Real-Time Filesystems

Guaranteeing Timing Constraints for Disk Accesses in RT-Mach

Anastasio Molano, Kanaka Juvva and Ragunathan Rajkumar
Real-Time and Multimedia Laboratory[1]
Department of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15146
{amolano, kjuvva, raj+}@cs.cmu.edu

Abstract

Traditional real-time systems have largely avoided the use of disks due to their relative slow speeds and their unpredictability. However, many real-time applications including multimedia systems and real-time database applications benefit significantly from the use of disks to store and access real-time data. In this paper, we investigate the problem of obtaining guaranteed timely access to files on a disk in a real-time system. Our study focuses on several aspects of this problem of providing a real-time filesystem. First, we consider the use of two real-time disk scheduling algorithms: earliest deadline scheduling and just-in-time scheduling, a variation of aperiodic servers for the disk. The latter algorithm is designed to improve disk throughput that can be hurt when a real-time scheduling algorithm such as EDF is applied directly. Admission control policies with practically acceptable properties of performance and usability are provided. Next, we design and implement a real-time filesystem on the RT-Mach microkernel-based system running a real-time shell. The new interface we develop is based on RT-Mach's resource reservation paradigm and provides guaranteed and timely access for multiple concurrent applications requiring disk bandwidth with different timing and volume requirements. Finally, we perform a detailed performance evaluation of the real-time filesystem including its raw performance. We show the following positive but rather surprising result: our real-time scheduling filesystem not only provides guaranteed and timely access but also does so at relatively high levels of throughput. Traditional disk scheduling algorithms offer completely unacceptable file access latencies for real-time applications and do so only at slightly higher throughput.

1. Introduction

Real-time and embedded systems are used in many application domains including feedback control systems, (manufacturing) process control, robotics, air traffic control, avionics, target-tracking, real-time object recognition, discrete assembly and vehicle navigation. Many of these systems such as air traffic control tend to be distributed in nature and also need to satisfy end-to-end timing requirements which span processor boundaries. While real-time applications are many, traditional real-time systems have largely avoided the use of disks. This is in part because they may be relatively slow for some real-time applications. However, many real-time applications can benefit from the use of disks to store and access real-time data (such as real-time database applications). Unfortunately,

- disk accesses can introduce unpredictable latencies, and
- disk access requests must also be managed in conjunction with processor scheduling.

An emerging set of commercially and strategically important real-time applications demand real-time disk accesses. These applications include real-time databases, C^3I systems, multimedia applications, on-demand services and intranet servers for in-house training and continuing education.

On the processor side, fixed priority algorithms allow a mix of tasks with different periodicity, and hence the disk subsystem must do too. The disk scheduling problem has not been studied extensively in the general context. We focus on a general computing platform (including desktops) where multiple applications with and without a range of timing constraints and file I/O constraints can be running concurrently. Finally, we extend the processor reservation model in RT-Mach to include guaranteed and timely access to disk bandwidth. In other words, an application can request a portion of the disk bandwidth with pre-specified timing constraints and if the request is accepted by the admission control policy, the application is guaranteed to obtain its requested share of the disk bandwidth on a timely basis.

1.1. Comparison with Related Work

A wealth of resource management schemes and scheduling algorithms exists from which one can draw. Some of our goals (such as resource centricity) are similar to those of Microsoft Research's Rialto kernel among others. The reservation model also has its counterparts in network reservation protocols as used in ATM and RSVP. However, the operating system problem seems more complex in one sense since inherently different resource types must be dealt with, while networks essentially deal with

[1]This research was supported by the Defense Advanced Research Projects Agency in part under contract number E30602-97-2-0287 and in part under contract number F30602-96-1-0160. Mr. Molano was funded by a research grant from the Community of Madrid and by the National R&D Program of Spain under contracts TIC96-0982 and TIC97-0438.

one type (namely network packets). Work described in [1, 2, 7, 8] study various aspects of real-time disks and filesystems. The approach used in [1] is a variation of the Scan algorithm in which the SCAN direction changes towards the Earliest Deadline First request only if the deadline is considered to be met. At any scheduling point the request with the Earliest *Feasible* deadline is chosen for service and the disk head scans towards it. Chen et al. [7] propose two algorithms which combine deadline information and disk service time information. They show that these algorithms minimize transaction loss ratio in real-time database applications. Both [1] and [7] are proposed and studied under soft deadline schemes. A scheduling model close to ours is found in [2] but its approach is one of using fixed priority scheduling, minimizing blocking through the use of "chunking" and using a static task set. Also, only simulation studies were carried out. In contrast, we use dynamic priority scheduling, exploit blocking instead of minimizing it and evaluate an implementation within our resource framework.

A multimedia storage server design and implementation is reported in [15]. In this scheme, the disk request queue is divided into two different queues, one for real-time threads and another for non-real-time threads. Both queues were scheduled following C-Scan with the real-time queue taking precedence over the non-real-time queue. The scheme is called a constant-rate access server (CRAS) and accurately reflects the attempt to only support a single rate of service at the disk. However, in a general real-time or multimedia application, many different applications would need to access the disk at different rates. Our scheme employs a different period for each real-time activity, and guarantees a "share" of that period to the activity. We show that the processor scheduling paradigm can indeed be adapted to be useful in the context of disk bandwidth scheduling. Our scheme can be used in conjunction with processor scheduling.

RT-Mach previously offered only traditional non-real-time support for real-time disk scheduling. In other words, disk requests were not served based on priorities or deadlines. As a result, a real-time application such as a video player was forced to read the file completely from disk, store it in memory and then play it back. The playing was treated as the real-time aspect and since strong processor scheduling support was available, many real-time and multimedia applications with concurrent audio and video streams could co-exist and meet their deadlines. The disk access portions, however, continued to be done in non-real-time. In this paper, we add real-time filesystem support to RT-Mach in a way that is completely compatible with its resource reservation model. Our detailed set of experiments presented later in this paper show that guaranteed timely access to files on disk is possible, and that this can be achieved at relatively low cost in terms of disk throughput.

1.2. A Brief Overview of RT-Mach

The primary goal behind the RT-Mach effort [9][2] is to design, develop, demonstrate and distribute an integrated framework that encompasses task scheduling, virtual memory management, synchronization mechanisms, interprocess communications, real-time disk scheduling, network protocol processing and distributed coordination. The framework is intended to ease and facilitate the development of predictable real-time applications. RT-Mach adopts the *resource reservation model* [10, 12], which allows applications to specify their resource demands independent of the scheduling algorithm actually used in the kernel. A real-time socket library provides predictable *and* efficient protocol processing for networked applications. A schedulability analysis tool and a distributed monitoring facility provide support for the analysis and debugging of distributed real-time applications. Extensive support is available for multimedia devices including full-duplex audio, real-time video capture, mobile networking with networking support.

1.2.1. Resource Kernels

A *resource kernel* [12] is defined to be one which provides timely, guaranteed and protected access to system resources. The resource kernel allows applications to specify only their resource demands leaving the kernel to satisfy those demands using hidden resource management schemes. This separation of resource specification from resource management allows OS-subsystem-specific customization by extending, optimizing or even replacing resource management schemes. As a result, this resource-centric approach can be implemented with any of several different resource management schemes.

The resource kernel gets its name from its resource-centricity and its ability to

- apply a uniform resource model for dynamic sharing of different resource types,
- take resource usage specifications from applications,
- guarantee resource allocations at admission time,
- schedule contending activities on a resource based on a well-defined scheme, and
- ensure timeliness by dynamically monitoring and enforcing actual resource usage.

1.2.2. RT-Mach as a Resource Kernel

RT-Mach is a resource kernel, and attains its capabilities by reserving resources for applications requesting them, and tracking outstanding reservation allocations. Based on the timeliness requirements of reservations, the resource kernel prioritize them, and executes a higher priority reservation before a lower priority reservation if both are eligible to execute.

Explicit Resource Parameters: The RT-Mach resource reservation model employs the following parameters: computation time C every T time-units for managing the net utilization of a resource, a deadline D for meeting timeliness requirements, a starting time S of the resource allocation, and L, the life-time of the resource allocation. We refer to these parameters, $\{C, T, D, S$ and $L\}$ as explicit parameters of our reservation model. The semantics are

[2] URL: "http://www.cs.cmu.edu/~rtmach"

simple, well-known in the real-time community and are as follows. Each reservation will be allocated C units of usage time every T units of absolute time. These C units of usage time will be guaranteed to be available for consumption before D units of time after the beginning of every periodic interval. The guarantees start at time S and terminate at time $S + L$.

Implicit Resource Parameter: The resource kernel implicitly derives, tracks and enforces the implicit B parameter for each reservation in the system. B represents the maximum (desirably bounded) time that a reservation instance must wait for lower priority reservations while executing. If its B is unbounded, a reservation cannot meet its deadline. Priority (or reservation) inheritance is applied when a reservation blocks, waiting for a lower priority reservation to release (say) a lock. As we shall see in Section 2.6, this implicit parameter B can also be used to deliberately introduce priority inversion in a controlled fashion to achieve other optimizations.

1.3. Organization of the Paper

The rest of this paper is organized as follows. In Section 2, we motivate and describe in detail the disk scheduling algorithms we adopt. We present an aperiodic server-like disk scheduling algorithm called "Just-in-Time" disk scheduling. In Section 3, we present the design and implementation of a real-time filesystem in the RT-Mach operating system environment. In Section 4, we perform a set of detailed experiments to evaluate the timeliness and throughput properties of our real-time filesystem, and compare it with the traditional Scan disk scheduling algorithm. Finally, in Section 5, we conclude with some remarks.

2. Disk Bandwidth Resource Management

Many real-time applications like real-time databases and C3I systems can benefit from having access to disks. Desktop multimedia systems also need to read from (or write to) disk storage relatively large volumes of video and audio data. In addition, these streams represent continuous media streams, and must therefore be processed by the disk subsystem in real-time. In other words, it would be very useful in practice if disk bandwidth can also be guaranteed in addition to managing processor cycles.

In this section, we present a simplistic disk scheduling algorithm based on earliest deadline scheduling. We then improve the algorithm by exploiting "slack" in the reservations to obtain a hybrid of earliest deadline scheduling and a traditional scan algorithm. Our evaluations of these schemes in Section 4 show that guaranteed disk bandwidth reservation can be obtained at only a small loss of system throughput.

2.1. Important Considerations

We now outline some important considerations which need to influence the design of a real-time filesystem:

- *Preemptibility issues*: Once a request is issued to the disk drive, it will not be preempted until it has finished, even if there are higher priority disk requests waiting for service. The time that a higher priority disk request may wait until being serviced is bounded by the longest disk request, which can still be rather long. The duration of the non-preemption window must ideally be small and perhaps even dynamically adjustable depending on the workload.
- *With Preemption*: by implementing fine-grained accesses to the disk, a higher priority disk request can preempt a lower priority disk request midway through the processing of its larger request. Rather than sending the whole disk request in one SCSI command (for example), one can send smaller disk requests successively with several SCSI commands, so that they can be preempted at smaller intervals. We shall study the impact of the disk block sizes on the disk throughput in Section 4.
- *Heterogeneity of the workload*: Consider very heterogeneous workloads where there are many small requests with deadlines, but they are prevented from execution due to larger low priority disk requests. Examples of such systems are heterogeneous C3I real-time databases.

  Consider homogeneous workloads such as multimedia storage servers, where *all* the requests are periodic ones. SCAN-based schemes are the most effective under these considerations since they avoid expensive disk head movements (seeks).

2.2. Filesystem Bandwidth Specification

Our resource specification model for disk bandwidth is identical to that of processor reservation in RT-Mach. In other words, a disk bandwidth reservation must specify a start time S, a processing time C to be obtained in every interval T before a deadline of D. The processing time C can be specified as # of disk blocks (as a portable specification) or in absolute disk bandwidth time in native-platform specification.

In the rest of this paper, we shall use the terms "filesystem reservation", "fs reserves", "filesystem bandwidth reservation" and "disk bandwidth reservation" interchangeably,

2.3. Resource Specification Notation

Let the set of n reservations requiring resource reservation be denoted as $\tau_1, \tau_2, \cdots, \tau_n$. Each reservation τ_i needs to obtain C_i units of time every T_i units of time. In addition, the C_i units of resource time must be available at or before D_i in each periodic interval separated by T_i.

2.4. Scheduling Policy

Instances of a disk bandwidth reservation become eligible to execute *every* T_i units (at times S_i, $S_i + T_i$, $S_i + 2T_i$, $S_i + 3T_i, \cdots$). Consider an instance which arrives at time $S_i + nT_i$. This instance has a deadline of $S_i + nT_i + D_i$. Similarly, all instances of all outstanding disk bandwidth reservations have corresponding deadlines. After each disk block access is completed, the disk scheduler makes another scheduling decision. It picks the next ready reservation instance with the earliest deadline and issues a disk access command corresponding to that instance's next disk access request. If there are no pending requests, the disk remains idle.

2.5. Admission Control

Our simplest disk head scheduling scheme employs the earliest deadline scheduling algorithm (EDF) [6]. Since EDF is a preemptive scheduling algorithm, a higher priority reservation must be able to preempt a lower priority reservation. However, instantaneous preemptions are not possible in disk scheduling. An ongoing disk block access must complete before the next highest priority disk block access request can be issued. This introduces a blocking (priority inversion) factor of a single filesystem block access when $D_i = T_i$ (as per [13]). When $D_i < T_i$, the required earlier completion time ($T_i - D_i$) is added to the blocking factor.

For the *rate-monotonic* [6] and the *deadline-monotonic* [5] fixed priority schemes, the completion time CT_i of a reservation τ_i with a resource allocation can be computed using a recurrence relation [3, 16]. If this completion time is less than the deadline for all reservations including the incoming one, a reservation request can be admitted. Else, it will be rejected.

For the *earliest deadline first* policy, the reservations requiring disk bandwidth are ordered according to rate-monotonic priorities, (i.e. $T_i < T_{i+1}$). The number of disk blocks to be read by an instance of τ_i is denoted as L_i. The admission control test for our disk bandwidth reservation scheme is given by

$$BU_{max} + \sum_{i=1}^{n} U_i \le 1.0$$

where, U_i is the utilization of task τ_i given by

$$U_i = \frac{t_{IPC-in} + t_{FSB}L_i + 2t_{Seek} + 2t_{Rot} + t_{IPC-out}}{T_i}$$

t_{FSB} is the time to read (or write) one single file system block given by

$$t_{FSB} = t_{fs-overhead} + \frac{\text{size of FSB in bytes}}{\text{disk BW in bytes/sec}}$$

where,

t_{seek} is the maximum seek time of the disk head,
t_{rot} is the maximum rotational latency for the disk,
$t_{IPC\text{-}in}$ is the overhead for invoking a read operation,
$t_{IPC\text{-}out}$ is the overhead for the result of the read op, and
$t_{fs\text{-}overhead}$ is the overhead incurred in the filesystem itself.
BU_{max} is the maximum utilization of the priority inversion encountered by all guaranteed reservations, given by

$$BU_{max} = max\left(\frac{B_1}{T_1}, \frac{B_2}{T_2}, \cdots, \frac{B_n}{T_n}\right)$$

and B_i is the priority inversion duration encountered by reservation τ_i given by

$$B_i = t_{FSB} + T_i - D_i$$

It must be noted that we add $t_{Seek} + t_{Rot}$ only twice to each periodic instance of a disk bandwidth reserve, one for switching the request stream and switching it out once. This seems logical particularly since this is how context-switching on the processor is handled. If all the disk blocks that an application needs are sequentially numbered, this situation is analogous to the processor case. Unfortunately, this may not be sufficient to guarantee disk access deadlines for all applications at all times. Many applications will create files or read file blocks not in sequence but potentially in some (pseudo-)random order. For such applications, this "context-switching factor" is not sufficient in the worst case. However, given that both the maximum seek time and the rotational latency can be fairly large, including them would typically mean that only a small portion of disk requests would be able to pass the admission control scheme. In Section 4, we show that under non-contiguous layout such an assumption does lead to deadlines being missed, but the number of deadlines missed is very small, that it perhaps will not be a concern for many applications. If an application cannot tolerate any deadline miss, then it has to include the context-switching for potentially each disk block it accesses, or use contiguously allocated files. There are three mitigating factors:

- A real-time filesystem can (as ours does) allocate disk blocks as contiguously as possible.
- Utilities (such as *tunefs* on Unix systems, filesystem optimizers) can be used to choose the block allocation policy within a filesystem and/or relocate file blocks so that they are contiguous.
- With contiguous layout, our current admission control test guarantees the requested bandwidth if the test succeeds. The test will succeed for relatively high bandwidths if the periods of the requests are large compared to the seek/latency factor used in the test. For example, suppose one wants to guarantee (say) 200 KBytes per second. Using parameters of the disk we used, one will consume 100% of utilized bandwidth (with overhead) by reserving 20 KBytes from disk every 100 ms. If instead, we transform the periods and reserve 200 KBytes per 1 second, the utilization is just 32.5%. With scaled up periods, more bandwidth can be guaranteed for contiguous files, or guarantees can be provided for non-contiguous files by including additional seek and rotational latencies for each non-contiguous block access. On the other hand, bigger application buffers will be necessary and longer end-to-end delays will be incurred.

2.6. Exploiting 'B': Just-In-Time Disk Scheduling

The earliest deadline disk scheduling blindly picks the next block with the earliest deadline irrespective of the current position of the disk head. Since the physical movement of the disk head and the disk's rotational latencies constitute significant durations of time, such dynamic scheduling can result in significant disk subsystem throughput reductions particularly under heavy disk traffic. The reductions can be directly attributed to the time wasted by the disk head moving from one end to another and the disk's rotational time. In summary, the deadlines are preferred over a block's physical location.

Traditional scan algorithms, in contrast, re-order the disk request queue such that the block closest to the current head position (in the direction of movement) is accessed next. As a result, a disk request which just arrived can be serviced before another disk request which has been waiting for a long time just because the latter is farther away from the head position. To summarize, the physical block location is favored over timeliness.

The earliest deadline scheduling algorithm and the scan algorithm are therefore at odds with one another. Fortunately, a hybrid scheme which can obtain all the benefits of the earliest deadline scheduling algorithm and at least part of the benefits of the scan algorithm is possible. Using the notion of slack-stealing inherent in all aperiodic servers [14], a scheduling algorithm can exploit the slack available to higher priority tasks to schedule accesses of other disk blocks which are closer to the current head position.[3]

A brief description of the just-in-time disk scheduling algorithm is as follows. The maximum "slack" available to each disk reservation is computed whenever a new request is admitted (or an existing reservation is deleted). At runtime, if the current slack of higher priority reservations is non-zero, another unreserved (or lower priority reserved) request can be scheduled if closer to the disk head. If slack is stolen, the slack of higher priority reservations is reduced by one. This process is then repeated. If the slack of a high priority reservation goes to zero, it will be serviced independent of its location.

2.7. "Just-in-Time" Slack-Stealing Algorithm for Disk Bandwidth

Let $K_i(t)$ denote the maximum "slack" in units of time available for each reserved disk data stream, and let K_i denote the maximum slack available in units of disk blocks. We have,

$$K_i(t) = D_i - T_i \sum_{j=1}^{i} (U_j)$$
$$K_i = \left\lfloor \frac{K_i(t)}{t_{FSB}} \right\rfloor$$

where U_j and t_{FSB} are as defined earlier.

2.8. The JIT Replenishment Algorithm

Data Structures: The maximum slack K_i of each reservation τ_i is determined at admission control time. At runtime, the current slack k_i of each instance of τ_i is also maintained. The current slack k_i of each new instance of τ_i is initialized to K_i. At any given time, let the set of undepleted reservations be $\{\tau_{valid\_reserves}\}$, and let the set of indices corresponding to the reservations in $\{\tau_{valid\_reserves}\}$ be {J}.

Usage by an unreserved activity: An unreserved activity can "steal" early access to a disk block from a reservation if $\forall j, j \in \{J\}, k_j > 0$. When such stealing happens, the current slack factors are updated as $\forall j, j \in \{J\}, k_j \leftarrow (k_j - 1)$.

[3] Such "slack-stealing" has been done in the context of processor scheduling theory in order to provide better response to aperiodic activities [4]. The optimization, cost functions and implementation tradeoffs seem to be different for the processor and the disk, however.

Stealing by a depleted reservation: A depleted reservation τ_i waiting for resource replenishment can steal early access to a disk block from a reserved reservation if $\forall j, j \in \{J\}, k_j > 0$. When such stealing is allowed to occur, the current slack factors are updated as $\forall j, j \in \{J\}, k_j \leftarrow (k_j - 1)$.

Stealing by an undepleted reservation: An undepleted reservation τ_i can also steal early access to disk blocks from a higher priority undepleted reservation if $\forall j, (j \in \{J\}) \wedge (j \leq (i-1)), k_j > 0$. When such stealing is allowed to occur, $\forall j, (j \in \{J\}) \wedge (j \leq (i-1)), k_j \leftarrow (k_j - 1)$.

3. Disk Bandwidth Reservation in Real-Time Mach

In this section, we describe the design and implementation of the real-time filesystem supporting disk bandwidth reservation in Real-Time Mach.

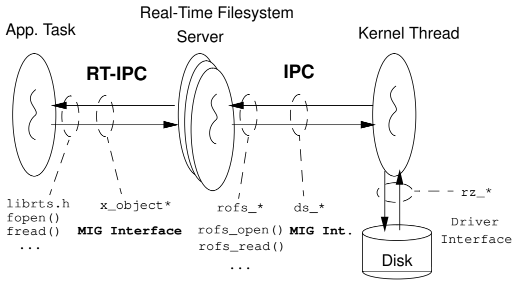


Figure 3-1: Real-Time File System Service Layers

3.1. The Architecture of the Reserved Filesystem

The architecture of our bandwidth-reserved filesystem follows a "traditional" scheme. A *Real-Time File Server* running on top of our resource kernel (based on the RT-Mach microkernel) manages the reserved real-time filesystem. RT-FS has multiple worker threads which receive and process filesystem access requests from real-time clients. Each worker thread stores the incoming request it is processing into a common I/O request queue. The worker thread responsible for issuing the current disk block access waits for its completion. It then awakens, and determines the next request based on the scheduling policy above. If the next request corresponds to another worker thread, that thread is signaled. Else, the worker thread continues to service its remaining disk access requests, if any. Priority inheritance is applied where necessary across the threads to avoid priority inversion problems.

Disk bandwidth reserves have been implemented within RTS (Real-Time server), which runs on top of the RTMach microkernel. RTS supports a process manager, a file system, a device manager, and a simple command interpreter. The most salient feature of RTS is the use of RT-IPC, a realtime interprocess communication mechanism to avoid priority inversion at the server side that includes priority based message queue ordering, establishment of the message buffer size, and control of the handoff policy [11].

The file system runs in the context of realtime worker threads which execute concurrently and attend user requests sent by means of RT-IPC messages (see Figure 3-1. The file system invokes I/O operations using the RT-Mach device driver interface (ds_routines()) implemented as IPC messages to the kernel. The disk driver executes in kernel context, and performs I/O to the disk controller for the actual transfer to the disk. This I/O interface is synchronous in that we wait for the completion of the request before issuing the next one to the disk. The file system has been modified and extended to support the file system reserves mechanism. A new module has been added that maintains file system reserves and implements the interface for creating, terminating and starting them. Additionally it manages the replenishment of file system reserves by means of a replenishment thread handler as it is described below. MIG is the Mach Interface Generator, an Intermediate Data Language for inter-process communications with Mach and RT-Mach.

The file system reserve scheduling mechanism manages all the invoking disk I/O requests, scheduling reserved disk requests under EDF/JIT, and unreserved ones or depleted requests under SCAN. The requests are depleted when they have consumed all their allocated budget (measured in number of disk accesses). The file system accounts for the current resources consumed for each of the pending requests updating their disk usage counter each time they have issued a request to the disk. Within the file system itself, new I/O operation primitives have been added to include the file system reserve control mechanism. The reserved file system maintains two queues of requests, one for reserved requests in reserved mode (with budget different from zero), and other for both unreserved requests and depleted requests (reserved ones which budget have reached zero) . The requests queue is protected against concurrent access by means of a mutex and a condition variable that allow the synchronization between all the worker threads executing within the file system concurrently.

The scheduler within the RT filesystem has been implemented as a cooperative scheduler, in the sense that it is executed by the current active thread, which will give control to the next one following the chosen scheduling policy. Worker threads invoking a I/O operation sleep in a condition variable till being signalled as the next request by the active thread currently executing the scheduling algorithm.

The file system is executed in a loop, sending a command to the disk controller each iteration to read/write a file system block. After finishing the I/O operation, the current active thread executes the scheduling algorithm to find out which one should be at this time the next request to execute. It updates its own disk bandwidth reserve budget by decreasing in one unit the number of remaining disk accesses. It checks the queue of unreserved requests to see if some depleted request has been replenished. In this case it removes the depleted request from the queue of unreserved requests and store it in the queue of reserved requests.

It executes the scheduling algorithm (EDF, EDF/JIT, SCAN) choosing the next request based on the current scheme. File system block numbers are used as a reference of physical placement on disk. If the next request is not the

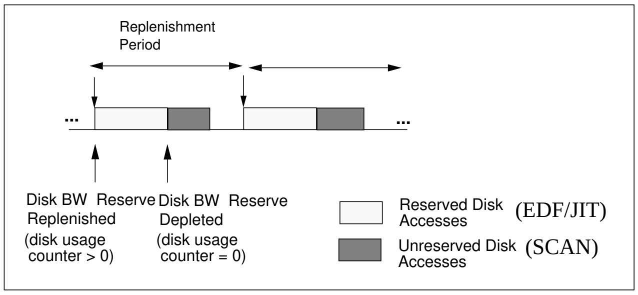


Figure 3-2: Disk BW Reserves Replenishment Scheme

current request then the thread executing the current request signals the thread that will execute the next one, and it is put to sleep after that. The system detects that some reservation has to be depleted when its disk usage counter reaches zero, and it detects when it has to be replenished by obtaining updates from a replenishment handler. A periodic timer and a replenishment timer are allocated to each of the active disk bandwidth reserves.

3.2. The API

A synopsis of the application programming interface we use is presented in Figure 3-3.

```
/* Real-Time File System Services for *
 * disk bandwidth reservation         */

disk_bw_reserve_create(*disk_bw_reserve);
disk_bw_reserve_request(disk_bw_reserve,
  period, start, deadline, bytes_reserved);
disk_bw_reserve_terminate(disk_bw_reserve);
disk_bw_reserve_set_name(disk_bw_reserve, name);
disk_bw_reserve_set_attribute(disk_bw_reserve,
   flavor, new_attr, new_attr_count);
disk_bw_reserve_get_attribute(disk_bw_reserve,
   flavor, old_attr, *old_attr_count);
```

Figure 3-3: Core Application Programming Interface for the Real-Time Filesystem

4. Performance Evaluation

We conducted a series of experiments to study the efficiency of the disk scheduling algorithms and their impact on applications. We start with a performance study of the raw disk and its sensitivity to the size of the data blocks read from the disk. We then summarize our results both quantitatively and qualitatively using two workloads, one synthetic and one real. In the synthetic workload, a single real-time thread uses both the CPU and the disk and therefore needs timeliness guarantees on both. Competing background priority threads accessing the disk continuously attempt to disrupt the timing behavior of this real-time application. Next, we test the real-time filesystem on a video playback application using a QuickTime movie player, which reads a movie file from a disk, processes it and displays the frames on the screen. Heavy competition from non-real-time threads continues in this experiment.

4.1. Disk Parameters

The disk we used for our performance evaluation was a 1 GB drive with a max. seek latency of 24 *ms*, and a maximum rotational latency of 14 *ms*. The disk blocks were 512-bytes in size while the filesystem used 4 KB blocks.

The time to access a disk block is on the order of 3 - 6 *ms*. The RT-Mach microkernel, Version RK97a, was used on Pentium 120 MHz workstations. The RTS environment described in the earlier section was used to support the filesystem utilizing the disk bandwidth reserves. The period of the real-time timer was set up to be 1 *ms*, a resolution that we found to be sufficient for our experiments.

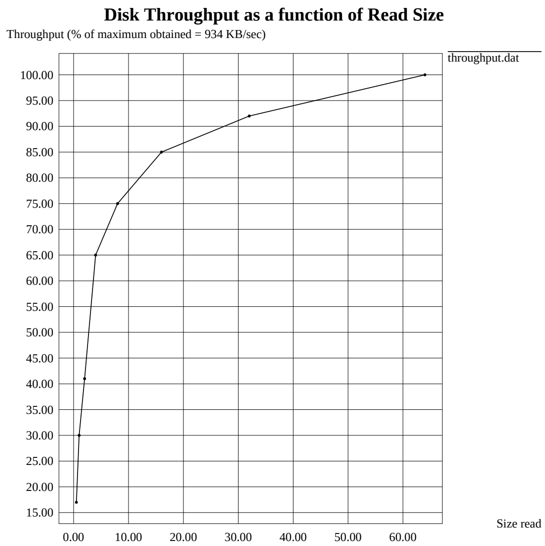


Figure 4-1: The Raw Throughput Obtained from the Disk as a function of data bytes read (in KB)

4.2. Overhead Components of Raw Disk Access

Size (KB)	Filesystem	ds_invoke	IO Transfer	ISR	DS_return
32	1145	36	26414	9	182
16	574	32	14425	9	144
8	336	32	8496	9	130
4	178	31	5111	9	150
2	93	30	3800	8.8	150
1	67	30	3300	8.5	160
0.5	50	30	2350	8	178
Comments	f(Size)	constant	f(size)	constant	constant

Figure 4-2: Overhead Components of Raw Disk Read - μs

We implemented and tested a Raw and Synchronous interface to the disk under RT-Mach to understand and measure the disk's behavior in terms of the size of the data blocks read from disk and the overhead encountered in various levels of the software. In specific, the interface we built bypassed the file system and accessed blocks on the disk in sequence. Each request must be satisfied before the next one can be submitted. The synchronous interface was then used by the real-time filesystem server to issue disk read requests one block at a time.

We simulated accesses with different granularity and with different layout (contiguous and non-contiguous accesses). The raw throughput obtained from the disk for various granularities of the data block is plotted in Figure 4-1. Higher the size of the data blocks read, higher is the efficiency obtained by the underlying disk controller. That is, disk throughput increases with the size of an issued read request to the disk. The disk throughput obtained for reading a 64 KB data block was 934 KB/second. The disk throughput obtained for other data block sizes are normalized to this value. As can be seen, the throughput drops off precipitously as the block size drops to 8 KB or below.

The performance overheads in the various software layers in the filesystem and the kernel are listed in Figure 4-2 for the same data block sizes. As can be seen, the invocation of the disk driver (`ds_invoke`) and its return (`ds_return`) have constant values as is the processing time for an interrupt service routine (`ISR`). As can be expected, the transfer time between the disk and the processor is proportional to the size of the data block transferred. Similarly, the filesystem code which receives the data spends time proportional to the block size read.

4.3. Experiment #1

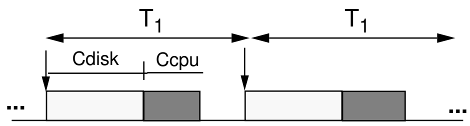
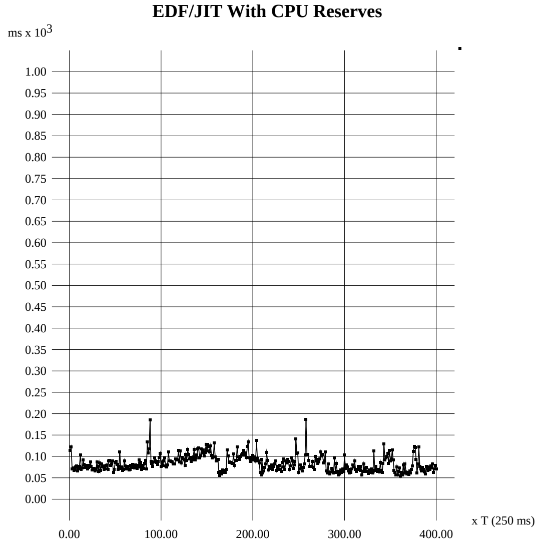


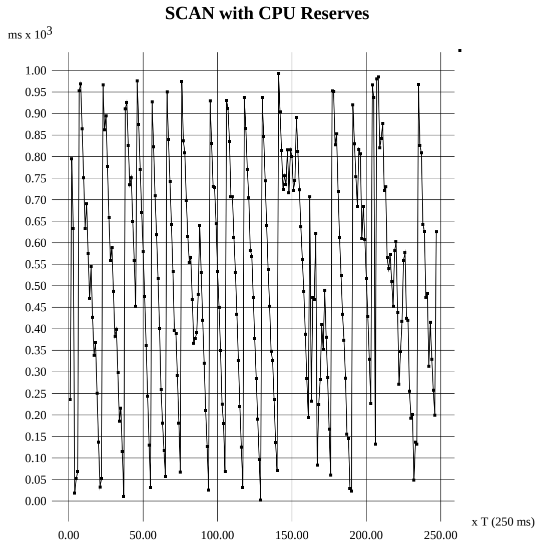
Figure 4-3: Thread execution pattern in Experiment #1

In this experiment, one periodic thread reads from the disk in several configurations: with and without disk bandwidth reserves, with and without cpu reserves, but all in the presence of competition. We ran the experiment in a window time-span of 100 seconds measuring the following parameters: completion time, disk utilization per period, and total disk utilization. The execution of the real-time thread is as shown in Figure 4-3, where T_1 is the thread period (= 250 *ms*), C_{disk} is the time (= 162 *ms*, time to read 48KB) to read disk blocks during each of its periodic intervals and D_{disk} is the deadline for disk access completion (= 162 *ms* also). D_{disk} was deliberately made the same as Cdisk forcing the admission control test to reach 100% of the utilization factor, thereby forcing a stringent test for the real-time filesystem. C_{cpu} (= 44 *ms*) is the maximum allocated execution time each period of the thread that uses data read using the disk bandwidth reserve. Six periodic threads (with no disk bandwidth reserves) having periods 340 *ms*, 380 *ms*, 400 *ms*, 420 *ms*, 450 *ms*, 630 *ms* were competing for disk accesses attempting to read 60KB, 50KB, 200KB, 100KB, 120KB, 150KB during each of their periodic instances. In addition, an unreserved aperiodic thread was in an infinite loop accessing 4KB for each loop iteration. However all the unreserved threads behave exactly the same as soon as they miss a deadline - they execute as soon as the disk is available as the period has been overrun.

We measured the completion times of the reserved thread



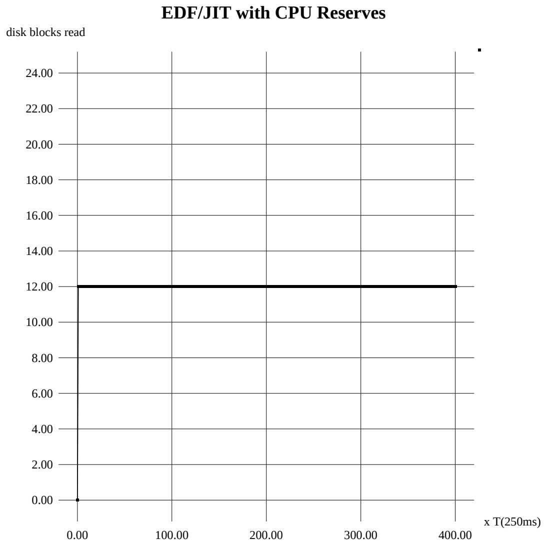
(a)



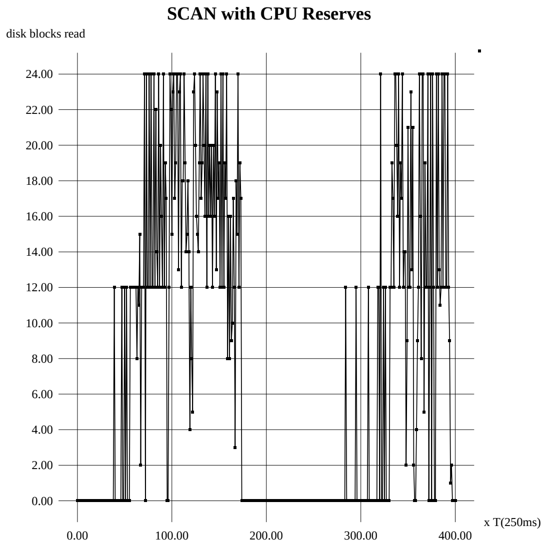
(b)

Figure 4-4: Experiment #1: Completion Times

each period (it should be smaller than D_{disk}). Our measurements were conducted with and without CPU reserves for the CPU portion of the real-time thread. The results in either case are qualitatively similar with the CPU reservation case being moderately better. This is because the competing threads had very limited CPU processing needs. Due to paper length limitations, we plot in Figure 4-4 only the results when a CPU reserve is used for the real-time thread. When we use EDF/EDF+JIT, the deadline of 162 *ms* is missed two times out of 400: the 88^{th} and 258^{th} instances complete in about 185 *ms*, which is still less than the period boundary of 250 *ms*. These misses happen because of two reasons. First, as mentioned in Section 2.5, this is because the file being read was not contiguously laid out on the disk, violating an assumption of the admission control policy. We verified this by reading a file as contiguous blocks in which case, *all* deadlines are indeed satisfied. Secondly, files which are longer than 48KB have



(a)



(b)

Figure 4-5: Expt. #1: Disk Usage Per Period

block maps (meta-data) which must also be fetched from the disk. This is not accounted for in our current admission control scheme.

In the SCAN case, there is no time to run the needed 400 disk accesses and only 248 access are actually completed during the duration of the experiment. The completion times are nearly always greater than the period itself (> 250 *ms*) and sometimes much greater. Note that the y-axis scales are very different between the EDF and Scan graphs. This shows that EDF w/ CPU reserves meets the timeliness constraints of the real-time application accessing the disk.

The number of disk blocks read by the real-time thread during each of its periods under the various algorithms are plotted in Figure 4-5. As can be seen, under the EDF schemes, the number of disk blocks read during each period

Thread and the number ofdisk blocks read	EDF with CPU Reserves	Scan with CPU Reserves	EDF without CPU Reserves	Scan without CPU Reserves
RT-Thread (reserved)	4788	2975	4800	2980
Thread 1	2100	2267	2100	2681
Thread 2	478	770	510	822
Thread 3	2355	4410	2190	4410
Thread 4	2088	2061	2107	2106
Thread 5	1824	1482	1862	1482
Thread 6	1172	600	1169	650
Looping Thread	2340	3428	2743	3428
Total (blocks read)	**17,945**	**17,993**	**17,481**	**18,559**
% of Scan	96.69	96.95	94.19	100
% degradation relative to Scan	**3.31**	**3.05**	**5.81**	**0**

Table 4-1: Expt. #1: Total Disk Usage of each Thread

is exactly 12 (* 4KB/disk block = 48KB), the requested value. In contrast, the Scan algorithm sometimes reads 0 blocks, sometimes 12 and sometimes 24, depending upon the competing stream of requests at any given point in time. However, since it does not and cannot maintain an average of 12 blocks read for the real-time thread, it falls behind for the real-time application.

However, it must be noted that the Scan algorithm was specifically designed to optimize disk throughput. Hence, the question of the relative disk throughput obtained by the Scan and EDF algorithms must be addressed. The disk throughput obtained by each thread under the various schemes is listed in Table 4-1. We obtained the somewhat surprising result that EDF actually performs within 3.5% of the throughput obtained by the Scan algorithm, while satisfying the timing constraints of the application. We shall address the question of whether this throughput difference is always small in Experiment #2.

4.3.1. Experiment #2

Experiment 1 involved synthetic workloads, and therefore its results may or may not correspond to real applications. In addition, that workload assumed a constant load for the real-time task accessing the disk. In practice, a task may also exhibit some stochastic behavior in accessing the disk. We address both of these issues in this experiment. We use a QuickTime movie player and run it within the real-time filesystem context. The player reads blocks from the movie file periodically, processes the data and displays the frames read. The QuickTime decompression algorithm only requires a new frame every 3 or 4 playback periods (it depends on the sequence).

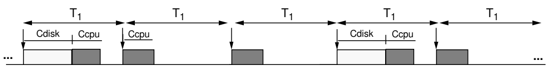


Figure 4-6: Disk accessing pattern of QTPlay_rts

The execution pattern of our QuickTime player application is as shown in Figure 4-6. The video playback thread is a periodic thread that reads from the disk to retrieve the video frames and process them to display the image on the screen. The period T_1 shown is the inverse of the frame rate, C_{disk} is the time to access the disk each period, and C_{cpu} is the execution time need to process a frame each period and display it on the screen.

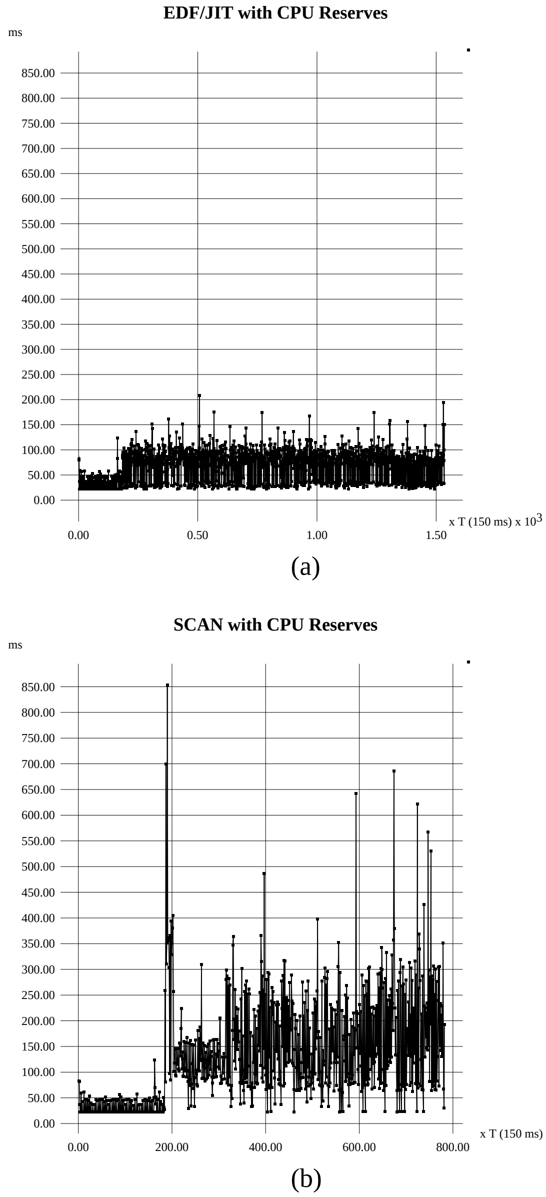
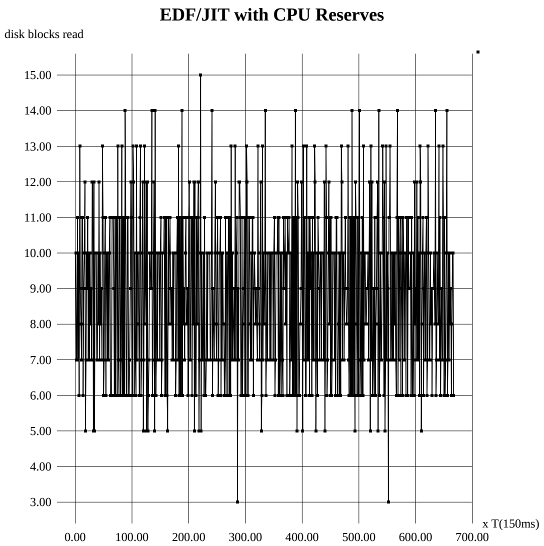
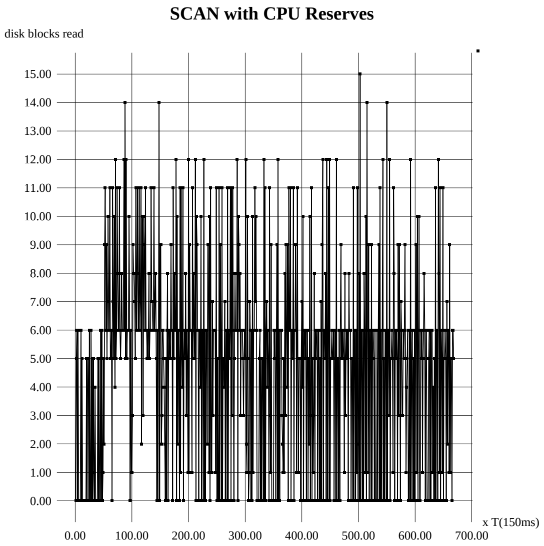


Figure 4-7: Expt. #2: QT-Play Completion Times

We ran the experiment under several configurations: with/out disk bandwidth reserves, with/out cpu reserves and under competition in all cases. We ran the experiment in a window time-span of 100 seconds measuring the following parameters: completion time, disk utilization per period, and total disk utilization. When used, the disk bandwidth reserve was for 40KB every 150 *ms*, with video frame sizes ranging from 19-32 KB. The deadline of the reserve was also 150 *ms*. The unreserved competition consisted of (a) two periodic threads with periods 180 *ms* and 380 *ms* and read 20KB and 100KB respectively during each of their instances, and (b) an aperiodic thread in an infinite loop reading 8KB in every iteration. The video display thread had a period of 50 *ms* (frame rate = 20). As before, all the unreserved threads execute as soon as the disk is available when they miss an end-of-period deadline. The competition is started after 180 periodic instances of video processing leading to an initial quiescent state.



(a)



(b)

Figure 4-8: Expt. #2: QT-Play Disk Utilization per Period

Test	Scan (total disk blocks read)	EDF/JIT (total disk blocks read)	EDF/JIT Performance Penalty (%) (Scan - EDF) / Scan
1	14951	12521	16.25%
2	13468	11029	18.1%
3	13704	9797	28.5%
4	14879	12601	15.31%
5	15115	13827	8.52%

Table 4-2: Total Throughput Comparison Over Multiple QuickTime Files with Different Disk Locations

We measured the completion times of the video thread for each of its instances. These are plotted in Figure 4-7. As can be seen, under EDF/JIT scheduling, all but a very small number of disk accesses meet their deadline of 150 *ms* (for the same previous reason of reading a non-contiguous file). Even those that miss the deadlines miss it by a very small amount. In contrast, under SCAN scheduling, deadlines are consistently missed with some deadlines missed by huge margins (taking up to 850 *ms*).

The # of disk blocks read by the QTPlay thread under both EDF and Scan is plotted in Figure 4-8. It can be immediately seen from the EDF plot that the QuickTime player does not access the disk every time it runs[4]. The average number of disk blocks read is centered around 9, and is generally above 5. It is also sometimes 0 because it may happen in the sequence that no new block is required in three consecutive periods (each period is 50 ms, and the points are plotted in intervals of 150 ms). In contrast, the Scan algorithm has an average of 6 disk blocks read every instance of the Video thread, with but 0 reads during an unacceptably high number of intervals. This implies that a real-time thread under Scan can fall farther and farther behind since it is using available disk throughput to satisfy the disk requests from other competing unreserved (non-real-time) threads.

4.3.2. Sensitivity of File Location to Disk Throughput

For the above experiment, we found that the total disk throughput under the Scan and EDF/JIT algorithms was 15111 and 10280 disk blocks respectively. In other words, EDF/JIT performs 32% worse than the Scan algorithm in this case. However, Experiment #1 had indicated that EDF/JIT can perform relatively close to Scan in terms of filesystem throughput while the QuickTime player showed that it could be higher. In order to understand the sensitivity, we re-ran Experiment #2 with many movie files at different locations on the disk. These runs are summarized in Table 4-2. We found that depending on the layout of the file in the disk relative to the files accessed by the non-real-time applications, the loss of system throughput could range from 8.5% to 32%. It must be noted that under all these conditions, the EDF/JIT algorithm continues to satisfy disk access deadlines.

The normal optimization strategy of improving filesystem throughput is to pack the disk periodically such that (a) the disk blocks corresponding to a file are close to one another and (b) the blocks are also laid out consecutively. Both of these optimizations will lead to narrower throughput differences between EDF/JIT algorithms and traditional algorithms like Scan. However, EDF/JIT will yield predictable and timely disk accesses for real-time applications, while traditional algorithms do not.

5. Concluding Remarks

Disk storage has been traditionally avoided in real-time systems because of their lack of resistance to vibrations, dirt, size, power, etc., but also because of their slow speeds and unpredictable behavior. However, with today's large disks and advances in packaging technology, the use of disks would be very useful. Desktop and other multimedia systems must necessarily use disks to read and store real-time

[4]The QuickTime movie format yields variable data sizes for each frame played back with the variability depending on the actual movie sequence.

audio and video data. In all these cases, for disks to be usable in a real-time context, disk accesses must be completed on a timely basis. Unfortunately, since disk speeds are limited by physical movements of a disk head, disk throughput is normally enhanced by trying to minimize head movements. This is in general anti-thetic to the real-time requirement, where a late arrival may have the earliest deadline and therefore must be serviced immediately independent of its disk block position. In this paper, we have considered real-time scheduling algorithms that can be used in a general context of concurrent but different types of disk accesses. We presented a "just-in-time" disk scheduling algorithm that attempts to meet timing deadlines while trying to keep system throughput. We have designed and implemented a real-time filesystem in RT-Mach using its resource reservation model. Our performance evaluation on real and synthetic workloads show that our real-time algorithms, and their design and implementation in a practical system yield significant benefits. First, the timing constraints of real-time disk accesses *can* be satisfied. Secondly, contrary to what one might expect, disk throughput does not drop significantly compared to traditional algorithms. The traditional Scan algorithm, for example, yields unacceptable latencies for disk accesses, but exhibits better throughput ranging from 3% to 30% depending on file layouts on the disk.

Our future work on this topic will involve many fronts. If non-contiguous files are used, the admission control scheme can be very pessimistic and lead to extremely low values at which disk bandwidth can be guaranteed. Contiguous files are therefore necessary to obtain more acceptable and predictable disk bandwidth. Hence, we are exploring the performance aspects of contiguous files, and the need for disk space reservation in order to create contiguous files. Finally, we are also studying the problem of co-scheduling processor and disk bandwidth access more extensively.

References

1. R. Abbott and H. Garcia-Molina. Scheduling Real-Time Transactions with Disk Resident Data X Server. Tech. Rept. CS-TR-207-89, Department of Computer Science, Princeton University, Feb., 1989.

2. S. J. Daigle and J. K. Strosnider. "Disk Scheduling for Multimedia Data Streams". *Proceedings of the SPIE Conference on High-Speed Networking and Multimedia Networking* (1994).

3. Joseph, M. and Pandya. "Finding Response Times in a Real-Time System". *The Computer Journal (British Computing Society) 29*, 5 (October 1986), 390-395.

4. Lehoczky, J. P., Sha, L., Strosnider, J. K. and Tokuda, H. "Fixed Priority Scheduling Theory for Hard Real-Time Systems". *Technical Report, Department of Statistics, Carnegie Mellon University* (1991).

5. Leung, J. Y., and Whitehead, J. "On the Complexity of Fixed-Priority Scheduling of Periodic, Real-Time Tasks". *Performance Evaluation 2*, 4 (Dec. 1982), 237-250.

6. Liu, C. L. and Layland J. W. "Scheduling Algorithms for Multiprogramming in a Hard Real Time Environment". *JACM 20 (1)* (1973), 46 - 61.

7. S. Chen, J. A. Stankovic, J. F. Kurose, D. Towsley. "Performance Evaluation of Two New Disk Scheduling Algorithms for Real-Time Systems". *The Real-Time Systems Journal 3* (1991), 307-336.

8. P. Lougher and D. Shepherd. "The Design and Implementation of a Continuous Media Storage Server ". *Proceedings of the 3rd International Workshop on Network and Operating System Support for Audio and Video* (November 1992).

9. C. W. Mercer and R. Rajkumar and J. Zelenka. Temporal Protection in Real-Time Operating Systems. Proceedings of the 11th IEEE Workshop on Real-Time Operating Systems and Software, May, 1994, pp. 79-83.

10. C. W. Mercer and S. Savage and H. Tokuda. Processor Capacity Reserves for Multimedia Operating Systems. Proceedings of the IEEE International Conference on Multimedia Computing and Systems, May, 1994.

11. Nakajima, T., Kitayama, T., Arakawa, H. and Tokuda, H. "Integrated Management of Priority Inversion in RT-Mach". *Proceedings of the IEEE Real-Time Systems Symposium* (December 1993).

12. R. Rajkumar, K. Juvva, A. Molano and S. Oikawa. Resource Kernels: A Resource-Centric Approach to Real-Time Systems. To Appear in the Proceedings of the SPIE Conference on Multimedia Computing and Networking, Jan, 1998.

13. Sha, L., Rajkumar, R. and Lehoczky, J. P. "Priority Inheritance Protocols: An Approach to Real-Time Synchronization". *Technical Report (CMU-CS-87-181), Department of Computer Science, CMU* (1987).

14. Sprunt, H.M.B., Sha, L., and Lehoczky, J.P. "Aperiodic Task Scheduling on Hard Real-Time Systems". *The Real-Time Systems Journal* (June 1989).

15. H. Tezuka and T. Nakajima. Simple Continuous Media Storage Server on Real-Time Mach. The USENIX 1996 Annual Technical Conference, San Diego, January, 1996.

16. Tindell, K. An Extendible Approach for Analysing Fixed Priority Hard Real-Time Tasks. Tech. Rept. YCS189, Department of Computer Science, University of York, December, 1992.

Table of Contents

List of Figures

List of Tables