Analyzing the Multimedia Operating System

Ralf Steinmetz
IBM European Networking Center

What makes an operating system able to handle multimedia? This survey outlines the main features an operating system must possess—from resource management to file system issues—with an emphasis on scheduling, and it provides a vision of the optimal multimedia system architecture.

The operating system shields the computer hardware from all other software. It provides a comfortable environment for program execution and ensures effective use of hardware. The operating system offers various services related to the essential resources, such as the CPU, main memory, storage, and all input and output devices. Integrating discrete and continuous multimedia data demands additional services from operating system components, especially real-time processing of continuous-media data. This survey focuses on basic concepts and internal tasks of a multimedia operating system because application interfaces are often implementation- and product-specific and change rapidly, while the fundamental mechanisms will remain the same for at least the near future.[1] A broader discussion of the most important aspects of multimedia appears elsewhere.[2,3]

This article surveys the unique services multimedia systems require of their operating systems. First it presents a model of the processing of continuous-media data. Then it shows how process management must take into account the timing requirements imposed by real-time and non-real-time multimedia data and apply appropriate scheduling methods. To accommodate timing requirements, resource management treats single components as resources reserved prior to data processing. File management services provide access to single files and file systems. Memory management provides access to data with guaranteed timing delay and efficient data manipulation functions. Resolving all these issues leads to an optimal architecture for multimedia systems.

Process management

Process management deals with the main processor resource, whose capacity is specified as processor capacity. The process manager maps single processes onto the CPU resource according to a specified scheduling policy such that all processes meet their requirements.

The main characteristic of real-time systems is the need for correctness. This applies not only to errorless computation, but also to the time at which the result is presented. Hence, a real-time system can fail not only because of massive hardware or software failures, but also because the system is unable to execute its critical workload in time.[4] When a system acts deterministically, it adheres to previously defined time spans for data manipulation; that is, it guarantees a response time. Speed and efficiency are not, as often assumed, the main characteristics of a real-time system. For example, the playback of a video sequence in a multimedia system is acceptable only when the video is presented neither too fast nor too slow. Multimedia systems must also consider timing and logical dependencies, both internal and external, among different, related tasks processed at the same time. In the context of multimedia data streams, this refers to the processing of synchronized audio and video data where the timing relation between the two media has to be considered.

Audio and video data streams consist of single, periodically changing values of continuous-media data, such as audio samples or video frames. Each logical data unit must be presented at a specific deadline. Jitter is allowed only before, not during, the final presentation. A piece of music, for example, must be played back with constant speed. However, recent research at IBM Heidelberg showed that users may not perceive a slight jitter at media presentation, depending on the medium and the application.[5]

Today's operating systems will form the base of continuous-media processing on workstations and personal computers for years to come. The market will be reluctant to accept newly developed multimedia operating systems; therefore, existing multitasking systems must cope with multimedia data handling, as the sidebar "Multitasking real-time processes" explores.

1070-986X/95/$4.00 © 1995 IEEE

Multitasking real-time processes

Unix and its variants, Microsoft's Windows, Apple's System 7, and IBM's OS/2, in descending order, are the most widely installed operating systems with multimedia capabilities. Although some include special priority classes for real-time processes, this is not sufficient for multimedia applications. For example, one group of researchers tested the SVR4 Unix scheduler, which provides a real-time static priority scheduler in addition to a standard Unix time-sharing scheduler.[1] The test ran three applications concurrently: "typing," an interactive application; "video," a continuous-media application; and a batch program. Only through trial and error did the SVR4 scheduler find a particular combination of priorities and scheduling class assignments that worked for a specific application set. This indicates a need for additional features for scheduling multimedia data processing.

OS/2 offers three possible models for multimedia support. First, the device-drivers-as-process-manager approach implements operating system extensions for continuous-media processing as physical device drivers (PDDs). In this approach, a real-time scheduler and the process manager run as PDDs activated by a high-resolution timer. In principle, this is the implementation scheme of the OS/2 Multimedia Presentation Manager, the multimedia extension to OS/2.

Second, when an enhanced system scheduler functions as the process manager, it can process time-critical tasks together with normal applications running in ring 3, the OS/2 user space. Each real-time task is assigned to a thread running in the time-critical priority class. (In OS/2, a thread is equivalent to a process in the overall discussion.) A thread is interrupted if another thread with higher priority—there are 32 levels—requires processing. Noncritical applications run as threads in the regular class, which also has 32 priorities. They are dispatched by the operating system scheduler according to their priority. For fairness, the scheduler itself may rearrange priorities of threads running in this class.

The main advantage of this second approach is the control and coordination of all time-critical threads through a higher instance, the system scheduler. This instance, running with a higher priority than all other threads, controls and coordinates threads according to the adapted scheduling algorithm and the respective processing requirements. It can observe the runtime behavior of single threads. Another entity, the resource manager, determines feasible schedules, takes care of quality-of-service calculation and resource reservation, and regulates competition for the CPU. An internal scheduling strategy and resource management allows processing guarantees, but it requires that the native scheduler be enhanced and that no other user assigns time-critical threads.

Third, in the metascheduler-as-process-manager approach, the normal priority-driven system scheduler schedules all tasks. A metascheduler then assigns priorities to real-time tasks. Non-time-critical tasks are processed when no time-critical task is ready for execution. Many Unix systems use this metascheduler approach. However, in an integrated system, the management of continuous-data processes will not require a metascheduler; it will be part of the system process manager itself.

Reference

1. J. Nieh et al., "SVR4Unix Scheduler Unacceptable for Multimedia Applications," *Proc. 4th Int'l Workshop on Network and Operating System Support for Digital Audio and Video*, Springer-Verlag, Berlin, 1993; appeared as Lecture Notes in *Computer Science*, Vol. 846, Springer-Verlag, 1994.

Scheduling

To fulfill the timing requirements of continuous media, the operating system must use real-time scheduling techniques. Traditional real-time scheduling techniques, used for command and control systems in application areas such as factory automation or aircraft piloting, demand high security and fault tolerance. These demands often conflict with real-time scheduling efforts applied to continuous media. Multimedia systems outside of traditional real-time scenarios have different—in fact, more favorable—real-time requirements:

- The fault-tolerance requirements of multimedia systems are usually less strict than those of real-time systems with a direct physical impact. A short-time failure of a continuous-media system, such as a delay in delivering video-on-demand, will not directly lead to the destruction of technical equipment or constitute a threat to human life (with the exception of applications such as support of remote surgery).
- For many applications, missing a deadline in a multimedia system is, though regrettable, not a severe failure. It may even go unnoticed: If an uncompressed video frame is not prepared on time, it can simply be omitted, assuming this does not happen for a contiguous sequence of frames. (Audio requirements are more stringent because the human ear is more sensitive to audio gaps than the human eye is to video jitter.)
- A sequence of digital continuous-media data results from periodically sampling a sound or image signal. Hence, in processing the data units of such a data sequence, all time-critical operations are periodic. Scheduling periodic

Scheduling experiments

The Advanced Real-Time Technology Operating System (ARTS) is a real-time operating system for a distributed environment with one real-time process manager, developed by the Computer Science Department of Carnegie Mellon University. It runs a network of Sun 3 workstations, connected with a real-time network based on the IEEE 802.5 token ring and Ethernet. To solve scheduling problems, the ARTS developers adopted a time-driven scheduler (TDS) with a priority inheritance protocol. This protocol prevents unbounded priority inversion among communication tasks. Tasks with hard deadlines are scheduled according to the rate-monotonic algorithm, with other scheduling methods included for experimental reasons.[1]

Yet Another Real-Time Operating System was developed at the University of North Carolina at Chapel Hill as an operating system kernel to support teleconferencing applications.[2] YARTOS includes an optimal, preemptive algorithm to schedule tasks on a single processor and an integrated synchronization scheme to access shared resources with the EDF algorithm. Here, a task has two notions of deadline, one for the initial acquisition of the processor and one for execution of operations on resources. To avoid priority inversion, tasks receive separate deadlines for performing operations on shared resources. No shared resource can be accessed simultaneously by more than one task, and even a single task only occupies a shared resource as long as absolutely necessary.

The split-level scheduler was developed within the DASH project at the University of California at Berkeley to provide better support for multimedia applications.[3] It applies a deadline/work-ahead scheduling policy under which critical processes have priority over all other processes and are scheduled preemptively according to the EDF algorithm. Interactive processes have priority over work-ahead processes as long as they do not become critical. The scheduling policy for work-ahead processes is unspecified but may be chosen to minimize context switching. Non-real-time processes use a scheduling strategy like Unix time-slicing.

The three-class scheduler was developed as part of a video-on-demand file service at Digital Equipment Corp. The design of the scheduler is based on a combination of weighted round-robin and rate-monotonic scheduling that supports three classes of schedulable tasks.[4] A general-purpose task is preemptable and runs with a low priority. The real-time class is suitable for tasks that require guaranteed throughput and bounded delay. The isochronous class supports real-time periodic tasks that require performance guarantees for throughput, bounded latency, and low jitter. The isochronous class—with the highest priority—applies the rate-monotonic algorithm, while the real-time and the general-purpose classes use the weighted round-robin scheme. The scheduler executes tasks from a ready queue in which all isochronous tasks are arranged according to their priority. At the arrival of a task, the scheduler determines whether the currently running task has to be preempted. General-purpose tasks are immediately preempted, real-time tasks are preempted in the next preemption window, and isochronous tasks are preempted in the next preemption window if their priority is lower than that of the new task. Whenever the queue is empty, the scheduler alternates between executing the real-time and general-purpose classes.

IBM's European Networking Center in Heidelberg developed a metascheduler for the operating systems AIX and OS/2 to support real-time processing of continuous media.[5] Rates are mapped onto system priorities according to the rate-monotonic algorithm. Experience with the OS/2 metascheduler shows the limits of this approach. For example, each single process in the system can run with a priority initially intended for real-time tasks. These processes are not scheduled by the resource manager and therefore violate the calculated schedule. A malicious process can block the whole system simply by running with the highest priority and not giving up control.

References

1. C.W. Mercer and H. Tokuda, "The ARTS Real-Time Object Model," *Proc. IEEE Real-Time Systems Symp.*, IEEE Press, Piscataway, N.J., 1990, pp. 2-10.
2. K. Jeffay, D.L. Stone, and D.E. Poirier, "YARTOS: Kernel Support for Efficient, Predictable Real-Time Systems," *Proc. IFAC Workshop on Real-Time Programming*, Pergamon Press, 1991.
3. D.P. Anderson, "MetaScheduling for Distributed Continuous Media," *ACM Trans. on Computer Systems*, Vol. 11, No. 3, Aug. 1993, pp. 226-252.
4. K.K. Ramakrishnan et al., "Operating System Support for a Video-On-Demand File Service," *Proc. 4th Int'l Workshop on Network and Operating System Support for Digital Audio and Video*, Springer-Verlag, Berlin, 1993, pp. 225-236.
5. A. Mauthe, W. Schulz, and R. Steinmetz, *Inside the Heidelberg Multimedia Operating System Support: Real-Time Processing of Continuous Media in OS/2*, IBM Tech. Report 43.9214, 1992.

tasks is much easier than scheduling sporadic ones.[6]

- The bandwidth demand of continuous media is not always that stringent. Since some compression algorithms can employ different compression ratios for different qualities, the required bandwidth can be negotiated. If not enough bandwidth is available for full quality, the application can accept a reduced quality over no service at all. The quality may also be adjusted dynamically to the available bandwidth, by changing encoding parameters, for example. This is known as *scalable video*.

In a traditional real-time system, timing requirements result from the physical characteristics of the technical process to be controlled: They are provided *externally*. Some multimedia applications must meet external requirements, too. A distributed music rehearsal is one example: Music played by one musician on an instrument connected to his workstation has to be made available to all other members of the orchestra within a few milliseconds, or the underlying knowledge of a global unique time is disturbed.

If human users are involved just in the input or output of continuous media, delay bounds are more flexible. Consider the playback of a video from a remote disk. The delay of a single video frame transferred from the disk to the monitor is unimportant. Frames must only arrive in a regular fashion. Users will notice any difference in delay only as an initial delay in response to their "start play" commands.

Most multimedia operating systems apply one of the methods discussed above. Some systems, such as those discussed in the sidebar "Scheduling experiments," replace the scheduler with a real-time scheduler. These systems can be viewed as new operating systems because they are usually not compatible with existing systems and applications. Other systems apply a metascheduler based on an existing process manager. Only these systems will have a commercial impact in the short and medium terms because they can run existing applications.

Processing requirements

Continuous-media data processing has to occur in precisely predetermined, periodic intervals. Operations on this data recur over and over and must be completed at certain deadlines. The real-time process manager determines a schedule that allows the resource CPU to make reservations and to give processing guarantees. The problem is finding a feasible schedule that allows all time-critical, continuous-media tasks to meet their deadlines. This must be guaranteed for all tasks in every period for the whole runtime of the system, since a multimedia system processes continuous and discrete media data concurrently.

A system scheduling multimedia tasks must consider two conflicting goals. On the one hand, noncritical process should not suffer unnecessarily because of time-critical processes. Multimedia applications rely as much on text and graphics as on audio and video. On the other hand, a time-critical process must never experience priority inversion, either between critical and noncritical tasks or between time-critical processes with different priorities.

Apart from the overhead caused by the schedulability test and the connection establishment, the cost of scheduling every message must be minimized. Such costs are critical because they occur periodically with every message at the start of real-time processing. The overhead generated by the scheduler and the operating system adds to the processing time, so should also be minimized. The timing behavior of the operating system and its influence on the scheduling and processing of time-critical data can lead to time-garbled applications. Therefore, operating systems in real-time systems cannot be assessed separately from the application programs, and vice versa.

Preemptable versus nonpreemptable task scheduling

The problems involved in attaining real-time processing are widely known in computer science.[7] The goals of traditional scheduling on time-sharing computers are optimal throughput, optimal resource utilization, and fair queuing. In contrast, the main goal of real-time tasks is to provide a schedule that allows as many time-critical processes as possible to be processed in time to meet their deadlines. The scheduling algorithm has to map tasks onto resources so that all tasks meet their time requirements.

One reason tasks are usually treated as preemptable is that for some task sets nonpreemptable scheduling is impossible, where preemptable scheduling might be possible. Figure 1 (next page) shows such an example.

Nagarajan and Vogt introduced the first schedulability test for nonpreemptable tasks.[8] They proved that a set of m periodic streams with periods p_i, deadlines d_i, and processing times e_i, where $d_i \leq p_i \; \forall \; i \in (1, \ldots, m)$, is schedulable if the time between the logical arrival time and the deadline of a task t_i is larger than or equal to the sum of the set's processing time e_i and the processing time of any higher priority task that requires execution during that time interval plus the longest processing time of all lower and higher priority tasks that might be serviced at the arrival of the task t_i. The schedulability test is an extension of Liu and Layland's.[9] Consequently, nonpreemptable continuous-media tasks can also be scheduled. However, the scheduling of nonpreemptable tasks is less favorable than for preemptable tasks because the number of schedulable task sets is smaller.

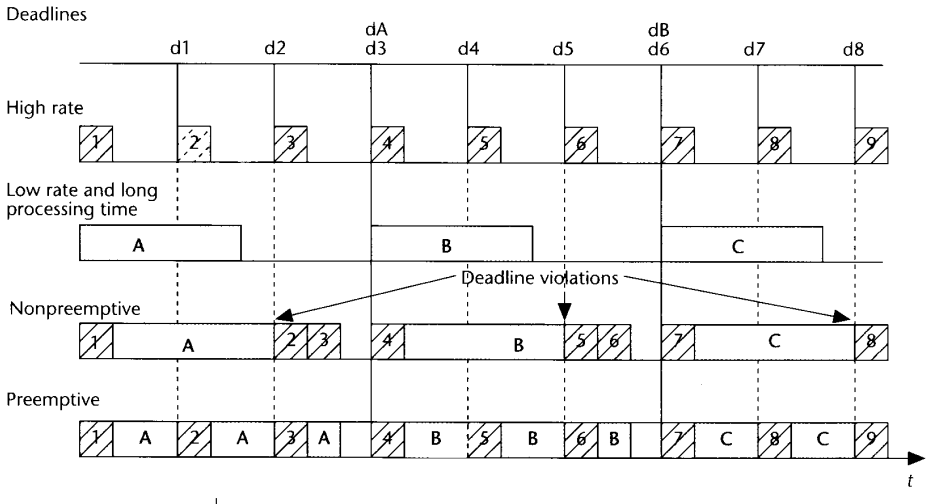


Figure 1. Preemptable scheduling methods often succeed in making a workable schedule where nonpreemptable methods fail, as this diagram of rate-monotonic scheduling shows.

To achieve full real-time capabilities, we must at least extend the native scheduler of the operating system. The operating system should be enhanced with a class of fast, nonpreemptable threads and the ability to mask interrupts for a short period of time. Priorities in this thread class should only be assigned to threads already registered by the resource manager and monitored by a system component with extensive control facilities. Another possibility is to enhance the performance of the scheduler itself by incorporating some mechanisms of real-time scheduling like EDF. In any case, the operating system should provide a time measurement tool that allows the measurement of pure CPU-time and a timer with a finer granularity.

System model

Because the essential aspect of any multimedia operating system is real-time operation, I will first establish a common, basic definition of real time as it relates to multimedia. The German National Institute for Standardization (DIN), similar to the American National Standards Institute (ANSI), defines a real-time process in a computer system as "a process which delivers the results of the processing in a given time-span." Data processing programs must be available during the entire runtime of the system, since the data may require processing at unexpected times.

The real-time system must enforce externally defined time constraints while considering internal dependencies and their related time limits. External events occur, depending on the application, either deterministically (at a predetermined instant) or stochastically (randomly, with unknown timing). A real-time system has the permanent task of receiving information spontaneously or periodically from the environment and delivering the processed data back to the environment on time.

For the purposes of discussion, I evaluated all scheduling algorithms here using the following real-time system model, whose essential components are resources, tasks, and scheduling goals. A *task* is a schedulable system entity, corresponding to the notion of a thread or a process. A *periodic* task is one that sends consecutive requests at constant intervals. A real-time system characterizes a task by its timing constraints as well as its resource requirements. The system model I used covers only periodic tasks without precedence constraints, in which the processing of two tasks is mutually independent, which poses no restriction for multimedia systems. Today, the playback of synchronized data, for example, requires only a single process in most of the available multimedia systems. On the other hand, a playback of synchronized streams by two or more processes is, in general, not a problem because related streams allow for a certain skew. This skew is usually higher than both the accuracy of scheduling—the *granularity* of the system—and the period time p.

We can define the time constraints of the periodic task T by $0 \le e \le d \le p$, where e is the processing time, d is the deadline, and p is the period of T. The rate r of T equals $1/p$. The starting point s is the first time where the periodic task requires processing (see Figure 2). After that, it requires processing at intervals of e.

At $s + (k - 1)p$, the task T is ready for processing in period k. This processing must be finished at $s + (k - 1)p + d$. For continuous-media tasks, we can assume that the deadline of the period $(k - 1)$ is the ready time of period k. This is known as *congestion-avoiding deadlines*: The deadline for each message coincides with the period of the respective periodic task.

Tasks can be preemptable or nonpreemptable. A preemptable task can be interrupted by any task with a higher priority, later continuing processing

at the interruption point. A nonpreemptable task, in contrast, cannot be interrupted until it voluntarily yields the processor. Any high-priority task has to wait until the low-priority nonpreemptable task is finished. In such a case, the high-priority task suffers priority inversion. For our purposes, therefore, all tasks processed on the CPU are considered preemptable unless otherwise stated.

In a real-time system, the scheduling algorithm must parcel out an exclusive, limited resource that different processes use concurrently such that all tasks can be processed without violating any deadlines. This notion can be extended to a model with multiple resources of the same type, such as a multiprocessor system. It can also cover different resources such as memory and bandwidth. A scheduling algorithm *guarantees* a newly arrived task when it can find a schedule where in every period over the whole runtime, the new task and all previously guaranteed tasks can finish processing by their deadlines.[10] To do this, the algorithm must be able to check the schedulability of the newly arrived tasks. A scheduling algorithm can use the processor utilization—the amount of processing time used by guaranteed tasks over the total amount of processing time—as a performance metric.[9]

Earliest deadline first

Most attempts to solve real-time scheduling problems are just variations on two basic algorithms for multimedia systems: earliest deadline first and rate-monotonic scheduling. The earliest-deadline-first (EDF) algorithm is the best-known algorithm for real-time processing. At any arrival of a new task, EDF immediately computes a new order; that is, it preempts the running task and schedules the new task according to its deadline. Processing of the interrupted task continues later. EDF handles not only periodic tasks, but also tasks with arbitrary requests, deadlines, and service execution times. However, in an arbitrary case of an overload situation, EDF cannot guarantee the processing of any task.

EDF is an optimal, dynamic algorithm. A *dynamic* algorithm schedules every instance of each incoming task according to its specific demands; it may reschedule periodic tasks in each period. For a dynamic algorithm like EDF, the upper bound of processor utilization is 100 percent. EDF is *optimal* in the sense that if a set of tasks can be scheduled by any priority assignment, it also can be scheduled by EDF.

When a single-processor machine with priority scheduling applies EDF to the scheduling of continuous-media data, process priorities are likely to be rearranged quite often. A priority-driven system scheduler like EDF assigns each task a priority according to its deadline. The highest priority is assigned to the task with the earliest deadline, the lowest to the one with the furthest. Common EDF systems usually provide only a restricted number of priorities. If EDF has already assigned the priority needed for a new process, the scheduler must rearrange the priorities of other processes until the required priority is free. In the worst case, the priorities of all processes have to be rearranged, which may cause considerable overhead.

An extension of EDF is the time-driven scheduler, which schedules tasks by deadlines instead of priorities. TDS handles overload situations by aborting tasks that can no longer meet their deadlines. If the situation continues, the scheduler removes tasks with a low "value density"—the importance of a task to the system.

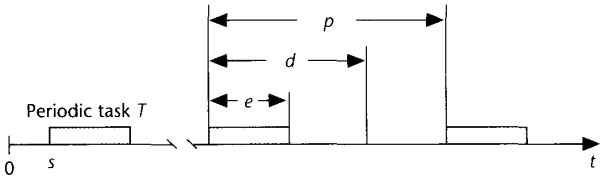


Figure 2. Any multimedia operating system must take into account the time constraints of the periodic task T, where s = starting point, e = processing time, d = deadline, and p = period.

Another priority-driven EDF scheduling algorithm[11] divides every task into a mandatory and an optional part. Tasks are scheduled with respect to the deadlines of their mandatory parts. A set of tasks is schedulable if all mandatory parts can meet the deadlines. A task is terminated according to the deadline of the mandatory part even if the task has not yet completed; the optional parts are then processed if the resource capacity is not fully utilized.

In the case of continuous-media data, this method can be combined with the encoding of data according to their importance. Take, for example, a single uncompressed picture in a bitmap format. Each pixel of this monochrome picture is coded with 16 bits. The processing of the eight most significant bits is mandatory, whereas the processing of the eight least significant bits can be considered optional. This method allows more processes to be scheduled. In an overload situation, the optional parts are aborted according to quality-of-service requirements (see "Negotiating QOS," below), decreasing the quality through media scaling. Alternatively, the user can intro-

duce QOS scaling parameter(s) that reflect the implementation. Overall, this approach avoids errors and improves system performance at the expense of media quality.

Rate monotonic algorithm

The rate-monotonic scheduling principle, introduced by Liu and Layland in 1973,[9] is a static algorithm applied in real-time systems and operating systems by the National Aeronautics and Space Administration and the European Space Agency. It assigns static priorities to tasks at the connection setup phase according to their request rates. Subsequently, each task is processed with the priority calculated at the beginning, with no further rearrangement of priorities required. The priority corresponds to the importance of a task relative to other tasks.

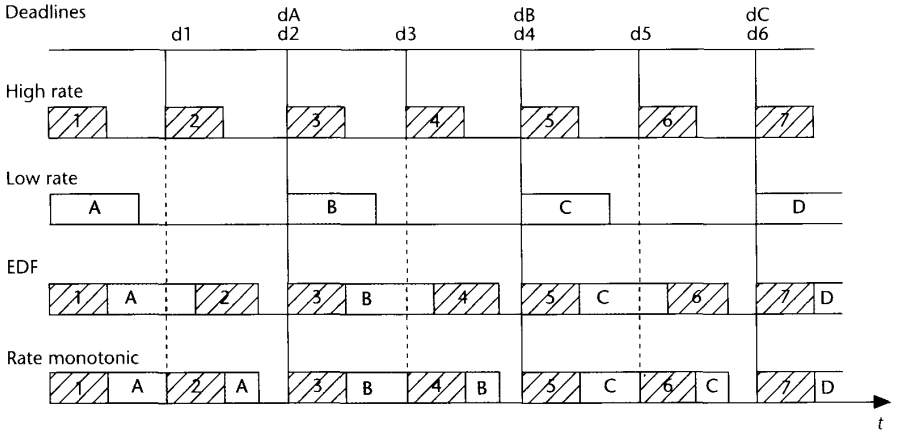


Figure 3. Of the preemptive schedulers, rate monotonic is more prone to context switching—changing over from processing video (lettered blocks) to processing audio (numbered blocks)—than EDF.

The task with the shortest period gets the highest priority, and the task with the longest period gets the lowest priority. It is an optimal, static, priority-driven algorithm for preemptive, periodic jobs. Optimal in this context means that no other static algorithm can schedule a task set that the rate monotonic algorithm cannot also schedule.

The following five assumptions are prerequisites to applying the rate monotonic algorithm:

1. The requests for all tasks with deadlines are periodic.
2. The processing of a single task must finish before the processing of the next task in the same data stream.
3. All tasks are independent. The requests of one task do not depend on the initiation or completion of requests for any other task.
4. Runtime for each request of a task—the maximum time a processor requires to execute the task without interruption—is constant.
5. Any nonperiodic task in the system has no required deadline. Typically such tasks initiate periodic or failure recovery tasks. They usually displace periodic tasks.

Further work has shown that not all of these assumptions are always mandatory to employ the rate monotonic algorithm.[11,12]

EDF versus rate monotonic

Consider an audio and a video stream scheduled according to the rate monotonic algorithm. Let the audio stream have a rate of 75 blocks of samples per second and the video stream a rate of 25 frames per second. The priority assigned to the audio stream is then higher than the priority assigned to the video stream. Messages arriving from the audio stream will interrupt the processing of the video stream, creating context switches.

If more than one stream is processed concurrently in a system, more context switches are likely with a scheduler using the rate monotonic algorithm than one using EDF, as Figure 3 shows.

The rate monotonic algorithm's processor utilization is upper bounded. The least upper bound is $U = \ln 2$, or about 69 percent.[9] Hence, we only need to check if the processor utilization is less than or equal to the given upper bound to find out whether a task set is schedulable. Most existing systems check this by simply comparing processor utilization to the value of ln 2.

On the other hand, EDF can achieve a processor utilization of 100 percent because all tasks are scheduled dynamically according to their deadlines. In practice, this 100 percent value is reduced by the need to provide processing power capabilities for interrupt handling, context switching, and other basic tasks. Figure 4 shows an example

of how the CPU can be utilized to 100 percent with EDF where rate monotonic scheduling fails.

The rate monotonic upper bound of 69 percent represents the worst-case execution time; calculations using that figure lead to underutilized processors. The problem of underutilizing the processor is aggravated by the fact that in most cases, the average task execution time is considerably lower than the worst case. Therefore, to use the processor as efficiently as possible, scheduling algorithms should be able to handle transient processor overload.

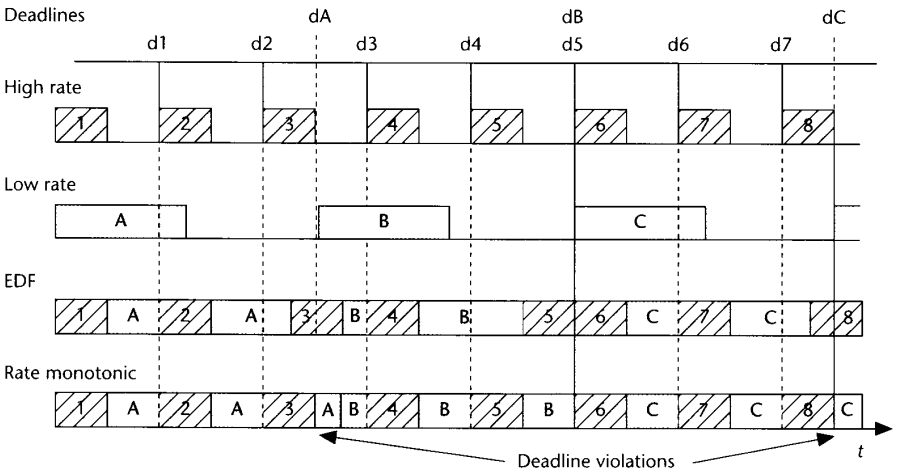


Figure 4. EDF can schedule tasks successfully at processor utilization rates up to 100 percent, while rate monotonic's utilization has a worst-case upper bound of 69 percent before failure.

On average, the rate monotonic algorithm ensures that all deadlines will be met, even if the bottleneck utilization is well above 80 percent. With one deadline postponement, the deadlines are met on average when the utilization exceeds 90 percent. The rate monotonic algorithm achieved a utilization bound of 88 percent for the Nowy's Inertial Navigation System.[13]

Applying the rate-monotonic algorithm

One extension to this algorithm divides a task into a mandatory and an optional part. Processing of the mandatory part delivers an acceptable result, while the optional part refines the result. The mandatory part is scheduled according to the rate monotonic algorithm, though different policies are suggested for scheduling of the optional part.[14]

Systems with aperiodic tasks next to periodic ones must be able to schedule both types of tasks. If the aperiodic request is a continuous stream, such as video images in a slide show, we can transform it into a periodic stream by substituting n items of minimal duration for each timed data item. The number of streams increases, but since the life span decreases, the result remains unchanged. The stream is now periodic because every item has the same life span.[15]

If the stream is not continuous, we can apply a sporadic server to respond to aperiodic requests. The server has a budget of computation time reserved for aperiodic tasks that is refreshed t units of time after it has been exhausted or earlier. The server may preempt the execution of periodic tasks only if the computation budget is not exhausted. Afterwards it can only continue to execute aperiodic tasks with a background priority. After the budget is refreshed, execution resumes at the server's assigned priority. The sporadic server is especially suitable for events that occur rarely but must be handled quickly, such as the movements of a telepointer in a computer-supported cooperative work (CSCW) application.[13]

The rate monotonic algorithm is particularly suitable for continuous-media data because it makes optimal use of their periodicity. Since it is a static algorithm, it rarely rearranges priorities and hence—in contrast to EDF—accrues no scheduling overhead to determine the next task. Problems emerge with data streams that have a very diverse processing time per message as MPEG-2 specifies, for example, a compressed video stream where one of five pictures is a full picture and all others are updates to that picture. The simplest solution is to schedule tasks according to their maximum data rate, which would decrease processor utilization. In any case, during the CPU's idle time all kinds of noncritical tasks can be processed.

Least-laxity-first algorithm

Besides EDF and rate monotonic, other scheduling algorithms have been evaluated for the processing of continuous-media data. The most prominent is least laxity first, which schedules the task with the shortest remaining laxity—the time between the current time and the deadline, minus the remaining processing time—first.[16] LLF is an

No scheduling technique is as suitable as EDF and rate monotonic.

optimal, dynamic algorithm for exclusive resources. It is also optimal for multiple resources if the ready times of the real-time tasks are the same.

Since laxity is a function of deadline, processing time, and current time, the processing time cannot be specified in advance. The calculation of laxity also assumes the worst case and is inexact. Moreover, the laxity of waiting processes changes over time. During runtime of one task, another task may get a lower laxity, causing it to preempt the running task. Consequently, tasks can preempt each other several times without a new task being dispatched, which can cause numerous context switches. At each scheduling point (either when a process becomes ready to run or at the end of a time slice) the laxity of each task must be determined anew, which creates a greater overhead than EDF. Since we usually have only a single resource to schedule, LLF has no advantage over EDF.

In future multimedia systems with multiple processors, LLF might look better. Most multimedia systems with preemptable tasks employ a variation of the rate monotonic algorithm. So far, no other scheduling technique has proven as suitable for multimedia data handling as the EDF and rate monotonic approaches.

Resource management

Multimedia systems with integrated audio and video processing often operate at the limit of their capacity, even with data compression and use of new technologies. Current computers do not allow any kind of manipulation and communication of these data according to their deadlines without reservations and real-time process management.

A multimedia system must enforce timing guarantees for continuous-media processing at every hardware and software component on the data path. Timing requirements depend on the type of media and the nature of the supported applications. For instance, a video image should not be presented late because the communication system was busy with a traditional discrete-data transaction. In any realistic scenario, we encounter several multimedia applications that access shared resources concurrently. Hence, even systems with high-bandwidth networks and huge processing capabilities require real-time mechanisms to guarantee data delivery.

In distributed multimedia systems, "resource management" covers several computers as well as communication networks. It allocates all resources involved in data transfer between sources and sinks. For instance, today a CD-ROM XA device has to be allocated exclusively: Transferring video data from the device takes up to 20 percent of the capacity of each CPU on the data path, up to 40 percent of the graphic processor's capacity, and a certain amount of network bandwidth. At the connection establishment phase, resource management ensures that the new connection does not violate performance guarantees already provided to existing connections.

Resources

We can extend the notion of resource management to cover the CPU (process management), memory management, the file system (file management), and device management. To simplify, I generalized the issue of reservation for all resources into a generic notion of resources.

A resource is a system entity that tasks require for manipulating data. Each resource has a set of distinguishing characteristics classified using the following scheme:[3]

- *Active or passive.* An active resource, such as the CPU or a network adapter for protocol processing, provides a service. A passive resource, such as the main memory, communication bandwidth, or a file system, denotes some system capability required by active resources.
- *Exclusive or shared.* Active resources are often exclusive; passive resources can usually be shared among processes.
- *Single or multiple.* A resource type that exists only once in the system is single, otherwise it is multiple. In a transputer-based multiprocessor system, an individual CPU is a multiple resource.

For example, an ISO 9660 file system stored on an optical disc in CD-ROM XA format is a passive, shared, single resource, while process management belongs to the categories of active, shared, and (most often) single resources.

Each resource has a capacity measured by a task's ability to perform in a given time span using the resource. In this context, capacity refers to CPU capacity, frequency range, or the amount of storage, for example. Real-time scheduling only

considers the temporal division of resource capacity among real-time processes.

QOS requirements

Each component of a multimedia system must fulfill the requirements of multimedia applications and data streams. Resource management maps these requirements onto system capacity. We can classify the transmission and processing requirements of local and distributed multimedia applications by four main characteristics:

1. throughput,
2. delay (local or global),
3. jitter, and
4. reliability.

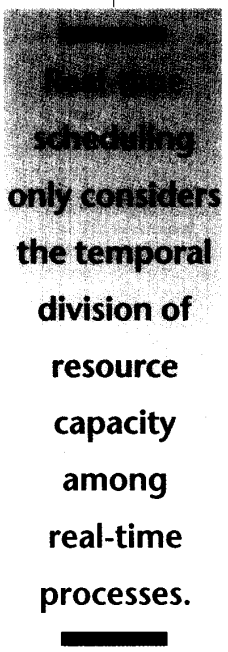
The *throughput* is determined by the data rate a connection needs to satisfy the application requirements, as well as the size of the data units. The *local delay* is the maximum time a resource takes to complete a certain task. The end-to-end, or *global*, delay is the total delay for a data unit traveling from a source to its destination. The *jitter*, or delay jitter, determines the maximum allowed variance in the arrival of data at the destination.

The *reliability* requirement defines error-detection and -correction mechanisms used for the transmission and processing of multimedia tasks. Errors can be ignored, indicated, or corrected. For instance, error correction through retransmission is rarely appropriate for time-critical data because the retransmitted data will usually arrive late. In such a case, a forward error-correction mechanism would be more useful. Reliability also refers to CPU errors caused by delays in processing a task which violate the demanded deadlines. In accordance with communication systems terminology, these requirements are also known as quality-of-service (QOS) parameters.

Negotiating QOS

A typical realization of resource allocation and management is based on the interaction between clients and resource managers. The client selects the resource and requests a resource allocation by indicating its requirements through a QOS specification. This is equivalent to a workload request. First the resource manager checks its own resource utilization and decides if the reservation request can be served. The resource manager stores all existing reservations to guarantee that each request receives its share of the resource capacity. This component also negotiates the reservation request with other resource managers if necessary.

During the connection establishment phase, the QOS parameters are usually negotiated between the requester (client application) and the addressed resource manager. In the simplest case, negotiation entails the resource manager checking whether the QOS parameters the application specified can be guaranteed. A more elaborate method is to optimize single parameters. In this case the application sets the values for two parameters, for example, throughput and reliability; the resource manager then calculates the best achievable value for the third parameter, delay. Negotiating the parameters for end-to-end connections over one or more computer networks requires resource reservation protocols like ST-II or RSVP. In such protocols, the resource managers of the single components within the distributed system allocate the necessary resources.



A resource manager and the individual schedulers provide services for the four phases of the allocation and management process:

1. *Schedulability test.* The resource manager checks whether, given the QOS parameters—throughput and reliability, for example—there is enough resource capacity to handle the additional request.
2. *Quality-of-service calculation.* The resource manager calculates the best possible performance—in this example, the lowest delay—the resource can guarantee for the new request.
3. *Resource reservation.* The resource manager allocates the capacity required to meet the QOS guarantees for each request.
4. *Resource scheduling.* Incoming messages are scheduled according to the given QOS guarantees. In process management, for instance, the scheduler allocates resources at the moment the data arrives for processing.

During the last phase, a scheduling algorithm is defined for each resource. The schedulability test, QOS calculation, and resource reservation depend on the algorithm used by the scheduler: Before making a reservation, you must know how much capacity you are allowed to distribute, as determined by the algorithm chosen (100 percent for EDF, 69 percent for rate monotonic).

In multimedia systems, file organization provides constant, timely data retrieval.

Allocation scheme

Resources can be reserved in either a pessimistic or optimistic way. The *pessimistic*, or guaranteed, approach avoids resource conflicts by making reservations for the worst case; that is, reserving resource bandwidth for the longest processing time and the highest rate a task might ever need. This can lead to underutilization of resources. In a multimedia system, however, discrete media tasks can use the remaining processor time—the time reserved for traffic but not used—and avoid wasting resource capacity. The pessimistic method results in a guaranteed QOS.

The *optimistic*, or statistical, approach reserves resources according to an average workload only, which could overbook resources. QOS parameters are met as far as possible. Resources are highly utilized, though an overload situation may result in failure. The optimistic approach, an extension of the pessimistic approach, requires a monitor to detect and solve resource conflicts. The monitor may, for instance, preempt processes according to their importance.

File management

The file system provides access and control functions for file storage and retrieval. From the users' viewpoints, the file system allows them to organize and structure files, changing how the files are represented externally. The internals are more important in our context: how the system represents information in files and how it accesses those files in secondary storage. In traditional file systems, the information types stored in files are sources, objects, program libraries and executables, numeric data, text, payroll records, and so on. In multimedia systems, stored information also includes video and audio, with real-time read and write demands that create additional requirements in the design and implementation of file systems.

Compared to the exponentially increased performance of processors and networks over the past decade, storage devices have become only marginally faster. The effect of this increasing disparity in speed is the search for new storage structures and storage and retrieval mechanisms. Applied to file systems, continuous-media data differs from discrete data in the following ways:

- *Real-time characteristics.* The retrieval, computation, and presentation of continuous media is time-dependent: The data must be presented (read) before a set deadline with minimal jitter. Thus, algorithms for the storage and retrieval of such data must consider time constraints and include additional buffers to smooth the data stream.

- *File size.* Compared to text and graphics, video and audio have very large storage space requirements. Since the file system has to store information ranging from small, unstructured units like text files to large, highly structured units like video and associated audio, it has to organize the data on disk in a way that efficiently uses the limited storage.

- *Multiple data streams.* A multimedia system has to support various media at once. It not only has to ensure that all of them get a sufficient share of the resources, it must also consider tight relations between streams arriving from different sources, such as the synchronized audio and video for a movie.

There are two basic approaches to supporting continuous media in file systems. In the first approach, the organization of files on disk remains as it is, with the necessary real-time support provided through special disk-scheduling algorithms and enough buffer capacity to avoid jitter. The second approach optimizes the organization of audio and video files, especially on distributed hierarchical storage like disk arrays. The basic idea is to improve the throughput and capacity by storing the data of each audio and video file on several volumes. Disk I/O bandwidth is maximized by striping, while seek times are minimized by grouping and sorting.[17]

Storage methods

In conventional file systems, the main goal of file organization is to use storage capacity efficiently (to reduce internal and external fragmen-

tation) and to allow arbitrary deletion and extension of files. In multimedia systems, however, the main goal is to provide constant, timely retrieval of data. Internal fragmentation occurs when blocks of data are not entirely filled. On average the last block of a file is only half utilized, so large blocks lead to a larger waste of storage. External fragmentation mainly occurs with contiguous storage. After deletion of a file, the resulting gap can only be filled by a file of the same size or smaller, leaving small, unused fractions between files. This leads to a dilemma: Either storage space for continuous media is wasted by internal fragmentation or huge amounts of data must be copied frequently to avoid external fragmentation.

Real-time storage. Providing an adequate buffer for each data stream and employing disk-scheduling algorithms optimized for real-time storage and retrieval of data offers flexibility at the cost of scattering the data blocks of single files. It also avoids external fragmentation and provides access to the same data by several streams via references. Even when using only one stream, this might be advantageous; for instance, the system could access the same block twice to play a repeating phrase in a sonata. However, even with optimized disk scheduling the data retrieval phase still requires large buffers to smooth jitter because of the large seek operations during playback. Therefore, this method can produce long initial delays at the retrieval of continuous media.

Another problem is the restricted transfer rate. Upcoming disk arrays, which might have 100 or more parallel heads, will achieve projected seek and latency times of less than 10 milliseconds, with a block size of 4 Kbytes at a transfer rate of 0.32 Gbits per second. However, this is still not enough for simultaneous retrieval of four or more production-level MPEG-2 videos compressed in HDTV quality, which may require transfer rates of up to 100 Mbps.[18,19]

Continuous storage. Approaches that use specific disk layout take the specialized nature of continuous-media data into account to minimize the cost of retrieving and storing streams. The much greater size of continuous-media files and the fact that they will usually be retrieved sequentially because of the nature of operations performed on them (such as play, pause, and fast forward) call for optimization of the disk layout.

Lougher and Shepherd's[20] application-related experience led them to two conclusions: (1) Continuous-media streams predominantly belong to the write-once-read-many category (WORM; see "Disk-scheduling algorithms," below), and (2) streams recorded at the same time are likely to be played back at the same time (for example, the audio and video of a movie). Hence, we should store continuous-media data in large data blocks contiguously on disk. Files likely to be retrieved together are grouped together on the disk, thus minimizing interference due to concurrent access of these files. With such a disk layout, the buffer requirements and seek times decrease.

The disadvantages of the contiguous approach are external fragmentation and copying overhead during insertion and deletion. To avoid these problems without scattering blocks in a random manner over the disk, a multimedia file system can provide constrained block allocation of the continuous media.[21] To serve the continuity requirements during allocation, the file system should introduce read-ahead and buffering of a determined number of blocks.[22]

Interleaved placement. Some systems using scattered storage employ a special disk-space allocation mechanism for fast and efficient access. Abbott performed the pioneer work in this field.[23] He was especially concerned about the size of single blocks, their positions on disk, and the placement of different streams. With interleaved placement, all nth blocks of each stream are in close physical proximity on disk. Two possibilities for interleaved placement are contiguous and scattered. With interleaved data streams, synchronization is much easier to handle. On the other hand, inserting and deleting single parts of data streams become more complicated.

We can categorize disks by information storage and recording methods.

Disk-scheduling algorithms

In general, disks can be characterized in two different ways. The first is how they store information: There are rewritable disks, WORM disks, and read-only disks like CD-ROMs. The second distinctive feature is the recording method, either magnetic or optical. The main differences between the methods are the access time and the track capacity: The seek time on magnetic disks is typically about 10 ms, whereas on optical disks 200 ms is still a common lower bound. Magnetic disks have a constant rotation speed, or constant angu-

Scheduling algorithms must find a balance between time constraints and efficiency.

lar velocity (CAV). Thus, while the density varies, the storage capacity is the same on inner and outer tracks. Optical disks have varying rotation speed, or constant linear velocity (CLV), so the storage density is the same on the whole disk while the capacity varies. The different recording methods mean that magnetic and optical disks make use of different algorithms. File systems on CD-ROMs are defined by ISO 9660, with very few variations allowed. Hence, I will focus on algorithms applicable to magnetic storage devices.

The overall goal of disk scheduling in multimedia systems is to meet the deadlines of all time-critical tasks. Closely related is the goal of keeping the necessary buffer space requirements low. As many streams as possible should be served concurrently, but aperiodic requests should also be schedulable without delaying service for a large amount of time. The scheduling algorithm has to find a balance between time constraints and efficiency.

The EDF strategy. Let us first look at the EDF scheduling strategy, described above for CPU scheduling but used in file systems as well. In the context of file systems, EDF would read the block of the stream with the nearest deadline first. The employment of strict EDF results in poor throughput and an excessive seek time. Further, since EDF is most often applied as a preemptive scheduling scheme, the costs for preempting one task and scheduling another are high. The overhead caused by this is on the same order of magnitude as at least one disk seek. Hence, a file system must adapt EDF or combine it with other file system strategies.

The SCAN-EDF strategy. This combines the seek optimization of the well-known traditional disk-scheduling method SCAN and the real-time guarantees of the EDF mechanisms in the following way:[24] As in EDF, the request with the earliest deadline is always served first. Among requests with the same deadline, the one that is first according to the scan direction is served first. The SCAN algorithm repeats this principle until no request with this deadline is left.

Since the optimization only applies for requests with the same deadline, its efficiency depends on how often it can be applied—that is, how many requests have the same or a similar deadline. The following trick can increase efficiency: Since all requests have release times that are multiples of the period p, all requests have deadlines that are multiples of the period p. Therefore the requests can be grouped together and served accordingly. Requests with different data-rate requirements can supplement SCAN-EDF with a periodic fill policy to let all requests have the same deadline.

SCAN-EDF can easily be implemented by slightly modifying EDF. If D_i is the deadline of task i and N_i is the track position, then the deadline can be modified to be $D_i + f(N_i)$. The function $f()$ converts the track number of i into a small perturbation that defers the deadline. Compared to pure EDF and different variations of SCAN, SCAN-EDF with deferred deadlines performs well in multimedia environments.[24]

Group sweeping strategy. This variation of SCAN, serves requests in round-robin cycles.[25] To reduce disk arm movements, GSS divides the set of n streams into g groups, served in fixed order. Individual streams within a group are served according to SCAN; therefore the time or order of individual streams within a group is not fixed. In one cycle a specific stream may be the first served, but in another cycle it may be the last in the same group. A smoothing buffer, sized according to the cycle time and data rate of the stream, assures continuity. Since the data must be buffered in GSS, the playout can start at the end of the group in which the first retrieval takes place. Whereas SCAN requires buffers for all streams, GSS can reuse the buffer for each group. GSS is a trade-off between optimizations of buffer space and arm movement.

To provide the requested guarantees for continuous-media data, we can introduce a joint deadline mechanism: We assign to each group of streams one deadline, the *joint deadline*. This deadline is the earliest deadline of all streams in the group. Streams are grouped in such a way that all of them have similar deadlines.

Mixed strategy. Abbott introduced a mixed strategy based on the shortest seek strategy (also called *greedy* strategy) and the balanced strategy.[23] Every time data are retrieved from disk, they are transferred into buffer memory allocated for the data stream. From there the application process retrieves the data. The balanced strategy attempts to maximize transfer efficiency by minimizing seek time and latency and to serve process require-

ments with a limited amount of buffer space.

While shortest seek serves the process whose data block is closest to the disk head first, thus saving seek time, the balanced strategy serves the process with the least amount of buffered data first. The crucial part of the mixed algorithm is deciding which of the two strategies to apply. For shortest seek, two criteria must be fulfilled: The number of buffers for all processes should be balanced (that is, all processes should have nearly the same number of buffered data), and the overall required bandwidth should be sufficient for the number of active processes so that none of them will try to immediately read data out of an empty buffer.

Abbot introduced the term *urgency* in an attempt to meet both criteria.[23] This number measures both the relative balance of read processes and the number of them. If the urgency is large, the balanced strategy is best; if it is small, it is safe to apply the shortest seek algorithm.[26]

Device management

Device management and access allows the operating system to integrate all hardware components. The physical device is represented by an abstract device driver, which hides its physical characteristics. In a conventional system such devices include a graphics adapter card, hard disk, keyboard, and mouse. Multimedia systems add devices like cameras, microphones, speakers, and dedicated audio and video storage devices. Yet in most existing multimedia systems, such devices are seldom integrated by device management and the respective drivers.

Addressing of a camera can be handled much like addressing of a keyboard. Existing operating system extensions for multimedia usually provide one common system-wide interface for the control and management of data streams and devices. In Windows and OS/2, this interface is known as the Media Control Interface (MCI). The multimedia extensions of Windows, for example, provide the following classes of function calls:

- *System commands* are served by a central instance, not forwarded to the single device driver (MCI driver). An example of such a command is the query concerning all devices connected to the system, "Sysinfo."

- *Compulsory commands* include the query for specific characteristics ("capability info") and the opening of a device ("open"). Each device driver must be able to process them.

- *Basic commands* refer to characteristics that constitute all devices. To process such a command, a device driver must consider all variants and parameters of the command. A data transmission, for example, is typically started by the basic command "play."

- *Extended commands* may refer to both device types and special single devices. The "seek" command for the positioning on an audio CD is an example.

Synchronization

Synchronization denotes the temporal relationship between different media data. A typical example is lip synchronization, which requires a tight temporal relationship between audio and video data. Most often this type of synchronization is guaranteed and enforced by having audio and the related video data stored and transmitted in an interleaved way defined at the MPEG system layer. Otherwise, time-stamping of the media packets (LDUs) and appropriate buffering at the presentation system allows the operating system to present the related data units of the different streams "in synch" to the user.

Addressing of a camera can be handled much like addressing of a keyboard.

Memory management

The memory manager assigns physical resource memory to a single process. Virtual memory is mapped onto available actual memory. The memory manager swaps less frequently used data between main memory and external storage using *paging*. Pages are transferred back into main memory when a process requires data on them. Note that continuous-media data must not be temporarily paged out of main memory. If a page of virtual memory containing code or data required by a real-time process is not in real memory when the process accesses it, a page fault occurs, meaning the page must be read from disk. Page faults seriously affect the real-time performance, so they must be avoided. One approach is to lock code and/or data into real memory. However, take care: Real memory is a very scarce resource to the system. Committing real memory by pinning (locking) will decrease overall system performance. For example, the typical AIX kernel will not allow more than about 70 percent of real memory to be committed to pinned pages.

The transmission and processing of continuous data streams by several components require very efficient data transfer restricted by time constraints. Memory allocation and release functions provide well-defined access to shared memory areas. Most cases require no real data processing, only a data transfer. For example, say a digital camera is the source and the presentation process is the sink. The essential task of the other components is the exchange of continuous-media data with relatively high data rates in real time. Processing involves computing, adding, interpreting, and stripping headers. The actual implementation can be realized either with external devices and dedicated hardware in the computer or with software components.

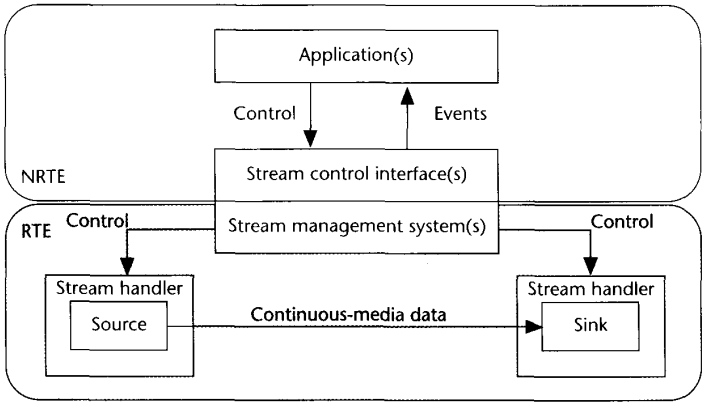


Figure 5. Within one multimedia computer, real-time and non-real-time environments use specialized architectures to meet differing data requirements.

Early prototypes of multimedia systems incorporated audio and video based on external data paths only. Memory management, in that case, merely controlled an external switch. A first step towards integration was incorporation of the external switch function into the computer by employing dedicated adapter cards that can switch data streams with varying data rates. Today, complete integration achieves a fully digital approach within the computer—a pure software solution. Data is transmitted between the single components in real time. Copy operations are reduced as far as possible to the exchange of pointers and the check of access rights, which requires access to a shared address space. Data can also be transferred directly between different adapter cards. The transfer of continuous-media data takes place in a real-time environment. This exchange is controlled but not necessarily executed by the application.

System architectures

The employment of continuous media in multimedia systems leads to new system architectures. A typical multimedia application does not require the application itself to process audio and video. Data is obtained from a source, such as a microphone, camera, disk, or network, and forwarded to a sink, such as a speaker, display, or network. The requirements of continuous-media data are satisfied best if the data takes the shortest possible path through the system by copying data directly from adapter to adapter. The program then merely sets the correct switches for the dataflow by connecting sources to sinks. Hence, the application itself never really touches the data, unlike in traditional processing.

A problem with direct copying is losing control over QOS parameters and device-specific headers and trailers. In multimedia systems, such an adapter-to-adapter connection is defined by the capabilities of the two adapters involved and the bus performance. An MPEG-2 program stream contains several layers, each with headers and trailers, whereas a communication protocol on the network adapter contains more information about the actual payload. Hence the multimedia application opens devices, establishes a connection between them, starts the dataflow, and returns to other duties.

As previously stated, the overriding need of multimedia applications is to meet temporal requirements at presentation time. Therefore, multimedia data is handled in a real-time environment: Its processing is scheduled according to inherent timing requirements of multimedia data. On a multimedia computer, the real-time environment will usually coexist with a non-real-time environment (NRTE), which deals with all data without timing requirements (Figure 5). Multimedia I/O devices in general can be accessed from both environments. A video frame, for example, is passed from the RTE to the display. The establishment of communication connections at the start of a stream does not need to obey timing requirements, but the data processing for established connections does.

All control functions are performed in the NRTE. The application usually only calls these control functions and does not actively handle continuous-media data. Therefore the multimedia application itself typically runs in the NRTE, shielded from the RTE.

In some scenarios, users may want applications to process continuous-media data in an applica-

tion-specific way. In this model, such an application comprises a module running as a stream handler in the RTE, while the rest of the applications run in the NRTE using the available stream control interfaces. (Stream handlers are all entities in the RTE in charge of multimedia data. Typical stream handlers are filter and mixing functions, but parts of the communication subsystem can be treated in the same way.) System and application programs such as communication protocol processors use this programming in the RTE. While applications like authoring tools and media presentation programs are relieved from the burden of programming in the RTE, they interface with and control the RTE services. An application determines processing paths and controls devices and paths to meet its data processing needs by defining the sinks, sources, and quality of service requested.

To reduce data copying, the RTE employs buffer management functions to implement data transfer. This buffer management is located between the stream handlers. Each stream handler has endpoints through which data units flow. The stream handler consumes data units from one or more input endpoints and generates data units through one or more output endpoints.

Applications access stream handlers by establishing sessions with them. Depending on the required QOS of a session, an underlying resource management subsystem multiplexes the capacity of the underlying physical resources among the sessions. NRTE control operations manage the RTE dataflow through the stream handlers. These functions make up the stream management system in the multimedia architecture. Some operations are provided by all stream handlers, such as operations to establish sessions and to connect their endpoints, and some operations are specific to an individual stream handler (they usually determine the content of a multimedia stream and apply to particular I/O devices).

The stream management subsystem specifies stream synchronization on a connection basis, expressed using the notions of clock or logical time systems. It determines points in time at which the processing of data units shall start. Regular streams can use the stream rates or sequence numbers to relate data units to synchronization points. Time stamps are a more versatile means for synchronization, as they can also be used for nonperiodic traffic. Synchronization is often implemented by delaying the execution of a thread or by delaying the receive operation on a buffer exchanged between stream handlers.

Conclusion

Scheduling concerns are paramount in multimedia systems. The need to deliver continuous media on time while accommodating discrete data informs every aspect of a multimedia operating system. It affects how the system manages processes, resources, files, and memory. The concepts employed by current multimedia operating systems were initially developed for real-time systems and were adapted to the requirements of multimedia data. Today's operating systems incorporate these functions either as device drivers or as extensions based on the existing operating system scheduler and file systems. The next step will bring an integration of real-time processing and non-real-time processing in the native system kernel.[27,28] **MM**

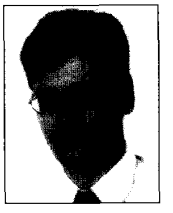
Acknowledgments

The author gratefully acknowledges the valuable advice and the work performed by Andreas Mauthe in this context. Klara Nahrstedt, Lars Wolf, Ian Marsh, and the anonymous reviewers suggested many improvements to all sections which were incorporated into this final version.

References

1. T. Burkow, "Operating System Support for Distributed Multimedia Applications: A Survey of Current Research," *Memoranda Informatica* 94-48, Univ. of Cambridge Computer Laboratory and Univ. of Twente Faculty of Computer Science, 1994.
2. R. Steinmetz, *Multimedia Technology: Fundamentals and Introduction* (in German), Springer-Verlag, Berlin, 1993.
3. R. Steinmetz and K. Nahrstedt, *Multimedia: Applications, Computing, and Communications*, Prentice-Hall, to be published in May 1995.
4. C.M. Krishna and Y.H. Lee, "Real-Time Systems," *Computer*, Vol. 24, No. 5, May 1991, pp. 10-11.
5. R. Steinmetz, "Human Perception of Jitter and Media Skew," to be published in *IEEE J. Selected Areas in Comm.*, Vol. 14, No. 1, Jan. 1996.
6. A.K. Mok, *Fundamental Design Problems of Distributed Systems for the Hard Real-Time Environment*, doctoral dissertation, Dept. of Electrical Eng. and Computer Science, MIT, 1993.
7. A.M. Van Tilborg and G.M. Koob, eds., *Foundations of Real-Time Computing: Scheduling and Resource Management*, Kluwer Academic Publisher, Norwell, Mass., 1991.
8. R. Nagarajan and C. Vogt, "Guaranteed Performance of Multimedia Traffic over the Token Ring," Tech. Report No. 439201, IBM-ENC, Heidelberg, 1992.

9. C.L. Liu and J.W. Layland, "Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment," *J. ACM*, Vol. 20, No. 1, Jan. 1973, pp. 46-61.
10. S.-C. Cheng, J.A. Stankovic, and K. Ramamritham, "Scheduling Algorithms for Hard Real-Time Systems—A Brief Survey," *Hard Real-Time Systems*, J. A. Stankovic and K. Ramamritham, eds., IEEE CS Press, Los Alamitos, Calif., 1988, pp. 150-178.
11. J.W.S. Liu et al., "Algorithms for Scheduling Imprecise Computations," *Computer*, Vol. 24, No. 5, May 1991, pp. 58-68.
12. L. Sha, M.H. Klein, and J.B. Goodenough, "Rate Monotonic Analysis for Real-Time Systems," in *Foundations of Real-Time Computing: Scheduling and Resource Management*, A. van Tilborg and G.M. Koob, eds., Kluwer Academic Publisher, Norwell, Mass., 1991, pp. 129-156.
13. B. Sprunt, L. Sha, and J. Lehoczky, "Aperiodic Task Scheduling for Hard Real-Time Systems," *J. Real-Time Systems*, Vol. 1, 1989, pp. 27-60.
14. J.W.S. Liu, K.-J. Lin, and S. Naturajan, "Scheduling Real-Time, Periodic Jobs Using Imprecise Results," *Proc. IEEE Real-Time Systems Symp.*, IEEE Press, Piscataway, N.J., 1987, pp. 252-260.
15. R.G. Herrtwich, "Time Capsules: An Abstraction for Access to Continuous-Media Data," *Proc. IEEE Real-Time Systems Symp.*, IEEE Press, Piscataway, N.J., 1990, pp. 11-20.
16. D.W. Craig and C.M. Woodside, "The Rejection Rate for Tasks with Random Arrivals, Deadlines, and Preemptive Scheduling," *IEEE Trans. on Software Eng.*, Vol. 16, No. 10, Oct. 1990, pp. 1,198-1,208.
17. P.M. Chen et al., "RAID: High-Performance, Reliable, Secondary Storage," *ACM Computing Surveys*, Vol. 26, No. 2, June 1994, pp. 145-186.
18. R. Steinmetz, "Data Compression in Multimedia Computing: Principles and Techniques," *Springer/ACM J. Multimedia Systems*, Vol. 1, No. 4, Feb. 1994, pp. 166-172.
19. R. Steinmetz, "Data Compression in Multimedia Computing: Standards and Systems," *Springer/ACM J. Multimedia Systems*, Vol. 1, No. 5, Mar. 1994, pp. 187-204.
20. P. Lougher and D. Shepherd, "The Design of a Storage Service for Continuous Media," *The Computer J.*, Vol. 36, No. 1, Feb. 1993, pp. 32-42.
21. J. Gemmell and S. Christodoulakis, "Principles of Delay Sensitive Multimedia Data Storage and Retrieval," *ACM Trans. on Information Systems*, Vol. 10, No. 1, Jan. 1992, pp. 51-90.
22. H.M. Vin and P.V. Rangan, "Techniques for Efficient Storage of Digital Video and Audio," *Computer Comm.*, Vol. 16, No. 3, Mar. 1993, pp. 168-176.
23. C. Abbott, "Efficient Editing of Digital Sound on Disk," *J. Audio Eng. Soc.*, Vol. 32, No. 6, June 1984, pp. 394-402.
24. A.L.N. Reddy and J. Wyllie, "Disk Scheduling in a Multimedia I/O System," *Proc. 1st ACM Int'l Conf. on Multimedia*, ACM Press, New York, 1993, pp. 225-233.
25. M.-S. Chen, D.D. Kandlur, and P.S. Yu, "Optimization of the Group Sweeping Scheduling (GSS) with Heterogeneous Multimedia Streams," *Proc. 1st ACM Int'l Conf. on Multimedia*, ACM Press, New York, 1993, pp. 235-241.
26. R. Steinmetz, "A Multimedia File Systems Survey: Approaches for Continuous Media Disk Scheduling," to appear in *Computer Comm.*, Vol. 18, No. 4, Apr. 1995.
27. S.J.Mullender, "Kernel Support for Distributed Systems," in *Distributed Systems*, S.J. Mullender, ed., Addison-Wesley, Reading, Mass., 1993, pp. 385-409.
28. S.J. Mullender, "Operating System Support for Distributed Multimedia," *Usenix Summer Conf. 1994*, Usenix Assoc., pp. 209-220.



Ralf Steinmetz manages the multimedia department at the IBM European Networking Center in Heidelberg. He is also a lecturer in distributed multimedia systems at the University of Frankfurt. He studied electrical engineering with a focus on communications at the University of Salford, UK and Darmstadt, Germany, and received an MS (Dipl.-Ing.) and PhD (Dr.-Ing.) from the University of Darmstadt in 1982 and 1986, respectively. He is an associate editor-in-chief for *IEEE MultiMedia*. His current research interests include multimedia issues in distributed systems and applications.

Contact the author at IBM European Networking Center, Vangerowstrasse 18, D-69115, Heidelberg, Germany, e-mail steinmetz@vnet.ibm.com.