Survey on Multimedia Operating Systems

P. DharanyaDevi, S. Poonguzhali, T. Sathiya, G.Yamini, P. Sujatha and V. Narasimhulu

Department of Computer Science, Pondicherry Central University,
Pondicherry - 605014, India.
{spothula, narasimhavasi}@gmail.com

***Abstract*:** Real-time applications such as multimedia audio and video are increasingly populating the workstation desktop. A growing number of multimedia applications are available, ranging from video games and movie players, to sophisticated distributed simulation and virtual reality environment. Multimedia is an increasingly important part of the mix of applications that users run on personal computers and workstations. Research in operating system support for multimedia has traditionally been evaluated using metrics such as fairness, the ability to permit applications to meet real-time deadlines, and run-time efficiency. In addition, the support for real-time applications is integrated with the support for conventional computations. This poster deals with the survey on multimedia operating systems, its process scheduling, disk management, file management and device management techniques.

***Keywords*:** Multimedia Operating Systems, CPU Scheduling, Memory Management, Device Management, File Management.

1. Introduction

Multimedia data demands strict time constraints for processing. In any multimedia application, we may have several processes running dependently on one another [12]. 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.

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 [27].

Personal computers running Windows XP, MacOS X, and Linux are capable of performing a variety of multimedia tasks accurately recognizing continuous speech, encoding captured television signals and storing them on disk, acting as professional-quality electronic musical instruments, and rendering convincing virtual worlds all in real time[5]. Furthermore, personal computers costing less than $1000 are capable of performing several of these tasks at once if the operating system manages resources well [21] [22]. The increasing pervasiveness of multimedia applications, and problems supporting them on traditional systems, has motivated many research papers over the past decade.

In this paper, in section 2 we illustrate some features of multimedia OS, section 3 describes CPU scheduling techniques, section 4 describes memory management, section 5 describes device management, section 6 describes file management, section 7 describes disk scheduling algorithms and finally section 8 describes the conclusion of the paper.

2. Multimedia Requirements

A general-purpose operating system (GPOS) for a personal computer or workstation must provide fast response time for interactive applications, high throughput for batch applications, and some amount of fairness between applications [10] [13]. Although there is tension between these requirements the lack of meaningful changes to the design of time-sharing schedulers in recent years indicates that they are working well enough. The goal of a hard real-time system is similarly unambiguous: all hard deadlines must be met. The standard engineering practice for building these systems is to statically determine resource requirements and schedulability, as well as over-provisioning resources as a hedge against unforeseen situations.

We have identified four basic requirements that the "ideal" multimedia operating system should meet [23]. Although it is unlikely that any single system or scheduling policy will be able to meet all of these requirements for all types of applications, the requirements are important because they describe the space within which multimedia systems are designed. A particular set of prioritizations among the requirements will result in a specific set of tradeoffs; these tradeoffs will constrain the design of the user interface and the application programming model.

1. *Meet the scheduling requirements of coexisting, independently written, possibly misbehaving soft real time applications:* The CPU requirements of a real-time application are often specified in terms of an *amount* and *period*; here the application must receive the amount of CPU time during each period of time. No matter how scheduling requirements are specified, the scheduler must be able to

meet them without the benefit of global coordination among application developers multimedia operating systems are *open systems* in the sense that applications are written independently [6].

2. *Minimize development effort by providing abstractions and guarantees that are a good match for applications requirements:* In the past, personal computers were dedicated to a single application at a time. Developers did not need to interact much with OS resource allocation policies. This is no longer the case. For example, it is possible to listen to music while playing a game, burn a CD while watching a movie, or encode video from a capture card while using speech recognition software. Therefore, an important role of the designers of soft real-time systems is to make it as easy as possible for developers to create applications that gracefully share machine resources with other applications [6].

3. Provide a consistent, intuitive user interface: Users should be able to easily express their preferences to the system and the system should behave predictably in response to user actions. Also, it should give the user (or software operating on the user's behalf) feedback about the resource usage of existing applications and, when applicable, the likely effects of future actions [6].

4. *Run a mix of applications that maximizes overall value:* Unlike hard real-time systems, PCs and workstations cannot overprovision the CPU resource; demanding multimedia applications tend to use all available cycles. During overload the multimedia OS should run a mix of applications that maximizes overall value. This is the "holy grail" of resource management and is probably impossible in practice since value is a subjective measure of the utility of an application, running at a particular time, to a particular user. Still, this requirement is a useful one since it provides a basis for evaluating different systems [6].

3. CPU Scheduling

3.1 Fixed-Time Allocation

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.

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 [8]. Conventional processes are then allocated a fixed amount of CPU time. The cycle continues, alternating between conventional and real -time execution.

The amount of time allocated for real -time and conventional computation depends on the workload ratio. 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.

3.2 Rate-Based Priority Scheduling

It is also called rate-based adjustable priority scheduling (RAP). The algorithm makes three assumptions about the real -time processes it schedules [8].

1. RAP does not assume a priori knowledge of resource requirements by MM applications.
2. RAP assumes multimedia applications can tolerate occasional delays in execution.
3. 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 [2]. 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.

3.3 Earliest Deadline First

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. The Earliest Deadline first scheduling is theoretically optimal under certain assumptions. Soft real time OS uses EDF as an internal scheduler. Only a few systems such as Rialto and SMART expose deadline-based scheduling abstractions to application programmers. Both systems couple deadline-

based scheduling with an admission test and call the resulting abstraction a time constraint.

Time constraints present a fairly difficult programming model because they require fine-grained effort: the developer must decide which pieces of code to execute within the context of a time constraint in addition to providing the deadline and an estimate of the required processing time. Applications must also be prepared to skip part of their processing if the admission test fails. Once a time constraint is accepted, Rialto guarantees the application that it will receive the required CPU time. SMART, on the other hand, will sometimes deliver an up call to applications informing them that a deadline previously thought to be feasible has become infeasible, forcing the program to take appropriate action [8].

3.4 Feedback-Based Scheduling

Multimedia OS need to work in situations where total load is difficult to predict and execution times of individual applications vary considerably. To address these problems new approaches based on feedback control have been developed. Feedback control concepts can be applied at admission control and/or as the scheduling algorithm itself. In the FC-EDF work [1] a feedback controller is used to dynamically adjust CPU utilization in such a manner as to meet a specific set point stated as a deadline miss percentage. FC-EDF is not designed to prevent individual applications from missing their deadlines; rather, it aims for high utilization and low overall deadline miss ratio.

SWiFT uses a feedback mechanism to estimate the amount of CPU time to reserve for applications that are structured as pipelines. The scheduler monitors the status of buffer queues between stages of the pipeline; it attempts to keep queues half full by adjusting the amount of processor time that each stage receives. Both SWiFT and FC-EDF have the advantage of not requiring estimates of the amount of processing time that applications will need. Both systems require periodic monitoring of the metric that the feedback controller acts on.

3.5 Hierarchical Scheduling

Hierarchical schedulers generalize the traditional role of schedulers by allowing them to allocate CPU time to other schedulers. The *root* scheduler gives CPU time to a scheduler below it in the hierarchy and so on until a leaf of the scheduling tree. The scheduling hierarchy may either be fixed at system build time or dynamically constructed at run time. *CPU inheritance scheduling* [3] probably represents an endpoint on the static vs. dynamic axis: it allows arbitrary user-level threads to act as schedulers by *donating* the CPU to other threads.

Hierarchical scheduling has two important properties. First, it permits multiple programming models to be supported simultaneously, potentially enabling support for applications with diverse requirements. Second, it allows properties that schedulers usually provide to threads to be recursively applied to groups of threads [7].

The comparison of fixed –time allocation, rate-based priority and hierarchical scheduling is given in Table1.

Table 1. Comparison of scheduling algorithms

Features	Fixed – Time Allocation	Rate-Based Priority Scheduling	Hierarchical Scheduling
Nature of Allocation	Static	Dynamic	Dynamic
Priori knowledge of needed resources	Required	Required	Not Required
Scheduling Mechanism	Earliest Deadline First, Round Robin	Priority Based	Not specific
Admission Control Used	No	Yes	No
Monitors Used	No	Rate Regulator. QoS Manager	No
Efficiency	Average	Good	Best
Support Hard and Soft real-time	No	No	Yes

3.6 CPU Schedulers

In the following subsections, we describe in more detail two distinct schedulers.

3.6.1 Rialto scheduler

The scheduler of the Rialto OS is based on three fundamental abstractions:

- *Activities* are typically an executing program or application that comprises multiple threads of control. Resources are allocated to activities and their usage is charged to activities.
- *CPU reservations* are made by activities and are requested in the form: "reserve *x* units of time out of every *Y* units for activity *A*". Basically, period length and reservations for each period can be of arbitrary length.
- *Time constraints* are dynamic requests from threads to the scheduler to run a certain code segment within a specified start time and deadline to completion.

The scheduling decision [6], i.e. which threads to activate next, is based on a pre-computed scheduling graph. Each time a request for CPU reservation is issued, this scheduling graph is recomputed. In this scheduling graph, each node represents an activity with a CPU reservation, specified as time interval and period, or represents free computation time.

For each base period, i.e. the lowest common denominator of periods from all CPU reservations, the scheduler traverses the tree in a depth-first manner, but back tracks always to the root after visiting a leaf in the tree. Each node, i.e. activity that is crossed during the traversal, is scheduled for the specified amount of time.

The execution time associated with the schedule graph is fixed. Free execution times are available for non-time-critical tasks. This fixed schedule graph keeps the number of context switches low and keeps the scheduling algorithm scalable. If threads request time constraints, the scheduler analyzes their feasibility with the so-called *time interval assignment* data structure. This data structure is based on the

knowledge represented in the schedule graph and checks whether enough free computation time is available between start time and deadline (including the already reserved time in the CPU reserve).

Threads are not allowed to define time constraints when they might block—except for short blocking intervals for synchronization or I/O. When during the course of a scheduling graph traversal an interval assignment record for the current time is encountered, a thread with an active time constraint is selected according to EDF [6]. Otherwise, threads of an activity are scheduled according to round-robin. Free time for non-time-critical tasks is also distributed according to round-robin. If threads with time constraints block on a synchronization event, the thread priority (and its reservations) is passed to the holding thread.

3.6.2 SMART scheduler

The SMART scheduler [6] is designed for multimedia applications and is implemented in Solaris 2.5.1. The main idea of SMART is to differentiate between *importance* to determine the overall resource allocation for each task and *urgency* to determine when each task is given its allocation. Importance is valid for real-time and conventional tasks and is specified in the system by a tuple of priority and biased virtual finishing time.

Here, the virtual finishing time [4], as known from fair-queuing schemes, is extended with a bias, which is a bounded offset measuring the ability of conventional tasks to tolerate longer and more varied service delays. Application developers can specify time constraints, i.e. deadlines and execution times, for a particular block of code, and they can use the system notification.

The system notification is an up call that informs the application that a deadline cannot be met and allows the application to adapt to the situation. Applications can query the scheduler for availability, which is an estimate of processor time consumption of an application relative to its processor allocation. Users of applications can specify priority and share to bias the allocation of resources for the different applications.

The SMART scheduler separates importance and urgency considerations. First, it identifies all tasks that are important enough to execute and collects them in a candidate set. Afterwards, it orders the candidate set according to urgency consideration.

In more detail, the scheduler works as follows [26]: if the tasks with the highest value-tuple are a conventional task, schedule it. The highest value-tuple is either determined by the highest priority or for equal priorities by the earliest biased virtual finishing time. If the task with the highest value-tuple is a real-time task, it creates a candidate set of all real-time tasks that have a higher value tuple than the highest conventional task. The candidate set is scheduled according to the so-called best-effort real-time scheduling algorithm.

Basically, this algorithm finds the task with the earliest deadline that can be executed without violating deadlines of tasks with higher value-tuples. SMART notifies applications if their computation cannot be completed before its deadline. This enables applications to implement downscaling. There is no admission control implemented in SMART. Thus, SMART can only enforce real-time behavior in underload situations.

3.6.3 EScheduler

EScheduler, an energy-efficient soft real-time CPU scheduler [6] for multimedia applications running on a mobile device. EScheduler seeks to minimize the total energy consumed by the device while meeting multimedia timing requirements. To achieve this goal, EScheduler integrates *dynamic voltage scaling* into the traditional soft real-time CPU scheduling: It decides *at what CPU speed* to execute applications in addition to when to execute what applications. EScheduler makes these scheduling decisions based on the probability distribution of cycle demand of multimedia applications and obtains their demand distribution via online profiling.

(a) Advantages of EScheduler

1. Scheduling is stable. This stability implies the feasibility to perform our proposed energy-efficient scheduling with low overhead.
2. EScheduler delivers soft performance guarantees to these codecs by bounding their deadline miss ratio under the application-specific performance requirements.
3. EScheduler reduces the total energy of the laptop by 14.4 percent; to 37.2 percent; relative to the scheduling algorithm without voltage scaling and by 2 percent; to 10.5 percent; relative to voltage scaling algorithms without considering the demand distribution.
4. EScheduler saves energy by 2 percent; to 5 percent; by explicitly considering the discrete CPU speeds and the corresponding total power of the whole laptop, rather than assuming continuous speeds and cubic speed-power relationship.

Table 2. Comparison of CPU schedulers

The comparison of CPU schedulers in terms of features is given in Table2.

4. Memory Management

Techniques such as demand-paging and memory-mapped files have been successfully used in commodity OS. However, these techniques fail to support multimedia applications, because they introduce unpredictable memory

Features	SMART	Rialto	EScheduler
Platform	Solaris 2.5.1	Not specific	Mobile device
Time constraints	Dynamic	Dynamic	Static
Scheduling mechanism Based on	Virtual finishing time	Recomputed graph	Dynamic voltage scaling
Admission Control	No	No	No
Support for Hard real time application	No	No	Yes
Scheduling Algorithm	Best-effort real-time	Hierarchical	Proportional share

access times, cause poor resource utilization, and reduce performance. In the following subsections, we present new approaches for memory allocation and utilization, data replacement, and prefetching using application-specific knowledge to solve these problems. Furthermore, we give a brief description of the UVM Virtual Memory System that replaces the traditional virtual memory system in NetBSD 1.4. [5]

4.1 Memory Allocation

Usually, upon process creation, a virtual address space is allocated which *contains* the data of the process. Physical memory [20] is then allocated and assigned to a process and then mapped into the virtual address space of the process according to available resources and a global or local allocation scheme. This approach is also called *user-centered allocation [6]*. Each process has its own share of the resources. However, traditional memory allocation on a per client (process) basis suffers from a linear increase of required memory with the number of processes. In order to better utilize the available memory, several systems use so-called *data-centered allocation* where memory is allocated to data objects rather than to a single process. Thus, the data is seen as a resource principal. This enables more cost-effective data-sharing techniques [16] [18]:

(1**) *Batching*** starts the video transmission when several clients request the same movie and allows several clients to share the same data stream;

(2) B***uffering*** (or *bridging*) caches data between consecutive clients omitting new disk requests for the same data.

(3) S***tream merging*** (or *adaptive piggy-backing*) displays the same video clip at different speeds to allow clients to catch up with each other and then share the same stream.

(4) C***ontent insertion*** is a variation of stream merging, but rather than adjusting the display rate, new content, e.g. commercials, is inserted to align the consecutive playouts temporally;

(5) P***eriodic services*** (or *enhanced pay-per-view*) assign each clip a retrieval period where several clients can start at the beginning of each period to view the same movie and to share resources.

These data-sharing techniques are used in several systems. All buffers are shared among the clients watching the same movie and work like a sliding window on the continuous data [14]. When the first client has consumed nearly all the data in the buffer, it starts to refresh the oldest buffers with new data. Periodic services are used in pyramid broadcasting .The data is split in partitions of growing size, because the consumption rate of one partition is assumed to be lower than the downloading rate of the subsequent partition. Each partition is then broadcast in short intervals on separate channels.

A client does not send a request to the server, but instead it tunes into the channel transmitting the required data. The data is cached on the receiver side, and during the playout of a partition, the next partition is downloaded. However, to avoid very large partitions at the end of a movie and thus to reduce the client buffer requirement, the partitioning is changed such that not every partition increases in size, but only each *n*th partition. Performance evaluations show that the data-centered allocation schemes scale much better with the numbers of users compared to user-centered allocation. The total buffer space required is reduced, and the average response time is minimized by using a small partition size at the beginning of a movie.

The ***memory reservation*** *per storage device* mechanism allocates a fixed, small number of memory buffers per storage device in a server-push VoD server using a cycle based scheduler. [19] In the simplest case, only two buffers of identical size are allocated per storage device. These buffers work co-operatively, and during each cycle, the buffers change task as data is received from disk. That is, data from one process is read into the first buffer, and when all the data is loaded into the buffer, the system starts to transmit the information to the client. At the same time, the disk starts to load data from the next client into the other buffer. In this way, the buffers change task from receiving disk data to transmitting data to the network until all clients are served. The admission control adjusts the number of concurrent users to prevent data loss when the buffers switch and ensures the maintenance of all client services [24].

4.2 Data Replacement

When there is need for more buffer space, and there are no available buffers, a buffer has to be replaced. How to best choose which buffer to replace depends on the application. However, due to the high data consumption rate in multimedia applications, data is often replaced before it might be reused. The gain of using a complex page replacement algorithm might be wasted and a traditional algorithm as. Nevertheless, in some multimedia applications where data often might be reused, proper replacement algorithms may increase performance. The *distance*, the *generalized interval caching* and the SHR schemes, all replace buffers after the distance between consecutive clients playing back the same data and the amount of available buffers [6].

Usually, data replacement is handled by the OS kernel where most applications use the same mechanism. Thus, the OS has full control, but the used mechanism is often tuned to best overall performance and does not support application specific requirements.

S*elf-paging* has been introduced as a technique to provide QoS to multimedia applications. The basic idea of self-paging is to "require every application to deal with all its own memory faults using its own concrete resources". All paging operations are removed from the kernel where the kernel is only responsible for dispatching fault notifications. This gives the application flexibility and control, which might be needed in multimedia systems, at the cost of maintaining its own virtual memory operations. However, a major problem of self-paging is to optimize the global system performance. Allocating resources directly to applications gives them more control, but that means optimizations for global performance improvement are not directly achieved.

4.3 Prefetching

The poor performance of demand-paging is due to the low disk access speeds. Therefore, prefetching data from disk to memory is better suited to support continuous playback of time-dependent data types. Prefetching is a mechanism to preload data from slow, high-latency storage devices such as disks to fast, low-latency storage like main memory [6]. This reduces the response time of a data read request dramatically and increases the disk I/O bandwidth.

Prefetching mechanisms in multimedia systems can take advantage of the sequential characteristics of multimedia presentations. For example, a read-ahead mechanism retrieves data before it is requested if the system determines that the accesses are sequential. The utilization of buffers and disk is optimized by prefetching all the shortest database queries maximizing the number of processes that can be activated once the running process is finished. Assuming a linear playout of the continuous data stream, the data needed in the next period (determined by a tradeoff between the maximum concurrent streams and the initial delay) is prefetched into a shared buffer [15].

In addition to the above- mentioned prefetching mechanisms designed for multimedia applications, more general purpose facilities for retrieving data in advance are designed which also could be used for certain multimedia applications.

The **informed prefetching** and caching strategy preloads a certain amount of data where the buffers are allocated / deallocated according to a global max–min valuation. This mechanism is further developed. Where the automatic hint generation, based on speculative pre-executions using mid-execution process states, is used to prefetch data for possible future read requests.

Moreover, the **dependent-based prefetching** captures the access patterns of linked data structures. A prefetch engine runs in parallel with the original program using these patterns to predict future data references. Finally, an analytic approach to file prefetching is described. During the execution of a process a semantic data structure is built showing the file accesses. When a program is re-executed, the saved access trees are compared against the current access tree of the activity, and if a similarity is found, the stored tree is used to preload files.

Obviously, knowledge (or estimations) about application behavior might be used for both replacement and prefetching. A multimedia object is replaced and prefetched according to its relevance value computed according to the presentation point/modus of the data playout.

4.4 Cache Management

All real-time applications rely on predictable scheduling, but the memory cache design makes it hard to forecast and schedule the processor time. Furthermore, memory bandwidth and the general OS performance have not increased at the same rate as CPU performance. Benchmarked performance can be improved by enlarging and speeding up static RAM-based cache memory, but the large amount of multimedia data that has to be handled by CPU and memory system will likely decrease cache hit ratios [6]. If two processes use the same cache lines and are executed concurrently, there will not only be an increase in context switch overheads, but also a cache-interference cost that is more difficult to predict. Thus, the system performance may be dominated by slower main memory and I/O accesses. Furthermore, the busier a system is, the more likely it is that involuntary context switches occur; longer run queues must be searched by the scheduler, etc. flushing the caches even more frequently.

UVM virtual memory system: The UVM Virtual Memory System replaces the virtual memory object, fault handling, and pager of the BSD virtual memory system; and retains only the machine dependent/independent layering and mapping structures [6]. For example, the memory mapping is redesigned to increase efficiency and security; and the map entry fragmentation is reduced by memory wiring.

In BSD, the memory object structure is a stand-alone abstraction and under control of the virtual memory system. In UVM, the memory object structure is considered as a secondary structure designed to be embedded with a handle for memory mapping resulting in better efficiency, more flexibility, and less conflicts with external kernel subsystems. The new copy-on-write mechanism avoids unnecessary page allocations and data copying, and grouping or clustering the allocation and use of resources improves performance. Finally, a virtual memory-based data movement mechanism is introduced which allows data sharing with other subsystems, i.e. when combined with the I/O or IPC systems; it can reduce the data copying overhead in the kernel.

4.5 Management of other resources

This section takes a brief look at management aspects of OS resources.

4.5.1 Speed Improvements in Memory Access

The term dynamic RAM (DRAM), coined to indicate that any random access in memory takes the same amount of time, is slightly misleading. Most modern DRAMs provide special capabilities that make it possible to perform some accesses faster than others [5]. For example, consecutive accesses to the same row in a page mode memory are faster than random accesses, and consecutive accesses that hit different memory banks in a multi-bank system allow concurrency and are thus faster than accesses that hit the same bank.

The key point is that the order of the requests strongly affects the performance of the memory devices. The most common method to reduce latency is to increase the cache line size, i.e. using the memory bandwidth to fill several cache locations at the same time for each access.

However, if the stream has a non-unit-stride (stride is the distance between successive stream elements in memory), i.e. the presentation of successive data elements does not follow each other in memory, and the cache will load data which will not be used. Thus, lengthening the cache line size increases the effective bandwidth of unit-stride streams, but decreases the cache hit rate for non-streamed accesses. Another way of improving memory bandwidth in memory-cache data transfers for streamed access patterns

4.5.2 Multimedia mbuf

The *multimedia mbuf* (mmbuf) is specially designed for disk-to-network data transfers [6]. It provides a zero-copy data path for networked multimedia applications by unifying the buffering structure in file I/O and network I/O. This buffer system looks like a collection of clustered mbufs that can be dynamically allocated and chained. The mmbuf header includes references to mbuf header and buffer cache header. By manipulating the mmbuf header, the mmbuf can be transformed either into a traditional buffer, that a file system and a disk driver can handle, or an mbuf, which the network protocols and network drivers can understand.

A new interface is provided to retrieve and send data, which coexist with the old file system interface. The old buffer cache is bypassed by reading data from a file into an mmbuf chain. Both synchronous (blocking) and asynchronous (non-blocking) operations are supported and read and send requests for multiple streams can be bunched together in a single call minimizing system call overhead. At setup time, each stream allocates a ring of buffers, each of which is an mmbuf chain.

5. Device Management

A device management system for supporting applications that reside on a multimedia client, the applications interacting with a plurality of stream devices associated with the multimedia client, comprising: a stream manager being configured to identify the plurality of stream devices and store a device identifier for each of said stream devices [6].

The first application being operative to initiate communication between a first stream device and said first application by sending a device identifier to said stream manager, said device identifier indicative of said first stream device; and said stream manager being operative, in response to receiving a device identifier from said first application, to stream data between said first application and said first stream device [6].

A stream device management system is provided for supporting applications that access a variety of stream devices associated with a conventional set-top box. More specifically, the stream device management system includes a stream manager configured to identify a plurality of stream devices and to store a device identifier for each of these stream devices, and a shared memory for storing stream data associated with each of the stream devices.

To initiate communication with a first stream device, a first application sends a device identifier indicative of the first stream device to the stream manager. In response to receiving the device identifier, the stream manager communicates an address for the shared memory associated with the first stream device to the first application. Lastly, the application uses this address to access the stream data.

6. File System

The file system plays a major role in every operating system. In multimedia operating system the file system stores the files with following issues [17]: (1) physical storage device (2) contiguous storage of files that improves the throughput at expense of management issues. (3)The disk scheduling to reduce the seek operation and fair disk [11].

For the multimedia disk scheduling the traditional disk scheduling approaches the substituted by EDF, SCAN-EDF, group-sweeping scheduling, mixed strategy, and continuous media file system. A life span of file system is longer than the execution of the program. The integration of discrete and continuous data needs additional resources. The time requirement is very important in multimedia applications. Thus disk scheduling techniques pay major role providing multimedia data [9].

6.1. Multimedia File System

Due to the need of immense storage and continuous media requirements the traditional tape drives are not feasible to store multimedia data [25]. But storage devices such as CD-ROM, RW-CDROM are used. The continuous media of multimedia system is differing from discrete data in the following conditions [11]:

Real time characteristics: The retrieval, computation and presentation time of continuous media are time independent.

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 data 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 different media at the same time. It not only has to ensure that each medium gets a sufficient share of the resources, but it also has to consider the tight relationships between different streams arriving from different sources. The retrieval of a movie, for example, requires the processing and synchronization of audio and video.

7. Disk Scheduling Algorithms

The overall goal of disk scheduling in multimedia systems is to meet the deadlines of all time-critical tasks. The goal of keeping the necessary buffer space requirements low is loosely related. As many streams as possible should be served concurrently, but aperiodic requests should also be schedulable without delaying their service for an infinite amount of time. The scheduling algorithm has to find a balance between time constraints and efficiency [19].

7.1 Earliest Deadline First

This algorithm is used in CPU scheduling and also it is used in disk scheduling. In this algorithm at every new ready state, the scheduler selects from the tasks that are ready and not fully processed the one with the earliest deadline. The requested resource is assigned to the selected task. At the arrival of any new task, EDF must be computed immediately, heading to a new order, i.e. the running task is preempted and the new task is scheduled according to its deadline. The new task is processed immediately if its deadline is earlier than that of the interrupted task. The processing of the interrupted task is continued according to the EDF algorithm later on. EDF is not only an algorithm for

periodic tasks, but also for tasks with arbitrary requests, deadlines and service execution times[19].In file systems the block of the stream with the nearest deadline would be read first. This results in poor throughput and an excessive seek time; no buffer space is optimized. Further, as EDF is usually applied as a preemptive scheduling scheme, the costs for preemption of a task and scheduling of another task are considerable.

7.2 SCAN-Earliest Deadline First Algorithm

The SCAN-EDF is the combination of SCAN and EDF mechanism. The nearest seek time will be read first [24]. If there is more than read with same seek time, it will be read based on SCAN direction. This optimization can be applied to the reads with same seek time. When there is more than one reads with same deadline, they are grouped on the basis of their finish time. Buffer space is not optimized. The throughput is larger than EDF.

7.3 Group Sweeping Scheduling

With Group Sweeping scheduling, requests are served in cycle or in round robin manner. To reduce the disk arm movements, a set of n streams is divided into g groups. Individual streams are within a group are served according to scan technology. There is no fixed time to serve the streams. If the SCAN scheduling is applied to the streams without grouping the playout of a stream cannot be until the previous stream finish its payload. As the buffers can be reused for each group the playout of each stream starts at end where the first retrieval takes place.

7.4 Mixed strategy

The mixed strategy is based on the shortest seek and the balanced strategy [25].The data retrieved from the disk is transferred to the buffer allocated for the respective data stream. In this algorithm the data block which is closest is served first. The employment of shortest seek follows two criteria (1) the number of buffers for all the processes should be balanced and (2) overall require bandwidth should be sufficient to all active processes.

7.5 Continuous Media File System

The Continuous Disk Scheduling is a non preemptive scheduling scheme designed for the continuous media file system. The notion of slack time is introduced here. The slack time is the time duration for which the CMFS is free to do non real time operations.

Table 3: Comparison of the disk scheduling techniques

Properties	Real time processing	Throughput	Seek time	Buffer
EDF	Yes	Poor	Excessive	No optimization
SCAN-EDF	Yes	Higher	Minimum	No optimization
GSW	Yes	Higher	Minimum	No optimization
Mixed Strategy	Yes	Maximum	Minimum	Optimization of buffer
CMFS	No	Maximum	Minimum	Optimization

8. Conclusion

This article gives an overview of the OS support for multimedia applications. This is an active area, and a lot of valuable research results have been published. Thus, we have not discussed or cited *all* recent results, but tried to identify the major approaches and to present at least one representative for each. The various currently available CPU scheduling mechanisms, along with specialized CPU schedulers are discussed.

The memory management techniques along with VoD memory model give an overview of the memory management in Multimedia operating systems. The device management techniques are briefed. We have also discussed the various file management techniques available. The various disk scheduling algorithms like EDF, SCAN-EDF, Group Sweeping, mixed strategy and continuous Media file system are also discussed along with their comparisons.

References

[1] C. Lu, J. A. Stankovic, G. Tav, and S. H. Son. "The design and evaluation of a feedback control EDF scheduling algorithm," In proc of the 20th IEEE real time systems symp., Dec 1999.

[2] D. C. Steere, A. Goel, J. Gruenburg, D. McNamee, C. Pu, and J. Walpole, "A feedback-driven proportion allocator for real rate scheduling," In proc of the 3rd symp on operating systems design and implementation , new Orleans, LA, Feb1999.

[3] B. Ford and S. Susarla, "CPU inheritance scheduling," In proc of the 2nd symp on operating systems design and implementation, Oct 1996.

[4] K. J. Duda and D. C. Chriton, "Borrowed- virtual time scheduling supporting latency – sensitive threads in a general purpose scheduler," In proc of the 17th ACM symp on operating systems principles, Kiawah Island, Dec 1999.

[5] T. Plagemann, V. Goebel, P. Halvorsen, O. Anshus. "Operating system support for multimedia systems," Computer Communications, 23 (2006), 267–289.

[6] T. Plagemann, V. Goebel, P. Halvorsen, O. Anshus. "Operating system support for multimedia systems," Computer Communications 23 (2000) 267–289.

[7] Pawan Goyal, Xingang Guo, and Harrick M. Vin. "A Hierarchical CPU Scheduler for Multimedia Operating Systems," Proceedings of the second USENIX symposium on Operating systems design and implementation, pp: 107-121, 1996.

[8] Daniel Alexander Taranovsky, "CPU Scheduling in Multimedia Operating Systems,"Research Report, 1999.

[9] Gifford, D W and O'Toole, J W 'Intelligent file systems for object repositories', Operating Systems of the 90s and Beyond, Int. Workshop, Dagstuhl Castle, Germany (2007), 20-24.

[10] P. A. Janson, "Operating Systems, Structures and Mechanisms," Academic Press, Orlando, FL

Tanenbaum, A S Operating System, Design and Implementation. Prentice-Hall, 2008.

[11] Cliffs Englewood, S. J. Mullender, 'Systems of the nineties-Distributed multimedia systems; systems of the 90s and beyond', Int. Workshop, 1999.

[12] Castle Dagstuhl, R. Steinmetz, "Data compression in multimedia computing: principles and techniques," Multimedia Systems, Vol 1 No 4, pp 166-172.

[13] R. Steinmetz, "Data compression in multimedia computing: standards and systems," Multimedia Systems, Vol 1 No 5, 1994.

[14] Lougher, P and Shepherd, D 'The design of a storage service for continuous media', The Computer J, Vol 36 No 1, pp 32-42, 1993.

[15] J. Gemmell, and S. Christodoulakis, "Principles of delay sensitive multimedia data storage and retrieval," ACM Trans. Infor. Syst., Vol IO No 1, January 1992.

[16] Rangan, P V, Klppner, T and Vin, H W, 'Techniques for efficient storage of digital video and audio', Proc. Workshop on Multimedia Information Systems, Tempe, AZ , February 2002.

[17] P. V. Rangan, and H. M. Vin, "Designing file systems for digital video and audio," Proc. 13th ACM Symposium on Operating Systems Principles, Monterey CA Operating Systems Review, Vol25 No 5, Oct., 1991.

[18] P. V. Rangan, and H. M. Vin, "Techniques for efficient storage of digital video and audio," Comput. Commun., Vol 16, pp 168-176, 2003.

[19] A. Karmouch, Wang, and Yea, "Design and Analysis of a Storage Retrieval Model for Audio and Video Data," Technical Report, Multimedia Information Systems, Department of Electrical Engineering, University of Ottawa, Canada, 1994.

[20] M. L. Dertouzos, "Control robotics," The Procedural Control of Physical Processing. Information Processing 74, North Holland, pp 807-813.

[21] S. Krakowiak, "Principles of Operating Systems," MIT Press, Cambridge, MA, 2008.

[22] J. Peterson, and A. Silberschatz, "Operating System Concepts," Addison-Wesley, Reading, MA, 1983.

[23] R. Steinmetz, and K. Nahrstedt, "The Fundamentals in Multimedia Systems," Prentice-Hall, Englewood Cliffs, NJ, February 1995.

[24] Y. N. Doganata, and A. Tantawy, "A cost/performance study of video servers with hierarchical storage," IEEE Proc. Int. Conf. Multimedia Computing and Systems, Boston, MA , 2005.

[25] Ralf Steinmetz, "multimedia file systems: approaches for continuous media disk scheduling," computer communications, volume 18, number 3, 1995.

[26] Sity Jason Neih and Monica S. Lam, "Computer Systems Laboratory," Stanford University, implementation and Evaluation of SMART:A Scheduler for Multimedia Applications.

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