CPU Scheduling in Multimedia Operating Systems

Daniel Alexander Taranovsky

Research Report

1999

Introduction

Relation to CPU Scheduling: In order to understand the importance of CPU scheduling in multimedia applications, it is useful to place the issue in context. An overview of multimedia is given, mentioning some characteristics of multimedia and their associated problems. Given this information, one gets a better understanding of multimedia as a whole, and will contribute to the understanding of multimedia CPU scheduling.
References: [1], [2], [4]

1.0

Multimedia applications have unique requirements that must be met by network and operating system components. There is extensive research in developing network and operating systems to meet the demands of multimedia computation. Certain problem solutions are exclusive to the operating system, some are unique to network research, and some problems cover both domains. In some research papers I surveyed, multimedia solutions are often discussed in the context of a specific component or system feature. For example, an enhanced network protocol to handle multimedia traffic, or storage systems for video data. Other research deals with the interrelation of components in a multimedia system, or the enhancement of an existing system to better handle multimedia applications. Research in multimedia spans not only the development of new system solutions, but evaluating existing systems as well. Some research papers attempt to prove empirically how useful or impractical a system may be for executing multimedia applications. Some research deals not with the evaluation of particular systems, but methods themselves for evaluating them.

Multimedia data demands strict time constraints for processing. In any multimedia application, we may have several processes running dependently on one another. For example, one process may generate video frames for an X-window process while another process generates an audio stream for an attached speaker system. These two processes must execute in parallel for the application to be of any worth. In other words, the processes require relative progress to one another. It is of no use to begin executing the audio process once the video is half finished. Certain media processes may require absolute time progress as well. For example, the video application should process frames at a constant rate with respect to world time. If steady absolute progress is not enforced, one would observe random stopping and starting of the video. If relative progress is not enforced, cooperating processes such as the audio and video application mentioned earlier will not function properly.

There is some discussion on how multimedia time dependencies should be expressed within an application. Some dependencies may be implied (e.g. live video sequences should be processed at constant intervals) or may be explicitly defined (e.g. a pre-formulated slide show). In either case, the multimedia system must have mechanisms to support it. A significant amount of research focuses on respecting time requirements (i.e. time-fault avoidance -- the topic is discussed in following sections). Research in time-fault avoidance covers everything from proper scheduling of resources to I/O hardware. Mechanisms for dealing with missed deadlines should be implemented (i.e. time-fault handling). Some research proposes simply discarding data that was not processed in time, while others choose to accept that not all deadlines will be met and processes the data despite its late arrival. In a multimedia system, one tries to overcome latencies

associated with storage, communication, and computation. These factors are responsible for random, frequent delays that can be inappropriate for multimedia applications, if not fatal. In addition, the problem is further complicated in distributed systems since data may be arriving from independent sources and require synchronization, despite the asynchronous nature of the network. Data may be retrieved from a remote file server, or computation may be executed on a remote processor. This increases the probability of delays in acquiring multimedia data that has strict time deadlines to respect. An important part of the solution is careful scheduling of I/O, storage, communication resources, and CPU cycles.

Multimedia can be classified as live-data applications or stored-data applications. Live-data is much harder to process effectively because there can be little or no data buffering to ensure consistent output. For live-data, displaying audio and video as it happens reduces the amount of slack time allowed for computation and resource scheduling. Live-data is simply more demanding in its temporal deadlines than stored multimedia data. Stored-data can be retrieved in bulk well in advance of output deadlines. This ensures data will be available most of the time for processing when required.[2]

Multimedia applications may wish to be implemented in a computer system, but not at the expense of functionality. Users may want other classes of applications to be able to run correctly. Batch class applications are computationally intensive processes such as compilers. Throughput is the most important performance feature for this type of computation. Interactive class applications are programs with frequent I/O bursts such as text editors and interface processes. Response time is the most important performance feature for this type of computation. Continuous media class applications (which includes multimedia) are processes that are time dependent. Video players are an example of such an application. Real-time processing is essential for such applications to run effectively. These three applications' priorities may conflict at times. It is the operating system's responsibility to effectively handle various classes of applications at the same time. This is difficult and is an ongoing topic in multimedia research.[1]

The problem of running a movie player on a workstation is not simply a matter of providing sound and video support. While I/O has risen to the challenges posed by multimedia (e.g. CD-ROMs and DVD technology), operating systems and programming languages have been lagging. The underlying problem with today's operating systems is its handling of I/O. I/O has been treated as a series of unrelated data movement operations on a set of independent devices. For example, each device driver in UNIX-based operating systems is left to coordinate one device queue independent of each other. Typically this is done on a first-come, first-serve basis. With this design, attempting to coordinate input from several devices (e.g. a video camera and a microphone) is impossible as an operating system primitive. The devices are unable to coordinate themselves with other devices to effectively schedule themselves. This is further complicated in distributed systems where devices and storage can be located at different geographical locations. Bulterman claims no amount of hacking existing operating systems will result in a comprehensive solution to processing time-based data.[4]

System Characteristics

Relation to CPU Scheduling: CPU scheduling is one of several system components that make a whole multimedia system. Scheduling problems of resources in general has significance in understanding CPU scheduling. An overview of computer systems as a whole puts the CPU in the context of a resource, which has scheduling and performance issues just like memory or I/O.
References:[3]

2.0

Multimedia is a real-time application. This implies a certain amount of data must be handled within a specified time frame. Multimedia requires a tremendous amount of resources to accommodate. Audio and video files consume large space on secondary storage. Computing and processing multimedia files requires many CPU cycles and efficient I/O. Some argue today's workstations and networks do not meet the requirements of current multimedia applications. Essentially, they cannot ensure consistent, on-time data delivery. There are three reasons for this:

- Capacity of system resources is too low (performance is too low)
- The existing resources are not assigned to tasks efficiently (resource scheduling is poor).
- Access to resources is not controlled properly to avoid conflict (resource reservation is insufficient).

Herrtwich argue all three are interrelated. This implies deficiencies in one area can be overcome by improvements in other areas. It also implies although each area on its own may not be adequate for multimedia computation, the overall system may be if all three areas are properly managed.

Performance is naturally essential for multimedia computation. If processor performance is too low, neither proper scheduling nor resource reservation will help the system cope. Today's multimedia presents tremendous amounts of data for the system to handle. In the recent past resources were not available or too costly to handle such computation. Today's high-speed RISC architectures and CD-ROMs provide the opportunity to take advantage of multimedia.

Anderson wrote a historical perspective on computer system performance. Anderson categorizes computer systems as having abundant resources (computation is not a problem), sufficient but scarce resources (careful management of resources is required to achieve acceptable performance), or insufficient resources (processors or peripherals can not cope). The three categories are mapped to particular types of computation. The computation categories range from simple typing processes to multimedia. Anderson believes today's economical computer systems fall in the category of "sufficient but scarce resources" with respect to multimedia. In other words, the system lacks the kind of brute-force computation power required to handle multimedia without clever algorithm support. Today's systems have resources capable of handling multimedia applications, but require careful management. For example, some workstations experience gaps in audio output while the mouse is moved or keyboard used.

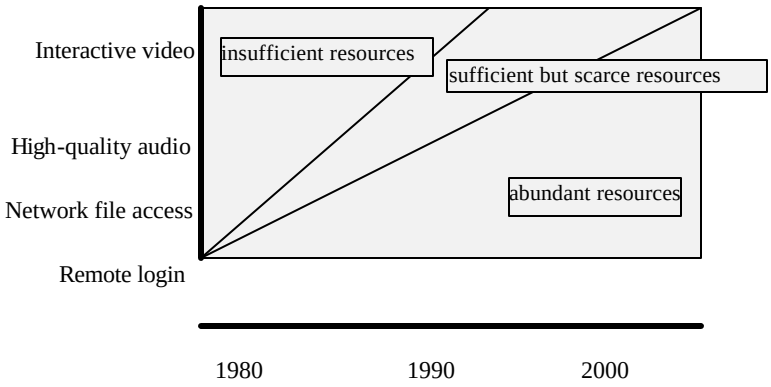


Figure 1.

Scheduling implies multiplexing a resource among several tasks to ensure all throughput requirements are met. For batch or interactive processes, secondary storage or CPU cycles can be scheduling using a round robin or first-come first-serve policy. These algorithms are not careful with respect to real-time but stress fairness. For real-time tasks (like multimedia), a more appropriate scheduling algorithm should be chosen. Earliest-deadline-first is the popular choice for many resources shared by real-time processes. Deadline scheduling chooses the optimal schedule for scenarios where one instance of the resource exists. The algorithm is popular because computation is fast and the amount of information required is small (the next deadline of each process is the only information required by the scheduler).

Some systems allow users to specify priorities to processes. A variation of priority scheduling is a two-tier system where real-time processes are handled separately and in priority of conventional computation. This idea is known as mixed scheduling and has several problems associated with it. If real-time processes are always selected in priority of conventional processes, conventional computation can starve. There is no general solution to handle these situations and is an on-going topic in research.

Resource reservation is important to implement a multimedia system. This implies the workload for which the resources were reserved is always schedulable. This may sound trivial, but is essential for efficient multimedia computation. Real-time processes cannot wait for resources to become available. Batch programs on the other hand, can simply wait several seconds for other processes to finish before resuming computation. This introduces another problem, however. An efficient algorithm must be developed for reserving resources. This problem is similar to the deadlock-avoidance problem, where deadlocked resource contention is avoided by attempting to estimate the amount of resources each process may use. In the context of real-time processing however, the goal is to ensure each time-dependent process can run when scheduled by reserving all the resources it will need at the start of execution. An optimistic estimation (reserving the minimum number of resources) may result in some processes being forced to wait for resources. This will result in degraded performance. A pessimistic estimation (reserving the maximum amount of resources) may result in some processes being refused to begin execution until the resources become available. However, greater quality of service is ensured for the few processes that are permitted to execute. There is a trade-off between performance and CPU utilization. Estimating the amount of resources required by a certain process is far from a trivial task in itself.[3]

Thread Models

Relation to CPU Scheduling: A thread is the basic schedulable unit by the CPU. Multimedia and other real-time processes require unique thread design to help the CPU be effectively scheduled. Multimedia has unique scheduling criteria that must be considered in thread architectures.
References:[5], [6]

3.1 Real-Time Threads

Tokuda and Kitayama proposed a real-time thread model that specifies thread timing attributes. *Periodic threads* are a type of real-time thread with timing attributes are characterized by a start time 'S', period 'T', and deadline 'D'. In Tokuda and Kitayama's model of a periodic thread, each thread is defined by timing attributes and a function f ( ). At the end of the function 'f()' execution, the thread will restart and begin executing again at the next period time boundary 'T'. It is assumed the function will be completed before the end of the period. Otherwise, the deadline will have been missed and a time-fault will occur. How the thread responds to a time-fault largely depends on the application.

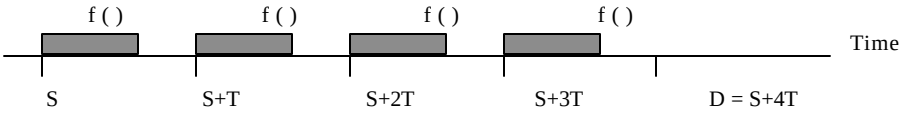


Figure 2.

```
R_Time_Thread ( ) {
//Model of real-time thread creation

   thread_id f_id;                  //defined thread id
   f_attr = {f, S, T, D};           //thread attributes and function name 'f'
  thread_create (f_id, f_attr);     //create 'f' thread
   }


f (args){
//function to be executed by a real-time thread. The real-time thread contains a pointer to
//this function.

   f's body;
   }
```

Figure 3. Pseudocode Thread Declaration

A good example implementing the above model would be a video process which starts at some time S, processes a new frame for delivery to an X-window server every period T, and is scheduled to finish at time D. Hence, the function 'f'()' would simply read in the next video frame from storage, decompress and fragment the image, and pass it to a buffer for the X-window server to process. This process should be executed within time T, and will be repeated at the beginning of every new period.

There is a problem if a process' deadlines cannot be met. Tokuda and Kitayama identify two reasons for time faults to occur. First, lack of real-time support from operating systems to provide better time-driven resource management. Second, a lack of resource management where the system can guarantee a requested level of service. This relates back to the paper by Herrtwich that clarifies the relationship between performance, resource reservation, and resource scheduling. As stated by Herrtwich, performance deficiencies are another reason why deadlines can be missed (which Tokuda did not mention). If the video frames are too large or complex to be rendered within a period 'T', the deadline will be missed. If several video processes are executing at once, the cumulative computation to render all frames may be too much for the processor to handle within defined time constraints. Both these examples are cases of CPU overloading, and are the direct result of lacking CPU performance.

Tokuda and Kitiyama's model of real-time threads has implications on scheduling CPU cycles. This model states explicitly how often a particular function will need to be executed. Based on this, one can estimate how many CPU cycles will be consumed during the life of the process. A long-term scheduler may use this information to determine if enough resources are available to run the process. The model also gives the short-term scheduler a good mechanism for determining the respective deadlines of the processes in the runnable queue. This is particularly useful if the scheduler implements the earliest-deadline-first or similar algorithm.

Assume a real-time thread is executing and its periodic deadline is missed. This occurs when the execution time of a thread's function 'f()' is greater than the period 'T'. Missing a deadline is known as a "time-fault". Time faulting is more serious for some applications than others, and the consequences of a time-fault are application dependent. A time-fault handler is invoked by the operating system that determines what action should be taken. If continuing execution is pointless after missing one deadline, the handler may abort the process. The handler may choose to simply ignore the missed deadline. This is useful for video applications, for example, where missing a few frames goes practically unnoticed by the viewer. The handler must also decide to discard the missed frame or process it late. Again, this depends on the application. Time-fault handling is discussed in further detail with quality of service.

The priority of the time fault handler is an issue. If the handler is long and is given priority over other real-time threads, it may cause other threads to miss their deadlines and invoke further time-fault handlers. Tokuda and Kitayama choose to simply suspend the faulting process and leave the handler priority up to the user to define. For their experiments this solution is feasible. However, other systems implement more sophisticated features. Herrtwich suggests a two-tier fault handling system where the first handler is implemented in hardware and determines the priority of the schedulable second handler.[5]

3.2 User-Level Threads

Tokuda and Oikawa developed a method of implementing user-level real-time threads. The benefits of user-level threads are efficiency and control. One can control the timing attributes of the threads without kernel intervention to efficiently adjust quality of service. A user-level scheduler must be implemented to schedule user-level threads. However, there is no costly context switch associated with scheduling new threads for execution. User-level threads improve CPU utilization at the cost of functionality (I assume the reader is familiar with user threads).

In order to implement user-level threads, Tokuda and Oikawa suggest several support mechanisms. There are two levels of schedulers: a user-level scheduler (ULS) and kernel-level scheduler (MLS). To avoid costly context switches, it must be possible for user threads to sleep and wake-up. To make a user-level thread sleep, the ULS maintains a list of 'sleep descriptors'. Each sleep descriptor is a data structure with a pointer to the thread descriptor and the time when the thread must be woken. When a thread wants to sleep, an available sleep descriptor is found and initialized. The sleep descriptor is added to the end of the linked list of sleeping user threads' sleep descriptors. A pointer called ‘uls_sleep’ points to the head of the linked list. Another pointer 'uls_sleep_hint' points to the sleep descriptor with the soonest wake-up time. A variable 'uls_sstart' contains the wake-up time of the next thread to be woken. The next thread to be woken is the thread with the soonest deadline.

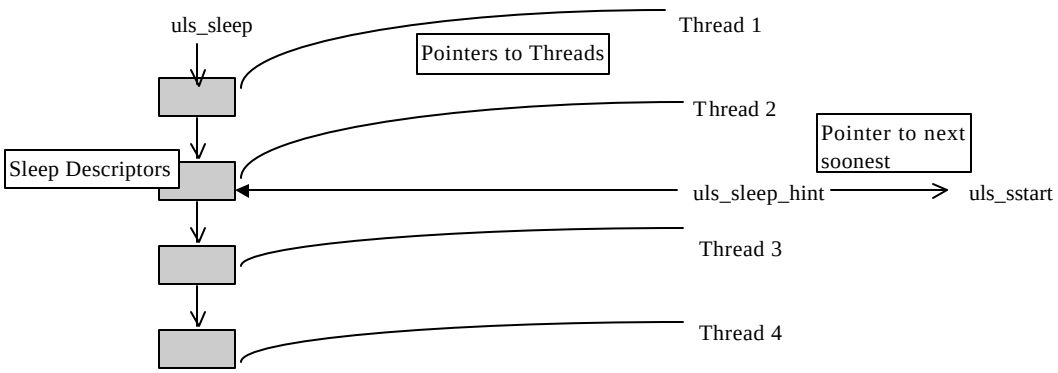


Figure 4.

A thread needs to be woken when the time in 'uls_sstart' has passed. The kernel checks the sleep descriptor pointed to by 'uls_sleep_hint' to obtain the thread id with the next deadline. The thread's priority is examined and the variable 'uls_rpriority' (which contains the thread id with the highest priority among runnable threads) is updated. The 'uls_sleep_hint' pointer is updated to point to the next soonest sleep descriptor. This description is a high-level overview with some details omitted.

The sleeping and waking up mechanisms described above are all implemented at user-level. This implies fast, efficient scheduling of user-level threads by the ULS (user-level-scheduler). This scheme relates to CPU scheduling by implementing a user-level scheduling scheme for real-time processing instead of kernel based process scheduling. The architecture

described above is an implementation for periodic real-time threads that is applicable to multimedia. Periodic deadlines are characteristic of most multimedia processes, so its design is of interest. The threads are "woken" by notifying the ULS in the 'uls_kev' event buffer when the thread's periodic time deadline is reached. The data structures are updated, and when the thread’s processing is complete for that period, it is put back to sleep. This periodic scheduling of user threads is entirely implemented by the ULS, so the overhead of kernel-level context switching is avoided.[6]

Quality of Service

Relation to CPU Scheduling: The CPU is a resource that services threads' demands for CPU cycles. Multimedia is sensitive to variations in the quality of service provided. The CPU scheduler must provide more reliable service to multimedia threads than other processes. This section explores ways to help the CPU scheduler provide this service. It also discusses solutions when the CPU scheduler is unable to provide adequate service.
References: [5], [6], [7], [8], [10]

4.1 Buffer Management

For multimedia applications to function correctly, there must be a steady stream of data for the output devices to process. For the viewer to perceive continuous media such as movies or music, the output devices have to output new media within strict time constraints (e.g. 30 frames per second for video applications). If there is no data for the output devices to process, there is buffer underflow. The media application will stall and wait for new data to be provided. Naturally, buffer underflow should be avoided whenever possible.

Buffer underflow occurs because the application was not able to compute new data for processing. This may be due to lack of performance of I/O devices or processors. Poor reservation of resources may be at fault, such as an overloading of CPU cycles and memory. Poor scheduling of resources may also be at fault. If the CPU is not scheduled appropriately, deadlines may be missed which will lead to buffer underflow. Application deadlines are established to ensure a certain quality of service. Essentially, if the deadline of an application is respected there will never be buffer underflow (i.e. the application is progressing at the appropriate rate).

Poor scheduling can lead to buffer underflow if process priorities are not established. If the CPU is scheduling a batch program at the expense of a multimedia process, the user may notice degradation in output quality (video or audio stalling). However, if the batch program waits several milliseconds for a real-time process to execute, the user may not notice or even care that his program took a fraction longer to execute. The user certainly will notice, however, if his video player starts stalling. Hence, even if the performance of processors and peripherals is excellent and each process is allocated all the resources it needs prior to execution, multimedia applications may still not work if proper scheduling mechanisms are not implemented.

Finding a working schedule for allocating CPU cycles is accomplished with earliest-deadline-first scheduling. Determining what to do if no schedule exists to meet all deadlines (called *CPU overloading*) is another problem. It can be avoided all together by proper reservation of resources prior to execution. This may lead to lower resource utilization, however. In terms of CPU cycles, a process may negotiate with the operating system the amount of cycles it will receive. If the processes require more than the estimated number of cycles, CPU overload may still occur. This can cause real-time processes from missing their deadlines that may lead to buffer underflow. If the performance of the processor is not great enough to support the load, no scheduling algorithm will be able to adequately service all processes. To alleviate this problem, the operating system can begin aborting processes until the deadlines of all executing processes are brought under control. This solution is not always feasible for obvious reasons.

A buffer of data can be accumulated to gracefully handle occurrences of CPU overload. If the CPU becomes overloaded, a buffer can provide time for the demand on the CPU to drop. If a process misses a deadline due to processor overload, data can be processed from the buffer until it is exhausted. The assumption is that the time-faulting process will resume execution before the buffer is empty. The size of the buffer is an implementation detail dependent on the application and system used. If a deadline is missed and data is removed from the buffer, there must be a way of replenishing the buffer. Scheduling a process more frequently after a time-fault until the buffer is replenished can do this. This assumes the CPU is able to produce data for the output buffer faster than it can consume it. Otherwise, buffer underflow will inevitably occur and the real-time application will stall.

If processor load suddenly drops dramatically, real-time processes may find they are being scheduled more frequently than necessary. Data may be produced several periods before its deadline. This can lead to buffer overflow. Buffer overflow occurs when the amount of data sent to the buffer exceeds its capacity. The operating system must have mechanisms to deal with this. The operating system may choose to not schedule a process until its deadline is approaching. This will ensure the buffer will have time to consume the data before more is produced. The process will be scheduled at sufficient intervals to prevent buffer overflow and underflow. The operating system scheduler can also be notified when the buffer is becoming too full. The process will not be scheduled until the buffer diminishes to an appropriate level. The advantage of implementing a buffer scheme to deal with CPU overload is simplicity. The attributes of the processes and the scheduler remain the same while CPU demand fluctuates. The disadvantage of buffers is the possibility of underflow and overflow, and the additional memory requirements.

4.2 Dynamic QoS

A more sophisticated solution involves adjusting the quality of service (QoS) dynamically as CPU workload increases or decreases. The adjustment of quality of service can be implemented as a user thread ("user thread" refers to an application thread, and not necessarily a user-level thread as described earlier) or kernel thread. Tokuda and Kitayama proposed a model for both schemes. If a process is time faulting, this is because the process is not being scheduled on time (assuming adequate CPU performance). One can suggest reducing the workload by aborting processes or upgrading system performance, but this is not always feasible. Tokuda and Kitayama's thread model gives facilities for implementing a dynamic quality of service scheme. In our example of a video player, the number of frames per second may be reduced. Essentially, the deadline D and period T are increased to accommodate the lagging CPU. Using their real-time thread architecture, a solution for a self-stabilization scheme and quality of service manager scheme is introduced.

Self-stabilization adjusts QoS at the application level. If a process begins time faulting too frequently, it will decrease its quality of service demands on its own. When the process has met a fixed number of deadlines in a row, it will demand better quality of service. The self-stabilization scheme responds to its own time-fault frequency.

Tokuda and Kitiyama's manager scheme monitors the CPU capacity at the kernel level. When the CPU becomes overloaded, the manager will ask some or all runnable processes to lower their QoS demands. Likewise, when availability of resources increases the processes will be

permitted to increase their service demands. The kernel-level thread responsible for monitoring system resources will notify the user threads when either situation arises.

In context of multimedia applications, the CPU scheduler determines quality of service rendered. The more CPU cycles scheduled to a process, the more data can be produced faster, which results in a better quality, more reliable output. In our example of a video player application, more CPU cycles would produce more frames, thus allowing more frames per second to be displayed on the workstation monitor. However, if several videos are executing at the same time there may not be enough CPU cycles available to produce all the video frames requested. If certain video frames are large and complex, there still may not be enough CPU cycles to meet the deadline, even if there is only one video player executing. In this case, the quality of service would be reduced to a level where all the frames can be computed within specified deadlines. This would result in, for example, displaying an image at 15 frames per second instead of 30 frames per second. The resulting video will appear less smooth and jitterier than before. To reduce the requested quality of service, the number of deadlines within a certain time frame must be reduced. This will reduce the amount of computation required by the CPU. The CPU will then be able to respect all deadlines, and the applications will execute without time faulting.

There may be some confusion why dynamically reducing quality of service is better than simply letting some applications time fault and catch up when the workload shrinks. If a video application time faults a lot, eventually the output buffer (if any) will be depleted and there will be no new video frames to display on the monitor. This will make the movie stop and start randomly. The video will appear to freeze for a moment while waiting for the next frame to be computed. The application may choose to discard all frames that missed its deadline. In this case the video will resume at the same frame if the stall had never occurred. This will result in "popping" or jumping ahead in the video without displaying any of the intermediary frames. The application may choose to catch up to the correct frame in time by speeding up the display of all frames that caused a time-fault. This will resemble watching the video in fast-forward for several moments after the stall occurred. The application may choose to display at regular speed all the frames which time faulted. In other words, the video application would stall, and resume at regular speed when frames become available for output. This will require extending the playtime of the whole video. This may not be a good solution if there is another process executing independently in parallel to the video. For example, if the video is a documentary and there is an audio process executing with narration, the discussion of the narrator should presumably correspond to the images displayed by the video application. If the video process lags behind the audio, eventually the discussion of the narrator will be unacceptably out of sync with the video. To solve this problem, we may make the audio process a slave and the video process a master. The audio output to the speaker will directly correlate the frames output by the video. However, this may also be unacceptable. Minor synchronization problems with the audio and video may execute unnoticed by the user. However, if the audio turns on and off every time the video process time faults, the resulting display may be worse than being out of sync.

As one may notice, there is no graceful solution to handling excessive time faulting. Reducing the quality of service provides a much more suitable solution. The QoS is diminished to a level where applications cease to time fault and execution is predictable. Assuming this dynamic QoS scheme, if our video player application begins to time fault frequently it will lower the frames per second displayed. There is a question of which frames to eliminate per time unit. If the video was originally displayed at 30 frames per second, and is subsequently reduced to 15 per second,

the application may choose to simply play all thirty frames over two time units. The video will be displayed as smooth as the original, although twice as slow. The application may choose to eliminate every second frame. The display will be the same speed as the original, although appear less fluid.[5] [6]

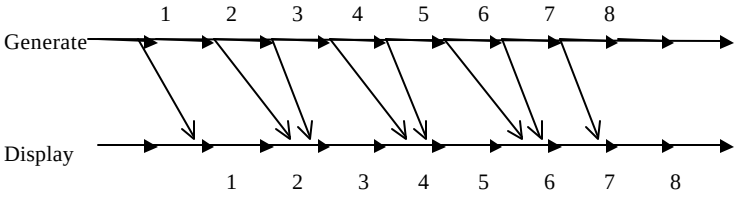
4.3 Adjusting Display Latency

Stone and Jeffay of University of North Carolina at Chapel Hill studied methods of dealing with delay jitter. *Delay jitter* occurs when the time to process data for display in a multimedia application varies. This is a problem because the probability of time faults increase. Stone defines a *gap* as the time a multimedia application spends stalling because it is waiting for data to display. If a video player executes with no gaps, then the output will be a steady, continuous stream of images. The *display latency* is defined by Stone as the total time from acquisition to display. The *end-to-end display time* is the time from acquisition to decompression. The display latency is the end-to-end display plus any time spent in output buffers. In order to observe a perfect execution with no gaps, the display latency must be greater than or equal to the worst-case end-to-end delay of each frame. If the output buffer is empty and the end-to-end display time of a frame is larger than the display latency, a gap will occur. We are dealing with a question of definitions rather than concepts. The terms used in Stone's paper are analogous to the discussion of time faulting described earlier. A time fault will result in a gap. The display latency and end-to-end display time refer to the period 'T' and frame processing time, respectively.

Stone discusses the trade off between display latency and gaps. His discussion is only relevant to "conference" multimedia or live communication applications, although this is not explicitly mentioned in his paper. If a live conference is being conducted between two people, it is best if images are displayed on one end close to the time they were captured. In other words, one would like to see a person talking as it occurs. Otherwise, there will be an effect similar to long distance telephone calls where one asks a question and receives an answer after an unusually long pause. If one party in the conference wishes to interrupt or perhaps nod in approval to a comment that is said, the time to display the reaction should be very close to the time that the comment was said. Essentially, gaps may be tolerated for the sake of shorter display latencies. Images of lower quality and mild display jitter may be a lesser evil than long display latencies.

Stone and Jeffay discuss two algorithms for dealing with a frame that arrives late and causes a gap. The I-Policy discards the frame. The E-Policy raises the display latency to the highest yet observed. If a frame arrives late in the E-Policy, all subsequent frames will be displayed regularly at the time it took the late frame to arrive. It is a worst-case scheme to ensure no more gaps occur. The I-Policy specifies a display latency at the beginning of execution and maintains it until the end. The trade-off of display latency and gaps is analogous to the trade-off between allocating more or less frames to a process in a virtual memory system. If allocated more frames, the process will page fault less frequently but will consume more memory. With a higher display latency, the process will time fault less but will take more time to execute.

The E-Policy out performs I-Policy when the end-to-end time of frames varies frequently. If the end-to-end display time varies from 1 to 10 time units, it would execute most consistently if the display latency were set to 10 at the beginning. The I-Policy out performs E-Policy when the end-to-end time of frames "spikes" infrequently. With the I-Policy, occasional time faults are sacrificed for a lower average display latency.

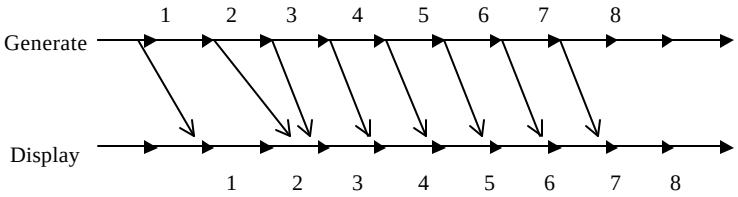


	E-Policy								**I-Policy**							
	Frames								*Frames*							
	1	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*
Display Latency	1	2	2	2	2	2	2	2	1	2	1	2	1	2	1	1
Queue Length	0	1	0	1	0	1	0	0	0	0	0	0	0	0	0	0

Figure 5.

In the above diagram, the E-Policy outperforms the I-Policy. Frames 2, 4, and 6 arrive late. The display latency is adjusted in the E-Policy after the first time-fault (or "gap"), so subsequent frames are displayed at larger intervals.

The queue size is an E-Policy disadvantage not mentioned by Stone. If the display latency in the E-Policy is set large due to big variations in end-to-end display times, the queue of frames ready for display must also be large. When a burst of frames arrives quickly (the end-to-end display times are low), the queue fills up. However, when the end-to-end display time is close to the display latency time (which is conservatively large due to the E-Policy), the queue remains empty. Depending on the system this can be a serious waste of resources.



	E-Policy								**I-Policy**							
	Frames								*Frames*							
	1	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*
Display Latency	1	2	2	2	2	2	2	2	1	2	1	1	1	1	1	1
Queue Length	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Figure 6.

In this example the I-Policy outperforms the E-Policy. Only one frame at the beginning arrives late. However, subsequent frames arrive on time. The display latency remains low for the remainder of execution. The E-Policy raised its display latency for the remaining frames in anticipation of more late frames. This did not occur however, and slowed down the execution of the program unnecessarily. A single burst of activity caused a frame to arrive late resulted in a permanent increase in display latency.

Stone proposes a new policy that includes advantages from both the I-Policy and the E-Policy. Stone refers to it as the queue monitoring policy. The policy is analogous to the working-set model for allocating frames in a virtual memory system. In queue monitoring we associate an array of counters and threshold values with the display queue. Each time a frame is to be displayed, the counter for each frame in the queue is incremented. Any counter that exceeds the threshold is discarded. The oldest frame is then displayed. Discarding the frames will result in lower display latency since the rest of the frames in the queue will be displayed earlier.

Stone described this policy very briefly and without specific details. As Stone described the policy, it is not evident to me how the queue-monitoring algorithm capitalizes on the advantages of the I-Policy and the E-Policy. Stone shows the policy results on a series of experimental runs. In most cases the queue-monitoring algorithm had lower display latency and less gaps than the I-Policy and the E-Policy.[7]

4.4 Time-Fault Avoidance

Andreas Mauthe and Geoff Coulson from Lancaster University wrote a paper dealing with display jitter on constrained periodic threads. Mauthe deals with display jitter (i.e. time faults) by avoiding CPU overload. If the CPU is not overloaded, it can meet all real-time thread deadlines. Hence, if all deadlines are met there will be no time-faults, and no gaps will occur in the display image. The paper proposes an admission test to ensure the CPU scheduler will be able to respect all deadlines.

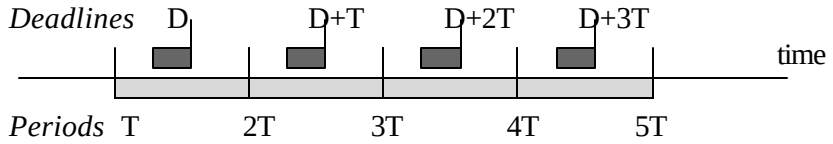
Constrained periodic threads are standard real-time threads where the specified deadlines may be handled earlier than the end of the current period. In other words, if a frame has a deadline at the end of period 't' to produce a frame, it may produce the frame anytime between the end of period 't-1' and the end of period 't'. This can be accomplished by buffering at output devices. The admission test determines if the thread can be added to the set of runnable threads so all deadlines can still be respected. If the admission test fails, the user may get an error message stating a lack of resources.

Mauthe describes a whole scheduling system rather than just a CPU scheduling algorithm. The algorithm of choice for the CPU scheduling algorithm is the earliest deadline first. This algorithm selects the thread with the earliest next deadline for execution. This algorithm is optimal, since it is guaranteed to find a schedule of threads that will respect all deadlines if such a schedule exists. It also fully utilizes the CPU resource.

Admission tests are applied to ensure a valid schedule exists for all threads in the runnable set. The first test applies the above fact that the earliest-deadline-first algorithm is optimal. Therefore, we must just ensure that CPU utilization does not pass 100%. The fraction: E(i) / P(i) represents the execution time of thread 'i' divided by the period (i.e. the time interval between

periodic deadlines) of thread 'i'. If we add E(n) / P(n) for the new thread ‘n’ to the existing CPU utilization, we will get the CPU utilization when the thread is added to the runnable set. If the new utilization is less than 100%, then it is safe to add it to the runnable set. The term "safe" implies that all deadlines can be met by the CPU (the CPU is not overloaded). This computation requires estimating the computation time of each thread, which is difficult and can be inaccurate. If several estimations of computation time are too conservative, CPU overload may still occur.

An alternative admission test considers explicit deadlines instead of period duration. In other words, replace the term P(i) with the deadline for thread 'i' (denoted 'D(i)'). The summation to compute CPU utilization becomes E(i)/D(i) for all 'i'. A definition of period and deadline is necessary to understand the concept. A period is the time from the start of execution for deadline 'i' to the start of execution for deadline 'i+1'. A deadline is the absolute time within a period where a certain amount of execution must be
complete. A period refers to a magnitude of time, whereas a deadline specifies a point in time. This implies that the execution to meet deadline D(i) does not have to start at the beginning of the period.



Period Time

Execution Time to meet Deadline (D, D+T, D+2T, D+3T)

Figure 7.

The problem with considering deadlines instead of periods is that some schedulable set of threads will be rejected by using the summation of E(i)/D(i). For example, consider the following two threads:

t1: p=3, d=1, e=1
t2: p=5, d=4, e=2

If only t1 is executing, we have E(1)/D(1) = 1.0 = 100%. However, t2 can be successfully scheduled, since the actual CPU utilization is only 33%. The new process ‘t2’ requires 50% of the CPU cycles (E(2)/D(2) = 50%), so the new CPU utilization will be 88%.

Mauthe gives the following insight into the scheduling problem: “A candidate thread can be accepted if, in the period length of each of its invocations, there is sufficient spare resource to schedule the required number of invocations of all other threads with an earlier or equal deadline, even when the scheduling times of all threads concerned coincide.”

It is unclear to me why Mauthe proposed replacing P(i) with D(i) in the original summation E(i)/P(i) for all 'i'. P(i) is a magnitude (an interval in time) while D(i) refers to a point in time. The two terms have little relationship to one another. Mauthe did propose a correct summation formula using the value of D(i) in the denominator. This admissions test formula is omitted due to its length and complexity.

Mauthe mentions a flaw with the above admissions tests; it assumes all threads will be ready for execution when the scheduler allocates the CPU. This is not always the case, and adds a further constraint on the admission test. The new constraint to deal with this circumstance is omitted.[10]

4.5 Application-level Solutions

Kevin Fall, Joseph Pasquale, and Steven McCanne of University of California at San Diego propose a radically different solution to dealing with time faults. We have seen mechanisms for avoiding time faults such as an admission test for new threads, dynamic quality of service imposed by the operating system, and buffer management by output peripherals.

Fall proposes a solution at the application layer. This solution requires no operating system support to manage real-time threads. The application itself is aware of its temporal constraints, and adjusts its demands on the CPU according to its capacity. The intention is to provide a mechanism for real-time computation without having to implement real-time scheduling by the operating system. The goal of the application is to minimize display jitter by gracefully adapting the application's requirements to available CPU resources.

Fall implements a load monitoring agent at the application level that determines when CPU resources become scarce. The load-monitoring agent accepts output frames from the application that must be dithered and displayed. This consumes most of the computation in video playback. If CPU workload increases above a certain level, the load-monitoring agent can discard frames to reduce load on the CPU. Since the agent discards before the majority of processing on the frame is done, significant load shedding can be achieved. The agent acts as an interface to the CPU to control the amount of processing requested. The agent monitors CPU workload by computing inter-frame display times (IDTs) and records the variance of recent samples. Fall concludes there is a direct correlation between IDT variance and the number of CPU-bound processes. Hence, the variance of the display times between frames increases as CPU workload increases.

Fall claims this agent will effectively allow real-time processing without system intervention. I do not believe this is possible, however. First, there must be some mechanism in the operating system to communicate with the load-monitoring agent to get display-time data. Second, the CPU scheduler cannot treat real-time applications the same way it does batch or interactive processes. If the scheduler uses a "shortest-job-first" type of algorithm, real-time applications with long execution times can starve. Load shedding will not be able to gracefully accommodate a starving process, no matter how clever it may be designed. The CPU scheduler must treat real-time processes with a higher priority than other processes to effective execute the job. The load-monitoring agent does, however, migrate the complex problem of load shedding to the application layer. This allows simpler operating system design, and provides an easily upgradeable solution.[8]

CPU Scheduling

Relation to CPU Scheduling: This section deals with algorithms implemented by the CPU scheduler. The traditional CPU scheduling algorithm for real-time processing is 'earliest-deadline-first'. However, researchers have implemented interesting variations to the algorithm, some of which are explored in this section.
References: [9], [11]

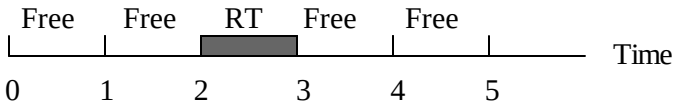
5.1 Fixed-Time Allocation

Jason Nieh and Monica Lam of Stanford University wrote a paper on a new CPU scheduling algorithm they developed for continuous multimedia applications. They correctly mention the problems with existing CPU scheduling algorithms. By giving real-time computation a higher priority than system and other user processes, it limits the utilization of the system and artificially constrains the range of behavior the system can provide. Priority real-time execution can cause system services to lock up, and the user can lose control over the machine. Nieh claims to have found the solution to all these problems. However, the paper only describes what it does and leaves out key implementation details. The scheduler supposedly handles real-time, batch, and interactive computation "seamlessly" without requiring any user parameters.

Real-time processes are allocated the CPU first. They are allowed to execute for a fixed amount of time. If some processes are not able to meet their deadline within the fixed amount of time allocated, the process is notified it will miss its deadline. The process may abort or continue executing, depending on the application. Conventional processes are then allocated a fixed amount of CPU time. The cycle continues, alternating between conventional and real-time execution. When the scheduler is in real-time mode, the processes are scheduled in an earliest deadline first scheme. Conventional processes are allocated in a round-robin discipline.

I believe the amount of time allocated for real-time and conventional computation depends on the workload ratio, although Nieh does not specify this. I have two serious problems with his solution. First, he claims his solution is based on fairness. Real-time processes are given their "fair-share" of CPU time, and conventional processes are given their "fair-share". He never specifies what "fair" means. Fair is a very subjective term, and depends on how critical the real-time processes' deadlines are. Second, he ignores the potential disastrous effects of a real-time process time faulting. Some real-time processes can deal with a certain level of service degradation. However, some applications are pointless unless all deadlines are met. If these processes cannot meet their deadlines within the fixed amount of CPU time given, the process is removed from the CPU and it is given to a temporally insensitive process anyway (like a text-editor or compiler). This is not always the best solution.

Nieh describes real-time computation as seizing the processor and not relinquishing it until it has finished. This is not always the case. If a process has a period of time 5, deadline 3, and execution time 1, then the CPU is free for conventional processing 80% of the time, even when the real-time process has a higher priority as it approaches its deadline.



'Free' = no real-time process scheduled
'RT' = real-time process scheduled to meet deadline

Figure 8.

As well, consider a real-time process that has a period of 1000, deadline 100 and execution time 100. The CPU is free for conventional processing 90% of the time. If the scheduler only allocates a fixed time of 99 time units, the process will never be able to execute properly. The application will time fault relentlessly. The solution of allocating a fixed amount of time to a processor can also be unfair to real-time processes that have almost finished the computation required to meet its deadline. Preempting the process to allocate time for a batch program is not a good scheduling technique.

Nieh's scheduler is a good solution when the real-time deadlines can be met within the fixed time allocated. In this case the scheduler provides adequate service for all classes of computation. It is not, however, a universal solution as Nieh seems to claim.[9]

5.2 Rate-Based Priority Scheduling

Yavatkar and Lakshman of the University of Kentucky developed a dynamic CPU scheduling algorithm to support multimedia processing. It is called rate-based adjustable priority scheduling (RAP). The algorithm makes three assumptions about the real-time processes it schedules. First, RAP does not assume a priori knowledge of resource requirements by MM applications. Second, RAP assumes multimedia applications can tolerate occasional delays in execution. Third, RAP assumes MM applications are adaptive in nature and can gracefully adapt to resource overloads by modifying their behavior to reduce their resource requirements.

At the beginning of execution, an application specifies a desired average rate of execution and a time interval over which the average rate of execution will be measured. RAP implements an admission control scheme that calculates the available CPU capacity and compares it to the requested execution rate. If an acceptable execution rate can be allocated, then the process is placed in the set of runnable processes. The queue of real-time processes is organized on a priority basis. Each process priority is based on the requested rate of execution. It is not clear how the priority relates to the rate of execution.

Once a process is admitted to the set of runnable processes, the scheduler allocates the CPU using a priority-based scheduler and a rate regulator. The rate regulator ensures a process which was promised an average execution rate R does not execute more than R times a second and executes roughly once every T=1/R time interval. After a process executes for the duration of one averaging interval, feedback is provided back to the application about the observed rate of progress. The quality-of-service manager assumed to be implemented in the application reacts accordingly. It may increase or decreased its desired rate of execution. RAP also has a mechanism that monitors CPU capacity. If the CPU is over or under-utilized, it can communicate

with application level processes to decrease or increase its resource demands by a fraction of its current demand, respectively.

This algorithm provides a good basis for future work in system and application layer cooperation. As opposed to some of the other scheduling techniques described which were entirely system or application based, this scheduler is effectively implemented at both the application and operating system level. Communication and cooperation of the two levels help establish a fair and adaptable scheduling discipline.[11]

OS Upgrades

Relation to CPU Scheduling: Developing a comprehensive new operating system for multimedia is a difficult and expensive task. Companies have sunk much capital into developing and testing existing conventional operating systems, so significant research is focused on upgrading instead of building from scratch. This section discusses research attempts at CPU scheduling upgrades to handle multimedia applications.
References: [1], [12], [13]

6.1 Ultrix-4.2

Tom Fisher from the University of California at Berkeley developed real-time scheduling support for the Ultrix-4.2 operating system. He describes mechanisms added to the operating system to support multimedia applications. There are five modifications to the kernel: real-time process establishment, preemption points, data structure locks, priority inheritance, and internal event processing. I will only summarize those related to CPU scheduling.

Real-time process establishment allows processes to define themselves as jobs with critical temporal constraints. A process will request real-time status from the kernel. The kernel will perform analysis on the process (the details of this analysis is not discussed by Fisher) and decide whether or not to grant real-time status. The kernel will allow only four real-time processes to exist at a time; the rest of the threads will be conventional user threads. Real-time threads have higher priority than conventional user threads. There are three types of real-time priorities for various levels of urgency.

Preemption points are placed in the kernel process. These points identify points in execution where the kernel process can be preempted in favor of a real-time process of suitable priority. There is dilemma in research concerning the priority of real-time processes in relation to kernel processes. If real-time processes are of higher priority, the system will lock-up until the multimedia application has finished execution. However, the scheduler should attempt to meet all real-time deadlines. The solution to both problems conflict, so there is no easy solution. Fisher's solution is to allow kernel preemption, but only at specific locations. Preemption at some points requires locking data structures since the kernel has saved data. The goal is to maximize the number of preemption points while minimizing any data locks required. Originally, Fisher tried to implement preemption points without having to lock any data structures. Data locks can cause processes to wait unnecessarily, so locks are avoided if possible. However, there were too few points in the kernel code that allowed preemption with no data locks. Urgent real-time processes would have to wait too long for the kernel code to reach a preemption point. Therefore, Fisher added more preemption points with limited locks on data structures.

The third interesting addition to Ultrix-4.2 is special handling of internal event processing. Traditional UNIX handling of internal time-out events takes too long; up to 5ms in some circumstances. Fisher has adapted this operating system feature to make it suitable for real-time processing. This implies making the interrupt faster for real-time processes, and blocking out all other normal callout processing during the execution of real-time processes.

These three modifications demonstrate how a conventional operating system can be modified to support multimedia applications. This paper is particularly interesting because of Fisher's preemption points in the kernel code. Although the solution is ad-hoc, his experimental results show a reasonable performance.[12]

6.2 Chorus

Coulson, Blair, Robin, and Shepherd of the University of Lancaster have extended the Chorus operating system to support continuous multimedia applications. Chorus is a micro-kernel based operating system developed in France. It supports the implementation of conventional operating system environments through the provision of sub-systems. For example, UNIX or Windows 95 may be a sub-system. The details of Chorus are not important to the discussion of multimedia scheduling, so it will be omitted.

Real-time scheduling exploits user-level threads whenever possible to minimize context switch overhead. Three types of threads are implemented: system threads which are kernel supported threads in supervisor mode, kernel threads which are kernel supported threads in user mode, and user threads which are multiplexed on top of kernel threads. All three types of threads are non time-sliced but preemptive.

The real-time scheduler uses the earliest-deadline-first policy. However, quality of service is not treated as an absolute requirement. There are no guarantees all deadlines will be met. Processes may begin time faulting if the processors become overloaded. Overloading can be avoided with an admission test for new threads. Non real-time threads are scheduled using the sub-system's standard policy (e.g. round-robin time sliced). The real-time thread support added by Coulson co-exists with existing Chorus thread facilities.

Real-time thread scheduling is a split-level scheme consisting of a single kernel scheduler (KLS) and multiple co-operating user level thread schedulers (ULS). The user level thread scheduler allocates the kernel thread to one of several user level threads multiplexed on the kernel thread.

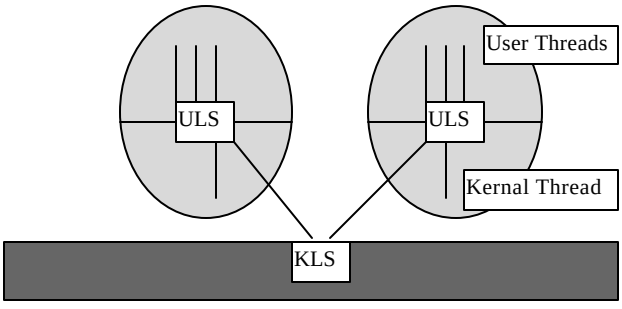


Figure 9.

The user thread scheduler allocates the kernel thread to the user thread with the earliest deadline. The kernel scheduler schedules the kernel thread that is executing a user thread with the globally earliest deadline. Both the ULS and KLS implement the earliest-deadline-first algorithm. In order for the KLS to be aware of user thread deadlines, a mechanism must exist for the ULS and KLS

to communicate. This is accomplished by shared memory and "upcalls" by the kernel. Real-time threads are given first priority over conventional threads. Real-time threads are allocated as much CPU time as needed to complete its execution. Any residual CPU cycles left over are given to conventional processes.

A bug exists in the split-level scheduling design. If a user-thread blocks on a system call, the associated kernel thread will block as well. If there is another user thread sharing the same kernel thread with the globally earliest next deadline, it will not be able to execute. One solution proposed by Coulson is non-blocking system calls. The details of its implementation are not discussed.

Coulson's paper introduces a simple, effective method of upgrading an operating system to make it multimedia capable. The solution gives insight into a new split-level scheduling design. This has the advantage of fast execution at the user-thread level, but suffers with system call inefficiencies.[13]

6.3 UNIX

Nieh from Stanford University writes another paper that contradicts all the research claiming to have successfully implemented a real-time thread management system for UNIX based operating systems. Nieh claims UNIX based operating systems are not only ineffective in handling real-time computation, but that any attempt to supply an add-on solution to the operating system is destined to fail. Nieh claims the UNIX process scheduler is ineffective and even causes system lockup. Even the UNIX mechanism for modifying the CPU scheduler's behavior cannot be used to find an appropriate setting to support real-time processes. Nieh's experimental results show poor application latencies and "pathological" system behavior during real-time execution.

Nieh chooses a flavor of UNIX claiming multimedia support called SVR4. SVR4 supports multiple scheduling policies. A real-time process scheduler arranges the processes in a particular priority. The timesharing scheduler (for batch and interactive processes) arranges its set of processes in a particular priority. A single scheduler finally allocates the CPU from both process queues. All real-time processes are considered to be of higher priority than timesharing processes.

Nieh identifies three types of computation classes: batch (e.g. compilers), interactive (e.g. text editors), and continuous media (e.g. continuous media). Each type of computation has a different definition of performance. Batch processes wants to minimize throughput. Interactive processes wants to minimize the average and variance of response time. Continuous media wants to minimize the difference between the average display rate and the desired display rate. During data collection, one process from each class was executing (i.e. three processes in total in the runnable queue). Nieh measured the most important performance factor for each type of process during the experiment. Each computation class is given a "baseline value". This value is the measurement of the most important metric to the process (e.g. response time in interactive processes) when executed in isolation. The data displayed by Nieh is expressed as a percentage of the baseline metric for each computation class. For each metric measured, both the mean and standard deviation is considered.

For example, consider a batch process executing in isolation. The mean and standard deviation execution time is recorded. If a continuous media application were run in isolation , the

mean and standard deviation of time between successive video frames would be considered over several executions. Given the computed baseline for all three computation classes, Nieh executes all three concurrently competing for the CPU and other resources. For each computation class the principal metric is measured again. The performance of the system is displayed as a percentage of the baseline value. For example, if a batch program took 10ms to execute in isolation but 20ms with other processes executing concurrently, then the system performance for the mean computation time in the batch class is 50% (10ms/20ms). If the metric improves by maximizing the value, then the baseline is placed in the denominator. If the metric improves by minimization, then the baseline is placed in the numerator. In the previous example, execution time improves by minimizing the value; hence, the baseline measurement of 10ms is placed in the numerator. Note that the final system performance cannot be greater than 100% (theoretically) since the baseline is measured in the optimal case of isolated execution.

An important scheduling problem with continuous multimedia is setting a priority for the X-window server. If the video application has a higher priority than the X-window server, then frames will be continuously processed but never displayed. They will remain in the X-window input buffer until the video process dies. If the X-window server has a higher priority, then the input buffer will not be filled with new frames at regular intervals. The X-window server will display all frames in the buffer until it is exhausted. Then the server will have to wait for the video process to produce more frames, which will appear as a stall or jitter to the user. Nieh added an X-window process to the experiment's runnable queue so the experiments were run with four processes: the X-window server, and one process from each computation class.

Nieh attempts to adjust the priorities of all four processes to try and yield reasonable performance. Again, performance is measured by its percentage of the baseline value. An attempt is made to classify some of the processes as "real-time" and others "time-sharing". No feasible combination of priority classification was found. Nieh showed many charts displaying every combination of priority adjustments and real-time/timesharing classification. Three of the more interesting charts are shown below:

	Interactive Mean	*Interactive Standard Dev*	*Batch Mean*	*Batch Standard Dev*	*Video Mean*	*Video Standard Dev*
All threads in Time Sharing class (TS)	0%	0%	99%	40%	5%	0%
X-Window and Video Threads RT. P(X-Win) > P(Video) Others TS	0%	0%	15%	0%	99%	> 100%
All threads in New Time Sharing class	90%	85%	40%	5%	65%	15%

Table 1.

Note that the percentages are the measured metrics compared to the baseline measurement. 0% indicates performance was virtually non-existent compared to the baseline measurement. >100% is an interesting anomaly where there was less variance in measurements than the baseline. This

is due to more rigid priority scheduling. 99% indicates the performance is practically identical to the baseline measurement. The 'New Time Sharing' class is described later.

Nieh explains the reasoning behind the poor results in some detail. I will merely summarize the two key arguments he claims are responsible for making UNIX unable to support multimedia applications. First, if no special status is given to real-time processes (i.e. all processes are considered timesharing), this severely degrades the performance of multimedia processes. Second, if some processes are considered to be of exclusive higher priority, they monopolize the CPU at the expense of the other processes' performance. Nieh argues that if some mix of priority assignment and real-time/timesharing classification can be found that works well, this is probably only for the particular applications executing, and would not perform as well given another mix of processes.

Nieh describes a new scheduling class for all processes instead of the two tier real-time and timesharing classes. The new class retains a bias toward interactive and real-time processes while ensuring steady progress of all processes. The details of this new class implementation are omitted. However, the results are shown in the above chart. The X-window server, batch, interactive, and continuous media processes were all classified under the new scheduling class.

The significance of this paper is that it contradicts the claims of many research papers. Nieh claims to have experimentally proven that any operating system using a two-tier approach to scheduling does not perform well in a mixed job environment. If some processes are classified as real-time with a higher priority associated with it, then other types of processes will not progress well. Although this may be true, some may argue that the system may not be designed to run three types of computation at the same time. The system may be such that a diverse mix of computation with associated performance metrics does not occur frequently. Hence, in practice the system may perform well despite Nieh's findings.[1]

References

[1] Jason Nieh, James G. Hanko, J. Duane Northcutt, and Gerard A. Wall. SVR4 UNIX Scheduler Unacceptable for Multimedia Applications. 4th International Workshop, NODSSDAV '93, November 1993.

[2] T.D.C. Little, and A. Ghafoor. Scheduling of Bandwidth-Constrained Multimedia Traffic. Second International Workshop, November 1991.

[3] Ralf Guido Herrtwich. The Role of Performance, Scheduling, and Resource Reservation in Multimedia Systems. Operating Systems of the 90s and Beyond, International Workshop, July 1991.

[4] Dick C.A. Bulterman, and Robert van Liere. Multimedia Synchronization and UNIX. Second International Workshop, November 1991.

[5] Hideyuki Tokuda, and Takuro Kitayama. Dynamic QOS Control based on Real-Time Threads. 4th International Workshop, NODSSDAV '93, November 1993.

[6] Shuichi Oikawa, and Hideyuki Tokuda. User-Level Real-Time Threads: An Approach Towards High Performance Multimedia Threads. 4th International Workshop, NODSSDAV '93, November 1993.

[7] Donald L. Stone, and Kevin Jeffay. Queue Monitoring: A Delay Jitter Management Policy. 4th International Workshop, NODSSDAV '93, November 1993.

[8] Kevin Fall, Joseph Pasquale and Steven McCanne. Workstation Video Playback Performance with Competitive Process Load. Network and Operating Systems Support for Digital Audio and Video, 5th International Workshop, NOSSDAV '95, April 1995.

[9] Jason Nieh, and Monica S. Lam. Integrated Processor Scheduling for Multimedia. Network and Operating Systems Support for Digital Audio and Video, 5th International Workshop, NOSSDAV '95, April 1995.

[10] Andreas Mauthe, and Geoff Coulson. Scheduling and Admission Testing for Jitter Constrained Periodic Threads. Network and Operating Systems Support for Digital Audio and Video, 5th International Workshop, NOSSDAV '95, April 1995.

[11] Raj Yavatkar, and K.Lakshman. A CPU Scheduling Algorithm for Continuous Media Applications. Network and Operating Systems Support for Digital Audio and Video, 5th International Workshop, NOSSDAV '95, April 1995.

[12] Tom Fisher. Real-Time Scheduling Support in Ultrix-4.2 for Multimedia Communication. Third International Workshop, November 1992.

[13] Geoff Coulson, Gordon S. Blair, Philippe Robin, and Doug Shepherd. Extending the Chorus Micro-Kernel to Support Continuous Media Applications. 4th International Workshop, NODSSDAV '93, November 1993.