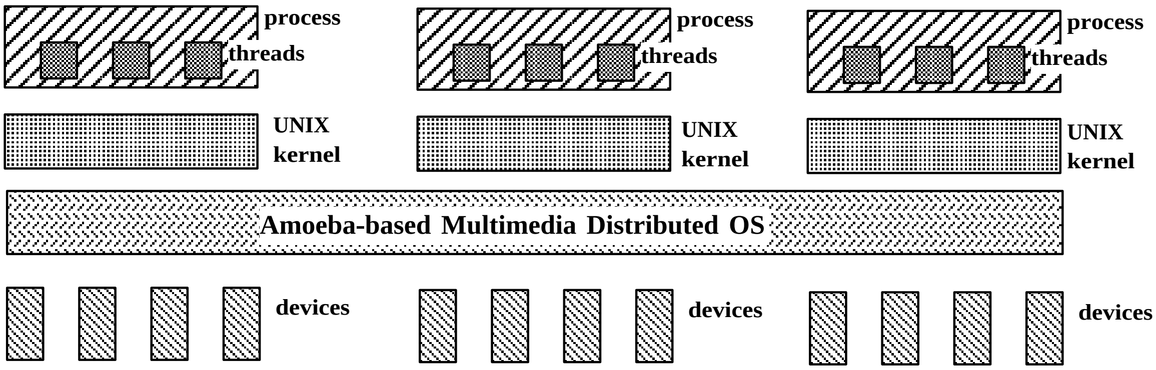

Figure 5: Multimedia support using an embedded distributed operating system.