# Multimedia support for MINIX 3

**David van Moolenbroek**

Masters Thesis in Computer Science

September 2007

vrije Universiteit amsterdam

# Table of Contents

# Chapter 1. Introduction

The general issue of multimedia retrieval, storage and transmission has become an important research subject in the past two decades. It continues to be so, as more and more applications make use of multimedia content in one way or another. In this thesis, we look at fulfilling the requirements of multimedia applications in the context of the MINIX 3 microkernel operating system.

## 1.1. Problem definition

So far, there has not been much experience with MINIX 3 as a multimedia platform. We pose the following questions: can MINIX 3 be made suitable for basic multimedia file serving, which changes would this require, and what advantages would be obtained from such an approach? To find answers, this project presents an effort to add multimedia-oriented capabilities to MINIX 3.

In order to keep the project within reasonable bounds, a number of simplifications had to be made. There are three main bottlenecks of multimedia processing: the disk, the CPU and the network. This project only concentrates on the first two: the disk and the CPU. In addition, we fully focus on retrieving multimedia from disk, rather than also storing multimedia on it. That restriction, however, is limited to the file server part of the project. Finally, given MINIX' limited hardware support, we concentrate on commodity systems. As such we do not take into account setups with multiple hard disks.

Thus, this project is mainly concerned with retrieving multimedia from a hard disk, and delivering it to a locally running application. We define three main goals:

1. To let multimedia applications spend as little time on periodic data retrieval as possible.
2. To optimize the multimedia data retrieval from disk to get the most out of the system.
3. To isolate multimedia applications from both each other and nonmultimedia workloads.

Combined, these points enable optimal performance of multimedia applications. The results can then be used in a variety of contexts and with varying external conditions.

In the process, the project allows us to experiment with a number of novel things in the context of MINIX 3. In particular, the new VFS/FS split, user-space threads, and asynchronous message passing.

## 1.2. Our approach

In practice, the first goal translates into a minimal duration of periodic read() calls from applications on multimedia files. We achieve this by prefetching all data before the read() call is made. The prefetching approach also allows for disk retrieval optimizations, and together with optimized data placement on disk, that fulfills the second goal. The third goal is related to the first goal, and implies that other workloads on the system must not affect the duration of the read() calls. It also covers CPU scheduling.

The microkernel architecture of MINIX, and in particular the implementation of its various system servers, pose various challenges in achieving these goals.  Figure 1.1 illustrates the current situation. It shows the chain of processes involved in serving a single read() request for data from the hard disk made by a user process.

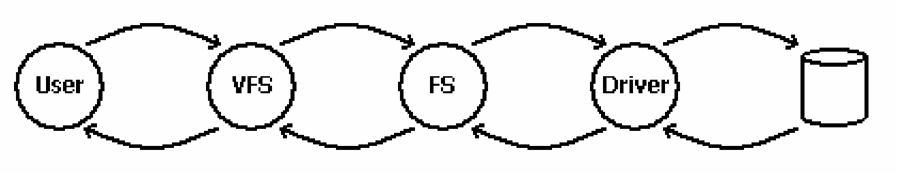


*Figure 1.1: the path of disk retrieval*

First, the user process makes a request to the Virtual File System (VFS) system server. This server acts as the main gateway for all file operations. It relays the request to the actual file system (FS) server, which in turn makes a request to the disk driver. The disk driver then starts retrieval of the data from disk. Once finished, it returns the results to the FS, which in turn reports back to VFS. Finally, VFS replies to the user process. Note that all these processes are reside in user space. The MINIX kernel is only used for message passing between the processes. There is currently only one FS server that is part of MINIX 3: the MINIX File System (MFS) server.

Early tests showed that the actual disk access is by far the dominating factor in the whole path of data retrieval. VFS, MFS and (for the most part) the disk driver are implemented as blocking processes. They can only handle one request at a time, and like the user process, they do nothing while waiting for a reply from the next part in the chain. This leaves VFS, MFS and the driver blocked waiting for the disk to fulfill requests, and by extension all processes concurrently making VFS requests.

The blocking nature of VFS prohibits development of a FS server that can deal with multiple requests at the same time. For example, a file server cannot effectively delay replying to a single read request until data becomes available, without also blocking VFS. Moreover, a nonmultimedia process that performs many heavy disk operations will block all multimedia processes immediately at the VFS level. This it makes it impossible to fulfill our first and third goal.

The first part of this project therefore consists of changing VFS to allow multiple outstanding user process requests to various FS servers. In other words, VFS is “unblocked”. The advantages of unblocking VFS extend beyond multimedia retrieval. For instance, it also provides necessary ground work for networked file systems and nonprivileged file system processes. A significant part of this project has therefore gone into unblocking VFS in a generic way, rather than hacking in support to achieve our goals. To avoid having to rewrite VFS from the ground up, our approach to the problem involves user-space threads. This allows much of the VFS code to remain untouched.

The separation of VFS and MFS in MINIX is a recent development. Instead of extending MFS, we decided to develop a new file server for multimedia retrieval, along with a new file system format. This allows us to make multimedia-specific disk placement optimizations, and experiment with the new VFS/FS interface. Unlike MFS, the new file server will be able to handle multiple concurrent requests from VFS. It uses guaranteed prefetching to let read() calls take as short as possible. Throughout this thesis we will refer to the new file server as MMFS: the multimedia file server. The original MFS remains (mostly) untouched.

The disk driver requires explicit support for multimedia as well. It is only here that both multimedia and nonmultimedia requests arrive at a single point. We extend MINIX’ generic disk driver framework to cooperate with MMFS instances. Throughout the whole process, various changes have to be made to the relevant message protocols.

Finally, we implement a periodic-task aware CPU scheduler on top of the standard MINIX CPU scheduler. The MINIX scheduler already works well for interactive and CPU-intensive processes; our extension adds support for the different demands of periodic multimedia processes. Its goal is to prevent deadline misses caused by other heavy CPU loads, thereby allowing multimedia processes to run more or less in isolation.

The rest of this thesis is laid out as follows. Chapter 2 describes the background of relevant previous research concerning multimedia. Chapter 3 documents the changes made to VFS and the VFS/FS protocol, and chapter 4 describes the implementation of the VFS and file system part of the new MMFS server. Chapter 5 then documents the extensions made to the FS/driver protocol, and the changes made to the driver. The implementation of the multimedia streams part of the MMFS server follows in chapter 6. Chapter 7 describes the CPU scheduling extension. Chapter 8 documents our testing efforts and outcomes. Chapter 9 concludes this thesis and lists possible future work.

# Chapter 2. Background

Multimedia applications have fundamentally different requirements with respect to system resource usage than nonmultimedia applications. Operating system support is necessary in order to fulfill the requirements, especially if the resources are shared by nonmultimedia applications. Section 2.1 gives an overview of various aspects that play a role in multimedia retrieval from an operating system point of view. In section 2.2, we briefly discuss the MINIX 3 operating system.

## 2.1. Multimedia

Typical multimedia application areas include local video serving, where a video stream is retrieved and displayed on the local system, and video-on-demand servers, where video streams are sent over the network to remote clients, as well as video editing, security video surveillance, digital libraries and distance learning [8]. Two main characteristics are shared by all multimedia applications [23]:

1. Continuous retrieval and storage of data.
2. Large data sizes, both in transfer and storage.

The first characteristic implies multimedia applications expect a more or less constant stream of data over time. This practically translates into *periodic* retrieval, storage and/or processing of data, and results in periodic deadlines before which each piece of work has to be done.

Unlike in realtime systems, however, multimedia deadlines are *soft*: it is not critical that all deadlines are met in every case. For example, in the case of video, missed deadlines would cause brief display stalls. Obviously this does degrade the end user experience and should preferably not occur at all, or at most very irregularly. Multimedia streams should therefore strive to sustain a steady rate with minimal delay and jitter.

Combined with the second characteristic, multimedia can use a significant amount of resources. The following subsections outline the relevant aspects.

### *2.1.1. Disk scheduling*

The main bottleneck in the path of retrieval from and storage onto a hard disk by an application is the hard disk itself. The main factors determining disk speed are disk arm movement (seek) times, rotational delay, and data transfer time [10]. Attempts to minimize the disk latency have led to the development of disk scheduling algorithms that reduce seek times and rotational delay as much as possible. The most well-known and widely used disk scheduling algorithm is SCAN, also known as the elevator algorithm. Given a batch of disk requests, this algorithm reduces disk latency by sorting the requests by their disk position relative to the current arm position. Many slight variations to SCAN have been developed. A simple example is CSCAN, which lets the disk arm move into only one direction per batch rather than two [31].

However, SCAN reorders and processes requests in an arbitrary order from the point of view of multimedia requirements. That makes it unsuitable for use in a multimedia supporting system. In contrast, the Earliest Deadline First (EDF) algorithm [15] orders and execute disk requests based on their associated deadlines. It is guaranteed to serve all requests in time if all deadlines can possibly be met at all. However, EDF orders the requests in arbitrary order from the point of view of disk positioning, resulting in high disk seek times and rotational overhead. It is therefore typically not used alone.

The SCAN-EDF algorithm [24] combines the advantages of both these two algorithms. It uses EDF to order requests by deadline, and applies SCAN to sets of requests that have the same deadlines. The optimization gained from applying the SCAN algorithm depends on how often requests with equal deadlines arrive. To actually force that to happen, all deadlines can be deferred to match a periodic interval. That results in a system of *rounds* or cycles [12,24]. Within each time-bound round, the requests can be SCAN-sorted. Rounds are typically set to take from one second up to several seconds [20].

The downside of deferring deadlines to match the round’s period is that it can take a long time for requests to be served: the SCAN sorting can lead to a maximum of two rounds between the arrival of a request and its result [12]. This is illustrated in figure 2.1. If all stream data is to be available in advance, a stream may have to wait up to two rounds after it is opened. This is the stream’s startup time.

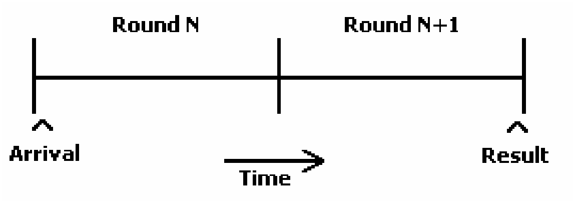


*Figure 2.1: worst case of a request taking two rounds*

The approach of using rounds for disk requests has become widely used [20]. Both round-robin and SCAN ordering of requests within rounds are used in practice. In this respect, an even more finetuned tradeoff between round durations on one hand and disk latency reduction on the other, is found in the Group Sweeping Strategy (GSS) approach [34]. In GSS, rounds are partitioned in groups and requests belonging to a single stream are always put in the same group. Per group, an algorithm like SCAN can be applied. The choice of the number of groups determines the tradeoff between the maximum duration between two subsequent requests, and the benefits resulting from the use of SCAN.

Typically, a system will occasionally have to process nonmultimedia requests as well. Different policies can be used to establish a balance between multimedia and nonmultimedia requests within each round [20,26]. Three desirable performance goals can be defined in this respect [26]:

1. Multimedia requests are all able to meet their deadlines.
2. Nonmultimedia requests are served with small average response times.
3. Starvation of nonmultimedia requests is avoided.

At the very least, nonmultimedia requests should be served whenever there are no multimedia requests. In general, a disk scheduling algorithm is called *work-conserving* when it never goes idle if there are any requests pending at all [29].

### *2.1.2. Disk and file placement*

The ability to optimize on disk latency also depends on the placement of the data on disk. If pieces of data have a high probability of being retrieved consecutively, then the retrieval will be faster as the pieces are located more closely to each other on disk. On one side, arbitrary locations can be used for data blocks belonging to a single file; this is the typical approach of nonmultimedia file systems. On the other side, multimedia files can be laid out contiguously on disk, minimizing the disk latency for the retrieval of consecutive data from a single file [31]. Completely contiguous placement is however rather inefficient for read-write file systems. To provide a more suitable tradeoff between efficient reading and efficient writing, alternatives such as constrained placement [3] and log-structured placement [16] have been developed [23].

Within a single multimedia file, there are various approaches to mapping actual file data to disk blocks, especially if a large disk block size is chosen [5,31]. Internally, multimedia files typically consist of many individual frames. A multimedia file can be stored as any other file by completely filling up individual disk blocks with data, but blocks may also be used to store a number of whole frames. The last approach can prevent single frames from spanning multiple disk block boundaries, and even to let each disk block represent a certain playout time. This comes at the cost of wasting the remaining part of each block.

To exceed the retrieval rate of a single disk, several schemes combine multiple disks. These schemes typically rely on generic multi-disk approaches to speed up multimedia retrieval [10,20]: for example by *striping*, where the disks operate together to turn several physical disk sectors into one larger logical sector, and *interleaving*, where consecutive file data blocks are stored on different disks that operate independently. Many video-on-demand systems have been developed specifically around the use of multiple disks [4,13,28].

Both with a single disk and with multiple disks, a main architectural choice is whether or not to integrate multimedia and nonmultimedia files, on disk as well as in working memory. Both approaches have their own advantages [27]. Examples of integrated systems can be found in [18,28], separated systems can be found in [3,13].

### *2.1.3. Memory buffers*

Memory consumption is another point of concern. Multimedia data retrieval from disk can be viewed as a basic producer/consumer problem. On one side, the disk “produces” data. On the other side, the application process “consumes” this data. The data must be stored in a memory area. As data is necessarily retrieved from disk in blocks, we can say that such a memory area is made up of several block buffers. The minimum retrieval block size is a single disk sector (typically 512 bytes), but retrieving data in larger blocks allows the disk to read consecutive data at an overall higher speed. Combined with the high data rate of multimedia streams, this leads to block sizes of tens to hundreds of kilobytes. Block sizes may be, but are not necessarily, related to the size of individual data frames within the multimedia file.

The simplest model uses a single block buffer for both production and consumption. First the disk fills the buffer, then the application consumes the data, and the process repeats. Given that retrieval from disk is not instant, this inherently lets block retrieval duration determine a part of the application speed. A better approach is a minimum of two buffers per stream [24]: while the application process consumes the data in one buffer, the disk is requested to prefetch data into the other buffer. The buffers’ roles are swapped when both are done. This, however, requires that producer and consumer are synchronized, which may be inconvenient. To let the application make read requests that are not aligned to the buffer size used, three buffers per stream are preferred. This has the additional advantage of allowing further disk retrieval optimizations [24].

The continuous availability of data can then be guaranteed if the production rate at least equals the consumption rate for each stream in every disk retrieval round. This approach is called *work-ahead augmenting* [3]. The explanation above assumes that a static number of blocks is retrieved for each active stream per round. If data rates vary between streams, higher-rate streams can be modeled as multiple lower-rate streams in terms of memory and disk bandwidth requirements. The minimum number of available buffers is then a function of the maximum number of minimum-rate streams, where the minimum rate of a stream is a function of the round duration and the block size used.

### *2.1.4. Prefetching, caching and sharing*

If a static number of buffers is allocated for use by streams, it can be highly desirable to prefetch beyond the two or three buffers per stream if more buffer space is available. This allows streams to deviate more from the actual stream rates.

Buffers that have been consumed but not yet reused make up the buffer cache. The Least Recently Used (LRU) cache replacement is policy is not effective for multimedia [22]. Instead, the linearity of stream progression allows for prediction of when a certain block from one stream is likely to be reused by another stream, and a distance-based caching policy can be derived from this [22]. Blocks that have the largest time or data distance from the current streams' positions, are evicted first. The more general concept of Interval Caching potentially allows the admission of more streams in total, if some streams are guaranteed to have their data available purely from the cache this way [7].

Such sharing may happen by chance, but it can also be forced. For example, in the case of video-on-demand, it is likely that multiple clients will be reading from the same file. If their playback can more or less be synchronized, then a wide range of clients can be supported with a minimal amount of disk access and memory requirements. Several ways to achieve client synchronization have been developed [11]: for example, besides application of the Interval Caching approach (bridging), batches of streams can be forced to start at the same time (batching), or have their rate adjusted to have one catch up with another (piggybacking).

### *2.1.5. Admission control*

Deadlines can only be guaranteed to be met if resource overutilization is avoided. For example, deadlines will be missed if the currently active streams combined require a higher data rate than the hard disk can provide. To avoid that resource overutilization, an *admission control* system determines whether a new stream can be admitted or not. Besides not applying admission control at all, there are two main classes of admission control: deterministic and statistical [6,12].

Deterministic admission control applies a known worst-case upper bound to the use of the resource (e.g., hard disk), and refuses to admit a new stream if the resource maximum is exceeded after adding the new stream's worst-case utilization to the current worst-case utilization. This approach guarantees that all deadlines be met in all cases, but will not use the system resources to the fullest extent if the average utilization is less than the worst-case utilization, which it almost always is.

In contrast, statistical admission control admits streams with a certain probability for the requests to meet their deadlines. No hard guarantees are made any more, but the overal system utilization can be closer to its maximum in the average case. Statistical admission control is typically measurement-based, basing its decision on current system performance rather than the theoretically worst case. Some systems use a combination of deterministic and statistical admission control, by statistically admitting a number of streams on top of deterministically admitted streams [12]. To maintain the deterministic guarantees, the requests belonging to the deterministically admitted streams are always served first.

### *2.1.6. Variable bit rates*

To make the most of available storage space and transmission bandwidth, multimedia files are typically stored and transmitted in compressed form. The applicability of compression depends on the multimedia content, and can result in different compression levels for various parts of the stream. These compressed streams are then constant in their time rate but not in their data rate. Such streams are called variable-bitrate (VBR) multimedia streams, as opposed to constant-bitrate (CBR) streams.

With a time-based continuity over time, the data rate of a VBR stream may fluctuate significantly [17]. This leads to concerns similar to those of admission control, but on a per-stream basis. Admitting a VBR stream with its maximum bitrate is the only way to guarantee that it will always meet its deadlines, but such an approach makes poor use of resources on average. Admitting a VBR stream with its average bitrate would lead to better total resource usage, but may cause missed deadlines during bitrate peaks.

Variable bit rates make the problem of continuous retrieval inherently harder. Many solutions have been proposed in this area. One generic technique is to make up for stream rate irregularities by using buffer space and prefetching, thereby “smoothing” retrieval [2]. More accurate techniques can be developed with specific knowledge of stream parameters, frame boundaries and/or the current client state [1,17].

### *2.1.7. CPU scheduling*

Besides the hard disk, the CPU can also form a bottleneck. The presence of many CPU-intensive processes – multimedia processes as well as nonmultimedia – may lead to a fight for CPU time that can also result in multimedia processes missing their deadline. Once again, multimedia processes do not require high interactivity or constant CPU time, but rather periodic execution with minimal jitter.

Early attempts to accommodate for such processes involved adding realtime priorities to the basic UNIX priority based scheduling. This turned out to work poorly, especially on overloaded systems [21]. Since then, new CPU scheduling systems have been applied to match the periodic nature of multimedia applications more closely. These are based on periodic-task scheduling algorithms such as the Rate Monotonic (RM) algorithm [15] and EDF, either as an extension to priority-based scheduling [19,33] or as a completely new base for scheduling [14]. Admission control is used to avoid allocation beyond the system's maximum.

Such scheduling algorithms are classified as *reservation based*, as opposed to *proportional-share resource allocation* algorithms [30], where each process gets a relative part of the available resources, and allocation therefore depends on the number of other processes present in the system. This approach allows for more flexibility and fairness, but cannot provide guarantees to processes.

## 2.2. MINIX 3

The platform of our choice, MINIX 3, is a microkernel OS designed for reliability, flexibility and security [36]. A detailed description of its architecture and implementation can be found in [32]. MINIX 3 is made up of a small kernel and a number of system processes that reside in user-space. In particular, the kernel only takes care of process scheduling, hardware interrupts, interprocess communication which is mostly implemented as message passing, and a small number of other tasks that can be performed in kernel space only. All other tasks, such as memory management, file management, and all drivers are implemented as user-space system processes.

MINIX is not a realtime operating system. Some efforts have been made to turn it into a realtime operating system though. One example is RT-MINIX [25], where MINIX 2 was extended to include realtime support for resource-constrained platforms. These efforts are concerned with hard guarantees and do not include disk I/O.

# Chapter 3. Unblocking VFS

In this chapter, we describe our efforts to turn VFS from blocking into nonblocking with respect to requests towards file server (FS) instances. We write a new threading library to facilitate this, and change VFS to make use of it. After making a minimal set of changes to the protocol between VFS and FS instances, we apply a model of mutual exclusion to VFS for correct operation.

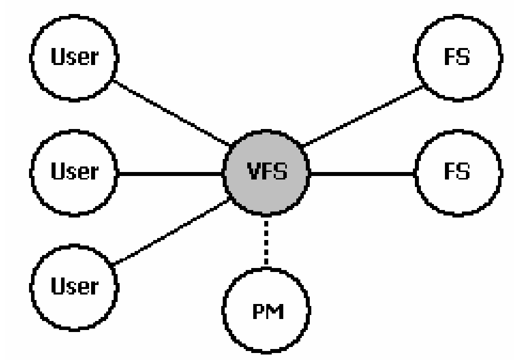


*Figure 3.1: VFS and neighbor processes*

Section 3.1 explains the choice for a (new) threading library, and section 3.2 describes this library. Section 3.3 outlines how the library is applied to make VFS multithreaded. Section 3.4 describes changes in the VFS/FS protocol necessary to allow a nonblocking VFS. Section 3.5 describes the design and implementation of our mutual exclusion model for VFS. Section 3.6 evaluates the results.

## 3.1. Rationale

The original design of VFS involves one main loop that receives a system call message from a user process, processes this message, and sends a reply. The main issue is that the system call operation in VFS could make one or more blocking sendrec() calls to file server (FS) system processes. To let VFS continue with other tasks while it is waiting for the response, the blocking sendrec() call has to be replaced with a new message passing approach. This new approach must allow VFS to continue on other work in the meantime.

VFS will therefore have to suspend the system call operation when it sends a request message to a FS, and resume the operation when the reply message from the FS comes in. To be able to resume the operation later, all state of the operation must be saved somewhere. There are two basic approaches to solve this: continuations and threads.

### *3.1.1. Continuations versus threads*

The continuations approach involves explicitly saving all of the state into a structure associated with the suspended operation. This already happens in various places within VFS, but analysis of the VFS code shows that applying this system to unblock the FS calls would involve a lot of work. There about fifty different locations that make a request call to a FS process, and many more different execution paths that lead up to these calls. With continuations, all of them have to be covered explicitly. Moreover, we need to ensure mutual exclusion of various objects in many calls, and this adds many more places where the current operation may be suspended and resumed later. In addition, there would be a substantial risk that the resulting code becomes completely unreadable, because a new function would have to be called every time an operation is resumed. In any case, the continuations approach would only be feasible if a large part of VFS were completely rewritten from the ground up with all this in mind, and that was not an option for this project.

In contrast, threads can solve this problem by blocking only one thread that is waiting for a reply, letting other threads do other work. On the code level, this would leave most code in place, and only a mutual exclusion model would need to be added to it. Highlevel calls can almost completely hide the thread suspension/resumption and state saving details. This approach would therefore involve much less work, and result in much more readable code. We have therefore chosen this approach.

### *3.1.2. A new threading library*

MINIX 3 does not support kernel threads at this time. It also does not support interruption of running system processes. In user processes, signals can be used for that purpose, but in system processes, signals are sent as messages which have to be received explicitly. For system processes, the only threading system possible is therefore one built around nonpreemptive user-space threads, also known as cooperative threads or coroutines. Fortunately, this is sufficient for our goal.

We decided to develop our own “*systhread*” threading library that exactly suits our needs. Although it may have been possible to find and port an existing nonpreemptive user-space threading library, additional requirements follow from the goal of using the library in a MINIX system server. In particular, a small memory footprint for the actual implementation is desired, and a MINIX-specific stack allocation issue needs to be addressed. Writing a library from scratch instead was deemed (and indeed proved to be) sufficiently simple.

## 3.2 The threading library

This section describes the new “systhread” nonpreemptive user-space threading library. Although it has been developed with VFS in mind, it is generic enough to be used in any other system or even user process. Subsections 3.2.1 and 3.2.2 discuss the creation of thread stacks and threads. Subsections 3.2.3 and 3.2.4 describe the basic usage and API of the library. Subsection 3.2.5 and 3.2.6 discuss practical aspects of the actual use of the threading library.

### *3.2.1. Thread stacks*

One very basic problem is the allocation of virtual stack space for each thread. Although this is a general problem that is hard to do in a portable way [9], a further restriction stems from MINIX’ memory model. In MINIX, each process is allocated a static (process-specified) memory area. This memory area is resident in memory at all times throughout the lifetime of the process, and it is not resizable at runtime. As a result, a process has a “gap” between its data segment and its stack segment, allowing both parts to grow and shrink towards each other (but not collide!) based on the process’s data and stack usage (figure 3.2a). While data space has to be allocated explicitly using a call to the Process Manager system server, the current stack usage of the process is determined based on its Stack Pointer CPU register.

MINIX refuses to allocate memory or call signal handlers if this stack pointer points into the data segment. This is generally a good thing, as it facilitates detection of data/stack segment collision. However, it also implies that allocating and using stack space in the data section stops the process from allocating more memory. System servers must be able to allocate more memory in the data section, and allocating thread stacks on the data segment is therefore impossible, even though this is the most common approach taken by existing thread libraries (figure 3.2b).

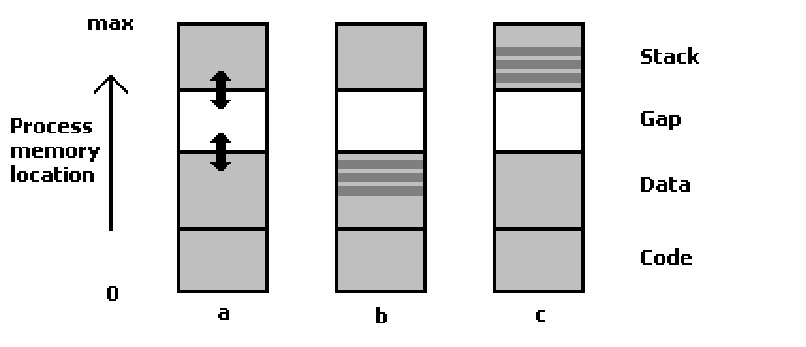


*Figure 3.2: process memory layout*

Our implementation solves the problem by subdividing the process stack into mini-stacks for the threads (figure 3.2c), by artificially advancing the stack pointer to skip a section of the stack during the creation of a thread. This effectively reserves the skipped stack section for use by that thread. In practice, this is a very straightforward process: if a function declares a simple array as local variable, then calling that function will cause the stack pointer to be advanced by the size of the array. We use this to artificially build virtual thread stacks, albeit using the alloca() API call to allocate stacks with a size as specified by the application.

### *3.2.2. Thread preallocation*

Another implication of the MINIX memory model is that there is no point in supporting dynamic thread allocation. The “gap” memory is allocated to the process anyway, and using more stack space does not actually consume more global system memory. To avoid a data/stack segment collision in the worst case, a certain amount of stack space must be reserved for threads. There is then no reason not to preallocate all threads statically.

The allocation process can now take place only once, on startup. The thread library is started with an init() call that has three parameters: the number of threads, the stack size that each of the threads gets, and a pointer to the procedure that gets executed as “main thread”. The threading system initializes a new thread with _setjmp(), then calls alloca() to allocate the stack space reserved for the thread, and recursively creates the next thread from the advanced stack pointer in the same way. When a thread is resumed with _longjmp() from its initialization point, it will have the reserved stack space available to it, without knowing that the stack space is reserved for a function that will never be returned to.

Once the startup routine has recursively allocated the specified number of threads, the application-defined main thread procedure is called. At the moment that this procedure returns, the whole threads system terminates and returns from the original init() call.

### *3.2.3. Thread usage*

Once the thread system has been started, the application can spawn worker threads using the start() call at any time. The first two parameters of this call specify the thread’s procedure entry point and a user-specified parameter. When the thread runs, this procedure will be called with the given parameter. The thread terminates when the procedure returns, after which it resets and becomes free to be reused.

It is possible, and allowed, that the application attempts to start more threads at once than there are threads allocated. In such cases, the thread library uses the third and last parameter to the start() call, a pointer to an opaque *store_t* structure, to queue the request until a thread becomes available. The start() call returns immediately, but the given structure must remain available to the thread system until the thread has actually been started. It is up to the application to determine how to allocate and maintain these structures. The idea is that the application has some way to link concurrently executing threads to existing data structures, so that each of those can have its own *store_t* structure. In return, the application does not have to worry about actual thread allocation.

A thread is runnable if it is in use and not blocked by a synchronization primitive (see the next subsection). The internal scheduler executes runnable threads in round-robin order. Threads can call yield() to let other threads run first. Exactly one thread is allowed to call yield_all() to wait for all other threads to run until they block, typically before it makes a process-blocking call such as the receive() primitive.

### *3.2.4. Synchronization*

The library supports two basic types that can be used for thread synchronization: *mutexes* and *events*.

A mutex (*mutex_t*) provides mutual exclusion between threads. Only one thread can lock a mutex at a time. Other threads that attempt to lock a mutex while it is locked by a thread, will be suspended. When the thread holding the lock unlocks the mutex, the next suspended thread in the queue is resumed. The queue is a simple FIFO list of threads. A thread can lock a mutex using the mutex_lock() call, and unlock it using the mutex_unlock() call. If a thread wants to acquire a lock only when doing so will not block the thread, it can use mutex_trylock(). Mutex objects must be initialized using the mutex_init() call before they can be used.

An event (*event_t*) provides a way for a thread to suspend itself waiting for another thread to wake it up. A thread can suspend itself on an event object using the event_wait() call; another thread can wake it up using the event_fire() call. An event object must first be initialized using event_init(). At this time, the implementation only supports one single waiting thread per event, and requires that event_fire() be called only on event objects that have a thread waiting on them.

The header file that contains the exported structures and functions of the library is provided in appendix A.1. Note that the "systhread_" prefix is used for all public structures and functions to avoid conflict with any naming scheme used in the application using the library.

### *3.2.5. Runtime overhead*

As indicated, the _setjmp() and _longjmp() C library calls are used to actually switch between threads. These are the only high-level intraprocess context switch functions that MINIX offers at this time. The calls save and restore only a minimal set of core CPU registers, and therefore have a negligible impact on overall execution speed.

Being based on FIFO queues, all of the scheduling and synchronization operations are of complexity O(1) in the number of threads. The only exception is yield_all(), which must wait for all other threads to suspend themselves in some way first. The yield_all() operation is the only basic operation for which the implementation contains a loop in its code at all.

### *3.2.6. Picking a stack size*

One problem is choosing the size of the stacks used for the worker threads. This size has to be chosen carefully. On one hand, if a thread exceeds the given stack size, it will overwrite a part of another thread's stack, with disastrous consequences. On the other hand, the minimum memory requirement of the library is a function of not only the number of threads but also this stack size. As such, it is highly desirable to lower the stack size as far as possible, especially for memory-resident servers. However, it is usually far from trivial to estimate the actual maximum stack usage of a certain execution path, and even more difficult to estimate the maximum stack size of all execution paths combined.

The library includes functionality to determine the maximum stack usage empirically. Starting with a relatively large stack size, the library fills each thread's stack with known values before starting the thread. When the thread terminates, the library determines the actual stack usage based on what part of these known values are still present afterwards. The maximum stack usage is then the maximum over all such measured stack usages of all threads. Although not perfect, it provides a good indication when used for a long time. Performing such stack checks is a computationally expensive operation, and it can be turned off at compile time.

The library can be used in nonsystem processes as well. However, if the process uses custom signal handlers, then the threads' stack sizes must be large enough to hold all the kernel-stored stack data and other stack space necessary for invocation of the signal handler. This comes on top of the normal maximum thread stack usage: the kernel uses the current stack pointer as starting point when firing a signal handler, and it cannot be predicted which thread will be active when the signal handler is fired.

## 3.3. VFS threading

With this new threading library, it is now possible to make VFS multithreaded. At this point, we are only concerned with the basic threading model for VFS. Communication changes and mutual exclusion are subject of later sections.

This section defines the new VFS threading model (subsection 3.3.1) and its exceptions (3.3.2), explains how global variables are dealt with (3.3.3), and briefly outlines the resulting implementation changes (3.3.4).

### *3.3.1. Process-based threads*

The new approach for VFS consists of one main thread and a number of working threads. The main thread receives messages and spawns a worker thread for each messsage. The worker thread performs the operation requested in the message, and sends the reply when done. During the operation, each worker thread may send a request to a FS process, suspending itself until the main thread receives a reply for that request and wakes up the thread. It appears to the thread as if it made a blocking call.

The next question is how to fit threads into the VFS model. One important observation is that a user process can never make more than one system call to VFS (or any other system process for that matter) at any time. Thus, each thread can be bound to a process, and we can simply put all data associated with threads in the process table.

This makes it easy to thread all the basic calls that VFS offers to user processes. Also, as only one thread can be active for a single process, by far most operations affecting a single process are automatically serialized. Mutual exclusion is therefore not necessary for most data belonging to a single process.

The flip side of the coin is that there could potentially be hundreds of processes on the system. The per-thread maximum stack usage obtained from actual tests is only about 1 kilobyte (after dealing with a special case described later), but creating one thread per process could lead to unacceptable amounts of memory consumption for the stack space alone. It must therefore be possible to have fewer threads allocated than there are processes. This is not a problem with our threads library: it merely involves adding the *store_t* structure to each process table entry in VFS.

However, having fewer threads than processes also implies that VFS's old concept of suspending a process (for example, because it made a blocking read() call on a pipe) cannot be replaced with new thread synchronization primitives. It takes another process to resume a suspended process (for example, by writing to the pipe). With fewer threads than processes, a sufficient amount of suspended processes could take up all threads and deadlock the system.

Specifically for this project, it should be noted that although we have implemented the support for such a limited number of threads, the multimedia nature of the rest of the project requires that a sufficient number of threads be present to avoid that time-critical multimedia calls are blocked in VFS after all. The number of threads, as well as the stack size, as passed to the thread library, are compile-time constants in VFS.

### *3.3.2. Non user-initiated calls*

The main thread must be able to continue receiving replies and resuming worker threads at all times, so it must never make a blocking call to a FS itself. This means that every piece of code that makes calls to FS processes must be called from a worker thread.

In this regard, we have to make one important exception to the model of associating every system call with a thread linked to the user process that makes the call. The reason is that the Process Manager (PM) system server relays certain duties to VFS. These duties are special PM-only request messages, and although PM is already nonblocking in this regard, the current PM-VFS interface relies on quick processing of at least some of them.

We moved most of the code that handles the PM requests into a dedicated thread that is activated whenever a “there is work pending” notification from PM comes in, and then retrieves and processes the PM requests. This was not sufficient for all cases: the PM_EXEC, PM_EXIT, PM_DUMPCORE, PM_UNPAUSE and PM_REBOOT requests (to perform the VFS part of the exec(), normal exit, coredump exit, signal-caused call abort, and reboot functionality, respectively) can make FS requests on behalf of a certain process. In the case of PM_EXEC, a process can never make a call to VFS while it is blocked making an exec() call to PM. The PM thread then launches a process-associated thread to perform the call *as if* it came from the destination process.

The same does not apply to PM_EXIT and PM_DUMPCORE: it is possible that a process is killed by a signal from another process while it is making a call to VFS itself, so these requests may arrive while a process already has a thread. Handling those calls in parallel is not possible, as this would break the per-process call serialization that we rely on for consistency.

Instead, the PM_EXIT or PM_DUMPCORE request message is stored separately in the process table, and a dummy thread is launched for the process if it it does not have a thread associated with it when the PM request arrives. Upon return from the (VFS-call or dummy) thread, the process table is checked to see if a PM request message is present. If so, it is handled at that point. No special care has to be taken to store new requests originating from the process while the process’s thread is dealing with one of those two PM messages: both PM messages kill the process, so when the thread is done, the process no longer exists.

The PM_UNPAUSE request is stored as a flag in the ‘fproc’ structure. The VFS thread stub function alternatingly executes a thread procedure and processes this flag if it is set, as long as there is work to do. If no thread was spawned for this process yet, a dummy thread is spawned instead. The same approach is taken for device notification messages from privileged driver processes. Since both these message types can be translated to a single flag, it was not deemed useful to let each thread have a generic message queue.

A different approach had to be taken for the PM_REBOOT call, because it iterates over all processes and closes some of their resources (e.g. open files) on their behalf, and only replies to PM once it is done. A mutex was added in each fproc structure to mutually exclude the reboot call from process threads (which also lock their mutex while active). Being mutually exclusive from process threads, the calls made by the PM_REBOOT code can be viewed as made by each process’s thread.

Finally, there are two more cases where functions can make blocking calls without being associated with a specific process: the mounting process upon initialization, and the processing of timer messages for select() timeout code. Both of these were changed to be handled in threads that are asssociated with VFS’s own ‘fproc’ process entry.

### *3.3.3. Global variables*

All state that is necessary for the correct operation of a thread, must remain present across thread suspension and resumption (i.e., across thread-blocking calls). The majority of the thread’s state is present on the thread’s stack, but a part of it is stored as global variables. These global variables have to be dealt with, by instead associating them with threads somehow. We only consider basic global variables that are actually part of the threads’ state at this point. Shared global structures require forms of mutual exclusion, and that is the subject of section 3.5. Although all function-local variables declared ‘static’ are also global variables, no such variables were found to be a danger in any way.

The resulting set of global variables that we are concerned with are listed in table 3.1. These are not the only basic global variables; the remaining global variables are not listed as they do not represent thread-specific state (for example, ‘susp_count’ that counts the number of processes suspended on a pipe).

| **Variable** | **Type** | **Description** |
|---|---|---|
| fp | struct fproc * | Pointer to the process making the current call. |
| m_in | message | The request message as received from the calling process. |
| m_out | message | The reply message being constructed for the caller process. |
| who_p | int | The process table number of the caller process. |
| who_e | int | The process endpoint number of the caller process. |
| call_nr | int | The number of the current call. |
| super_user | int | A flag indicating whether current process has root UID. |
| user_fullpath | char [] | An array to store paths in. |
| cum_path_processed | int | An offset into the user_fullpath buffer. |
| err_code | int | Error number carried across calls. |

*Table 3.1: global variables containing thread-local state*

The ‘user_fullpath’ and ‘cum_path_processed’ are variables used only in path lookups, and were eliminated by saving them on the stack instead; more about this in subsection 3.4.5. The ‘m_out’ and ‘err_code’ variables turned out not to be used across blocking calls or synchronization primitives at all, and could therefore remain as is. This is one of the main advantages of the nonpreemptiveness of the threading model, and we will use this many times more later.

The remaining variables are stored as global variables mostly for fast access, and there is a high level of redundancy between them. In particular, the remaining integer variables can be fully expressed in terms of the two variables ‘fp’ and ‘m_in’, and were replaced with #define directives (table 3.2).

| **Variable** | **Redefined as** |
|---|---|
| who_p | `((int)(fp - fproc))` |
| who_e | `(fp->fp_endpoint)` |
| call_nr | `(m_in.m_type)` |
| super_user | `(fp->fp_effuid == SU_UID ? TRUE : FALSE)` |

*Table 3.2: redefinitions of some global variables*

This leaves us with two essential global variables: ‘fp’, the pointer to the process table entry of the process associated with the current thread/operation, and ‘m_in’, the message containing the user process system call number and data. These two variables are retained across blocking thread calls by placing them onto the stack whenever the thread is about to be suspended on a blocking call, restoring them whenever the thread is resumed.

This was the simplest approach possible. The ‘fp’ process entry pointer needs to be present at all times to allow other thread-specific data to be associated with the thread. Other than being saved and restored as global variable, it could be passed as variable between function calls, but this would add overhead to every single function call and require a large part of the existing functions and function calls to be changed.

Also, the ‘m_in’ message structure could be saved in the process table instead of on the stack. However, this would mean that every access to any of the fields of ‘m_in’ would require a pointer dereference, and there are many of those. Instead, we further optimized the current approach by letting all mutex lock operations use the thread library’s mutex_trylock() call first. If this is successful, ‘fp’ and ‘m_in’ need not be saved and restored at all.

Some further changes were necessary for this approach. In particular, the do_fslogin(), free_proc() and unpause() functions altered the current ‘fp’ pointer on the fly. These functions were changed to perform their tasks without (permanently) changing the ‘fp’ variable. Using #define’s to redefine variables also means that shadowing variables is not possible any more, and some functions had to be changed to use differently named local variables and parameters in order to avoid conflicts with the ‘#define’s.

### *3.3.4. Implementation*

Most of the basic thread functionality is confined to a new ‘thread.c’ module. This includes starting and terminating of threads, and saving and restoring of the global ‘fp’ and ‘m_in’ variables.

Some extra process-associated state is required to ensure that at most one thread per process is active. A substantial part of the ‘main.c’ code was rewritten to spawn threads for every operation rather than to perform the operation immediately. Other changes include initialization of the threads library from the main() entry point, and the insertion of a call to the yield_all() threading library API call before the call to receive(), in order to never block the whole VFS process while there are still threads that are runnable.

To facilitate the creation and maintenance of threads along with all their thread state, the fields listed in table 3.3 were added to the ‘struct fproc’ structure. The use of some of them will become apparent in section 3.5 when we discuss mutual exclusion.

| **Field** | **Description** |
|---|---|
| store_t fp_store; | Place to store data for a pending but not yet executing process thread in. |
| message fp_msg; | Call request message from the user process, also stored in ‘m_in’ while the process thread is running. |
| void (*fp_proc)(void); | Pointer to the procedure that is to process the ‘fp_msg’ message. |
| message fp_msg_pm; | Call request message from PM (see subsection 3.3.2). |
| void (*fp_proc_pm)(void); | Pointer to the procedure that is to process the ‘fp_msg_pm’ message. |
| message fp_msg_fs; | Request message to, or reply message from, a file system (FS) process. |
| endpoint_t fp_fs_e; | Endpoint of the FS process to send the request to and receive a reply from. |
| int fp_thread; | Flag indicating whether a thread has already been started for this process. |
| int fp_unblock; | Flag indicating whether the process should be unblocked after the current call. |
| int fp_notified; | Flag indicating that the (privileged) process has sent a notification. |
| struct fproc | Pointer to the next process in the FIFO queue that this process is in. |

| | |
|---|---|
| *fp_q_next; | |
| int fp_q_type; | The optional requested lock type for queues as part of a lock (see subsection 3.5.3). |
| event_t fp_event; | Generic event used to wake up the thread after it has been suspended waiting in a queue, for a lock or a message. |
| mutex_t fp_mutex; | Process mutex, acquired by the process thread and the PM_REBOOT PM call. |

*Table 3.3: fields added to the 'fproc' structure*

## 3.4. VFS – FS communication changes

With the basic threading model in place, we can move on to the communication between VFS and FS processes. One subgoal here was to keep the protocol changes to a bare minimum. This avoided having to change large parts of not only VFS, but also the MFS file server, which is currently the only FS implementation that is a part of MINIX. The changes made to the protocol would therefore have to allow for multiple concurrent pending requests to new FS servers (in particular, instances of our new MMFS), while allowing only one pending request to each individual MFS process for backwards compatibility.

Subsection 3.4.1 describes how the global behavior of VFS was changed from blocking to nonblocking. Subsection 3.4.2 describes the message passing primitive used for this, and 3.4.3 describes the implications for the message protocol. Subsection 3.4.4 describes how backwards compatibility with MFS is maintained. Subsection 3.4.5 describes the only other protocol change that was needed to allow VFS to be multithreaded. Subsection 3.4.6 describes one more VFS/FS protocol change that was made, although not related to multithreading.

### *3.4.1. From blocking to nonblocking*

Previously, VFS used a simple blocking sendrec() call to send a message to a FS, and block until the FS replies with another message. This call was already wrapped in a fs_sendrec() call to implement a driver recovery mechanism. Only the implementation of this fs_sendrec() call had to be changed to turn the VFS behavior towards FS processes from blocking to nonblocking.

Our new implementation of the fs_sendrec() call executes a mechanism to send the request message to the FS, after which the calling thread suspends itself waiting for an event stored in the thread's associated process table entry ('fp_event'). When the main thread receives the corresponding reply from the FS process, it both stores the reply, and fires the event, in the suspended threads's process table entry. The suspended thread then continues and returns from the fs_sendrec() call with the reply message. All of this is implemented in a new 'fscom.c' FS communication module, along with a somewhat altered implementation of driver recovery. The fscom module will be described further in subsection 3.5.11.

### *3.4.2. Asynchronous send primitive*

The actual sending process was changed to use the new senda() asynchronous send primitive that was introduced in a fairly recent SVN version of MINIX 3. There are two main reasons to use this approach instead of the send() primitive:

- The send() primitive blocks until the message is delivered to the receiving process. This means all of VFS would be blocked sending a message to a FS if that FS process is temporarily busy with something else. With senda(), the message is kept in an array at the sending process until the message is delivered, after which it is marked as delivered. This is done by the kernel, allowing the process to continue with other work while the message is pending.
- With multiple concurrent requests, it can occur with send() that VFS and a FS process attempt to send a message to each other at the same time, leading to a deadlock situation. Although this case is detected by the MINIX kernel, a system that involves retrying the send() after random intervals is hardly ideal. Other, notification-based solutions are possible, but they would always require extra protocol overhead (more messages sent). Senda() avoids this problem by not blocking the sending process while sending the message.

These reasons are sufficient to choose senda() for VFS, but the use of this primitive is not without implications. At least the current senda() kernel implementation does not maintain the order in which multiple messages are sent to a single other process, that is, senda() provides asynchronous, non-FIFO communication. As we will see later, this imposes a fundamentally stricter mutual exclusion model on VFS.

With high rates of messages sent asynchronously by one process to one other process, it could potentially even lead to a single message arriving much later than other messages sent after it. This is not an issue as long as the receiver can receive faster than the sender can send, and we have not observed any such effect to be noticeable in our tests. Improvement of the senda() primitive in this respect could be subject of future work.

VFS' implementation of the senda() call is in 'asyn.c'. One call queues a single message for asynchronous sending to a FS process. Another call notifies the kernel (using the actual senda() call) that new messages are to be sent asynchronously. To minimize overhead, the latter is called only immediately before VFS makes the blocking receive() call in its main loop, rather than right after each individual message is sent. The asyn.c implementation was later reused in MMFS and libdriver as well.

### *3.4.3. Message IDs*

With the combination of multiple concurrent outstanding requests and non-FIFO communication to single FS servers, VFS needs to have some way to link replies from FS servers to the requests sent to them. Without such a mechanism, VFS would not be able to determine which suspended thread to resume and pass a reply to. The solution to this problem is to include a certain value (or combination of values) in the reply that are unique to the request, typically using a unique identifier (ID) value present in request and reply.

Unfortunately it turned out that messages were already "full": several VFS/FS request messages use all the fields in the message. To make things worse, both request and reply messages come in two different internal layout formats (types m2 and m6 [32]). This makes it impossible to establish a location for an ID value in the message structure that is consistent across both the message structure's m2 and m6 unions. Although the basic message structure could be expanded to make room for an ID value, that would make all messages used throughout the whole MINIX operating system bigger, and add more overhead to every single call involving messages.

A better solution resulted from the observation that the (32-bit) m_type field of the messages was not used to the fullest extent in the VFS-FS protocol. In this field, requests carry the request type (a small number), and replies carry the result code (either a zero “OK” value or a negative, small error number). This allows the m_type field to be split in two, using the lower bits of the field to store the original request and error codes, and the higher bits of the field to carry the message ID. The split we used is 16/16 bits, although this could easily be changed later. Addition, retrieval and removal of the message ID are done with three simple macros. The first one returns the ID, the last two return a new m_type value.

```
#define      GET_ID(type)        ((type) & 0xffff)
#define      ADD_ID(type, id)    (((type) << 16) | ((id) & 0xffff))
#define      DEL_ID(type)        ((short)((type) >> 16))
```

VFS then attaches a message ID to each message, which the FS process retrieves, removes and saves. When the FS process replies, it attaches the same message ID to the reply message, allowing VFS to link the reply to the request. The message ID is opaque to the FS process. For maximum simplicity, efficiency and convenience, the ID that VFS actually uses is the process table index of the requesting thread.

### *3.4.4. Maximum  number of concurrent requests*

To retain backwards compatibility with MFS, VFS needs to distinguish between FS servers that are capable of handling multiple concurrent requests and servers that are not. The FS server must therefore be able to specify this property.

To this end, the REQ_READSUPER handshake was slightly modified; specifically, the reply message layout was extended to include a new RES_MAXREQS field. When sending the reply, the FS process fills the RES_MAXREQS field with the maximum number of concurrent requests that it can handle.

To enforce this maximum, the “send” part of the changed fs_sendrec() routine does not actually always send the message to the FS right away. Instead, it keeps a per-FS counter of outstanding requests. More on this in subsection 3.5.11.

MFS was changed to send the RES_MAXREQS value ‘1’ to VFS. This ensures that MFS only receives one request at a time, so that MFS does not need to support asynchronous communication at all.

### *3.4.5. The shared path buffer*

Another issue that had to be resolved, was the use of a global shared buffer for path lookups. A path lookup is the process of resolving a file path string to a specific file entry (inode) on a file system. During a path lookup, VFS asks one or more FS servers to resolve a part of the path. It does this in a loop, sending a REQ_LOOKUP request to each FS server that has to resolve the next part.

During the resolution, a FS server may encounter a symbolic link. If so, the lookup may have to continue on the contents of the symbolic link. The FS server then has to inform VFS where to continue next. It copies the unresolved part of the symbolic link to a buffer in VFS' address space, and sends back a special ‘ESYMLINK’ error code. That buffer in VFS is the ‘user_fullpath’ variable mentioned in subsection 3.3.3. There is only one such buffer; it is shared amoung all FS servers. Its address is specified by VFS to each FS process once, in the initial REQ_READSUPER handshake request.

As long as only one lookup takes place at a time, this approach works. With multiple concurrent lookups, however, the integrity of the buffer contents cannot be guaranteed any more. To let multiple concurrent path lookups take place, this global buffer had to be eliminated.

Each REQ_LOOKUP request already contains an address for another buffer: the buffer containing the part of the path string that is still to be resolved. This buffer is on the stack of the requesting thread, and therefore safe from all other concurrent lookups. The REQ_LOOKUP protocol was changed so that FS processes now copy the contents of a symbolic link to that buffer on the stack, instead of the global buffer.

The REQ_LOOKUP request message layout had to be changed for this. Previously, VFS passed the unresolved part of the path string in the request as a <*pointer into stack buffer, remaining length*> pair. The pointer could be far into the buffer, leaving too little room for a FS process to store the contents of the symbolic link in. The <*pointer into stack buffer, remaining length*> fields were replaced with <*stack buffer base, offset into buffer, remaining length*> fields in the request. For the offset, the unused short integer field 'm6_s3' was used.

The global ‘user_fullpath’ buffer could now be eliminated, and the REQ_READSUPER protocol message was changed not to include its address any more.

### *3.4.6. Ability to query file systems*

One more protocol change was made, although not related to threading. As will be described later, our approach requires that user processes can specify a stream rate to MMFS. This requires a direct communication from a user process to a FS server. We therefore introduced a new call, “QUERYFS”.

Instead of adding a new request type to the VFS/FS protocol, we extended the current implementation of fstatfs(). This system call lets a user process retrieve information about the file system that an open file resides on. The system call specifies a file descriptor and a buffer address. When processing the call, VFS passes the inode number of the open file and the given buffer to the corresponding FS process. The FS process fills the buffer with the corresponding information.

The function and request were extended to allow the user process specify a subcall type and the size of the buffer. One subcall type is for FSTATFS, turning fstatfs() into a subcall of the new queryfs() call. Another subcall type will be introduced for MMFS later. With the explicit specification of the buffer size, every subcall can have its own arbitrary buffer layout.

The header file of the new queryfs() API is included in appendix A.2.

This concludes the changes made to the VFS-FS protocol – the remaining changes now concern VFS only. In particular, MFS did not have to be altered beyond the small set of changes outlined so far.

## 3.5. VFS locking

With all basic issues related to threading VFS out of the way, the most complicated and important issue remains: that of mutual exclusion. It is clear that at least some form of mutual exclusion is required: the request calls to file systems that previously blocked VFS, now only block the current (process-bound) thread, and these requests are by no means atomic or independent from everything else that happens in the VFS process. This section addresses the complete issue.

Subsection 3.5.1 defines a set of requirements for the mutual exclusion model, both to correct operation and for the minimal blocking required by our MMFS. Subsection 3.5.2 discusses which VFS objects (i.e., structure elements) require mutual exclusion. The most important objects are vmnts, vnodes and filps. Subsections 3.5.3 and 3.5.4 define basic rules and systems used for these three objects, and subsections 3.5.5 to 3.5.7 discuss the mutual exclusion systems for each of them in detail. Subsections 3.5.8 to 3.5.10 look at the other objects. Subsections 3.5.11 to 3.5.13 discuss remaining serialization issues.

Appendix B lists the resulting locking requirements of all calls to VFS, and the locks that are active during VFS' requests to FS processes.

### *3.5.1. Requirements*

The locking model that we apply to VFS must guarantee correct functioning even with several threads executing semi-concurrently and in any arbitrary order. The latter point is imposed by the fact that requests and replies use non-FIFO communication, and by the fact that FS instances may take an arbitrary time to process the requests.

This leads to the following five requirements. The first two are essentially a result of the fact that the state of the FS processes cannot be accessed and modified directly by VFS. The last three are mostly the result of use of mutual exclusion to solve the first two.

- *Consistency of replicated values*. Various VFS system call operations rely on having the latest data values on the object they are dealing with. These values may be a replica or representation of state in a FS process, and updating them involves a request to a FS process. To make sure VFS always uses the correct value present on the FS process, mutual exclusion must be used. For example, it is important that VFS knows the correct file size of open files at all times. If not, read calls may operate on an old file size, and for example fail to read the whole file. Worse, write calls that append to the file may end up overwriting earlier write calls. Worse yet, in the case of pipes, this may even lead to deadlocking of processes. To prevent this, system calls that update the file size (i.e., writes) must be mutually exclusive from system calls that use it (reads and other writes).
- *Isolation of system calls*. Even in the absence of replicated state, many system call operations involve more than one request to a FS process. Concurrent requests from other processes must not cause an otherwise impossible outcome of the system call. It is therefore important to identify such cases and provide a sufficient level of mutual exclusion for them. We make one necessary relaxation which results in an isolation level which is not entirely “serializable”. That is, in rare cases, the outcome of one of two concurrently made system calls may be different than in the original blocking VFS. We will discuss this later.
- *Integrity of objects*. From the point of view of threads, acquiring mutual exclusion is a potentially blocking operation as well. The integrity of any objects used across such a blocking call must be guaranteed, and this may require extra mutual exclusion. For example: even though file positions into open files are maintained only in VFS, the ‘filp’ objects that keep such file positions are used across blocking calls. To keep the file position correct even with multiple threads concurrently accessing the same object, mutual exclusion is needed for ‘filp’ objects. Such additional protection can easily lead to an avalanche of recursively required locking of objects, and we will heavily rely on the nonpreemptiveness of the threading model to prevent this where possible.
- *No deadlock*. No two or more concurrent calls may cause all of them not to complete. Deadlock situations are typically the result of two threads that each hold exclusive access to one resource and that want exclusive access to the resource held by the other thread. The resources are data objects in VFS’ case. Conflicts between locking of different types of objects can be avoided by keeping a *locking order* of object types: objects of different types must always be locked in the same order, so that the conflict scenario cannot occur.

If multiple objects of the same type are to be locked, then first a “common denominator” must be locked which is higher up in the locking order.

- *No lockout (starvation)*. Even in the absence of deadlocks, VFS must guarantee that every system call can complete within finite time. For example, where reads and writes conflict, an infinite stream of multiple read calls must not be able to block a single write call forever. The FIFO-based scheduling and synchronization of our threading library simplifies prevention of lockout. However, we will show later that strict use of FIFO everywhere (e.g., “a pending write would always block new reads”) could lead to deadlock situations, so we have to be careful to guarantee both requirements at the same time. Lockout is a much more important issue for VFS than it is for asynchronous sending (see the note in subsection 3.4.2), as blocking calls in VFS can potentially take much longer and do not depend solely on communication speed.

All this has to be combined with the requirements that we have for the MMFS multimedia file server. In particular, the read() calls to a multimedia file must be able to complete virtually instantly, regardless of any other activity on both that file system and on other file systems. Only then, the read calls are independent from other process activity. This translates into the following points:

- A request to one FS process must not block access to other FS processes. This means that most forms of locking cannot take place at a global level, and must at most take place on the file system level.
- An operation that does not affect a certain file on disk, must not block a read() call to that file. This means that not all locking can take place on the file system level, and may have to take place on the file level.
- No read-only operation on a regular file must block an independent read call to that file. In particular, (read-only) open and close operations may not block such reads, and multiple independent reads on the same file must be able to take place concurrently. In this context, independent reads are reads that do not share a file position between their file descriptors.

### *3.5.2. Objects*

We now turn to the relevant objects in VFS. Table 3.4 lists the dynamic data structures (object types) that may require a form of locking.

| **Structure** | **Object description** | **Discussed in** |
|---|---|---|
| fproc | Process; most notably, this includes the process's file descriptors. | Section 3.3. |
| vmnt | Virtual mount: a currently mounted file system. | Subsection 3.5.5. |
| vnode | Virtual node: an open file. | Subsection 3.5.6. |
| filp | File position into an open file. | Subsection 3.5.7. |
| lock | File region locking state for an open file. | Subsection 3.5.8. |
| select | State for an in-progress select() call. | Subsection 3.5.9. |
| dmap | Mapping from major device number to a device driver. | Subsection 3.5.10. |

*Table 3.4: VFS object types*

The threading model already revolves around mapping threads to processes, and section 3.3 has covered the implications of this for the 'fproc' structure. Of the remaining structures, the vmnt, vnode and filp structures are the most important ones. We will consider these first.

A vmnt object is a mounted file system. Vmnt objects can be created with a mount() call, and destroyed with an umount() call. To prevent unmounting of file systems that are in use, the unmounting call counts the number of in-use vnodes on that file system. A vmnt object always has a device number ('m_dev') and a file system endpoint ('m_fs_e'). Both are unique for that object: a file system always operates on a device (typically a hard disk partition) and this device may never be mounted twice. Also, there always is a single FS process handling the file system.

A vnode object is the VFS representation of an open inode (file entry) on a file system. In this context, "open" means that it has a nonzero reference count on the FS. The object also has a reference count within VFS ('v_ref_count') of processes and filp objects using this vnode. The VFS reference count is nonzero iff the reference count as known by the FS server ('v_fs_count') is nonzero, although the two counts may not always be equal. Vnodes are created as a first process opens or creates the corresponding file on the FS, and destroyed as the last process closes it. A vnode's identity is based on the combination of a FS server endpoint ('v_fs_e') and the inode number on that file system ('v_inode_nr'). Additionally, a vnode contains a number of file properties that are replicated from the FS, for example its file size.

A filp object contains a file position within an open file. After creation, the object is always linked to one vnode throughout its lifetime, although not all vnodes are linked to by filp objects. A file descriptor always links to a single filp. The filp has a reference count (‘filp_count’) which is equal to the number of file descriptors referring to it. A filp object does not have a unique identity and cannot be opened, only created (whenever a file is opened or an anonymous pipe is created) and later duplicated as a result of a dup() or fork() call. As such, it is not necessarily in use by only one process.

### *3.5.3. Locking order and basic integrity rules*

From the requirements we defined, it follows that we need locks for the vmnt, vnode and filp objects. The first two need more than one type of locking granularity. We start by defining an initial locking order:

vmnt > vnode > filp

That is, no thread may lock a vmnt while holding a vnode lock, and no thread may lock a vnode while holding a filp lock. This is the only order possible: creation of a new vnode object requires access to a vmnt, and creation of a new filp object requires access to a vnode.

The next issue is then a good and safe definition of the objects’ identity, state of being in use, and lock. We look at the very narrow definition of the object as a structure that is an element in an array in VFS at this point, and do not yet consider what the object actually represents.

If an object has an identity, then no two objects must be created with the same identity. However, after destruction, an object may still be locked by the party that is destroying it. it may not yet be reused at that point because it is still locked. To prevent all kinds of problems resulting for this, the following rules are defined for vmnt, vnode and and filp objects:

1. An object that has an identity (vmnt/vnode) must have separate fields for identity and for being in use.

2. An object is free for reuse if its fields indicate that it is not in use, *and* its lock is completely cleared.
3. If after acquiring a lock, the object is not in use any more, then the lock is exclusive for that object, regardless of the requested access (lock type).

Retrieving/creating an object based on a given identity, which is a very common operation, can then use the following pseudocode:

```
object *get_object(identity, lock_type, do_not_create)
{
        obj_ptr = find_object(identity);

        if (obj_ptr == NIL_OBJ) {
                if (do_not_create)
                        return NIL_OBJ;

                obj_ptr = get_free_object(identity);
        }

        lock_object(obj_ptr, lock_type);

        return obj_ptr;
}
```

The first call always finds the object if an object was present with that identity, in use or not (rule 1), locked or not. If not found, a free object has to be found (using rule 2) and given the new identity. The next step is to lock the object. If a free object had to be acquired, locking always succeeds immediately (rule 2). The resulting object may or may not be marked as in use afterwards; of course, it is certainly not in use if a free object had to be acquired. However, it is locked exclusively if it is not in use (rule 3), which means the caller can mark it in use and fill in any further fields as desired.

This scheme is simple, and works for both creation and mere retrieval. Moreover, it allows the object to be destroyed in an arbitrary way, as long as destruction takes place while a lock is held on the object. To prevent that any activity taking place during the destruction can interfere with a concurrent creation, it is desirable (but not strictly required) to use mutually exclusive lock types for creation and destruction.

This is the basic scheme we use for vmnt and vnode objects. Filp objects have no identity, but rules 2 and 3 are used for finding and locking free entries there as well. The nonpreemptiveness of the threading library is used to avoid having to acquire (for example) a global lock for each object type first.

Table 3.5 lists the definitions of the properties of the three mentioned object types that were chosen. They are mostly derived from the old situation, but required several changes to the code to actually hold everywhere.

| **Object** | **Property** | **Choice** |
|---|---|---|
| vmnt | Identity | m_dev |
| | In use iff | m_fs_e != NONE |
| | Lock | Three-level lock |
| vnode | Identity | v_fs_e + v_inode_nr |
| | In use iff | v_ref_count > 0 |
| | Lock | Three-level lock |
| filp | Identity | - |
| | In use iff | filp_count > 0 |
| | Lock | Systhread mutex |

*Table 3.5: properties for main objects*

### *3.5.4. The three-level lock*

From the requirements discussion of subsection 3.5.1, it is clear that for vmnts and vnodes, at least two locking types are required. Simply put, *read* and *write*. Concurrent *reads* are allowed, but *writes* are exclusive both from *reads* and from each other. This is the classic readers/writer lock. It turns out that it is more convenient (and sufficient) to define *three* locking types, for both vmnt and vnode objects. We developed a simple generic three-level lock datatype for VFS that supports maximum concurrent access, falls back on exclusive access for not-in-use objects (as per the last subsection), and guarantees *no lockout*. Their actual application will be described in the following subsections.

The three locking types are *concurrent* (TLL_CONCUR), *serialized* (TLL_SERIAL) and *exclusive* (TLL_EXCL). The *concurrent* type, similar to the “read” primitive above, allows an unlimited number of other threads to hold the lock with the same *concurrent* type (N * *concurrent*). The *serialized* type similarly allows an arbitrary number of *concurrent* accesses at the same time, but only one thread can get *serialized* access to the lock at a time (1 * *serialized* + N * *concurrent*). The *exclusive* type provides full mutual exclusion, similar to the mentioned "write" primitive (1 * *exclusive*). Figure 3.3 illustrates this. In figure 3.3a, horizontal lines represent threads having access to the lock. Figure 3.3b shows the same example, adding gray areas to indicate mutual exclusion from new access requests for each of the locking types.

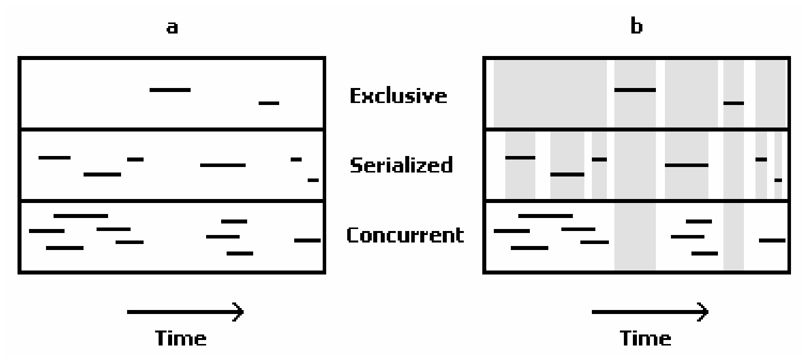


*Figure 3.3: example lock access with different locking types*

If there are no *concurrent* access requests, the *serialized* type is essentially equal to the *exclusive* type, but in the absence of *exclusive* access requests, the *serialized* type never blocks *concurrent* access.

It is possible for a thread to upgrade its *serialized* access to *exclusive* access while holding the lock. In that case, it waits for all threads with *concurrent* access to finish (if any), while any new *concurrent* threads are queued. As we will show, this is necessary for vmnt objects. A thread can also downgrade its access from *exclusive* to *serialized* or *concurrent*, without having to block. However, it is impossible to upgrade from *concurrent* to *serialized*, as that would effectively involve releasing and then reacquiring the lock.

The implementation requires two queues. Access to the lock can be seen as batches of combined *concurrent* and *serialized* access, separated by *exclusive* access (requiring one queue). Each batch allows all *concurrent* access to take place in parallel but grants the *serialized* access one by one (requiring another queue with higher priority). The use of FIFO queues guarantees *no lockout*: an *exclusive* access request/upgrade blocks any subsequent access requests of any type, and a *serialized* access request always blocks any subsequent *serialized* and *exclusive* access requests.

Finally, if the three-level lock module is informed that the object that the lock operates on is not in use (this is a parameter in all calls), it will treat all queued requests as *exclusive* requests for the time being. Maximum parallel access is reallowed once the object becomes in use again. Outstanding concurrent access cannot be revoked once the object stops being in use, but this is not an issue in practice: it cannot happen for either vmnts or vnodes.

Table 3.6 lists the structure field used for the three-level lock structure ('struct tll').

| **Field** | **Description** |
|---|---|
| int t_type; | Current locking level type: TLL_CONCUR, TLL_SERIAL or TLL_EXCL. |
| int t_count; | Number of TLL_CONCUR accesses currently active (0 if TLL_EXCL). |
| struct fproc *t_owner; | The owner of the lock if the level is TLL_SERIAL or TLL_EXCL. |
| int t_upgrading; | If set, the owner is upgrading and wants an event when t_count reaches 0. |
| struct queue t_excl_q; | The main queue of threads that are waiting for the lock. |
| struct queue t_seri_q; | The priority queue of TLL_SERIAL threads that are waiting for the lock. |

*Table 3.6: fields of 'tll' structure*

The queue structure, which maintains a simple FIFO list of 'fproc' structures, requires two fields in the 'fproc' structure of each queued thread. One field is needed to store the requested lock type ('fp_q_type'), and another is needed to store a pointer to the next process in the queue ('fp_q_next'). This is sufficient, as a thread can never be in more than one queue at any time.

### *3.5.5. Vmnt (file system) locking*

With the proper mechanisms in place, the next step is to decide which FS operations require which level of vmnt locking. Besides fulfilling the list of requirements, it is obviously desirable to let as many FS operations take place in parallel as possible. We therefore start off from the point of locking as little as possible, and with the lowest locking level. Then we apply restrictions to increase the locking to a necessary minimum level.

In this subsection, we first identify and classify the system calls that need vmnt locking. Then we look at the main complicating factor: path lookups. Finally, we assign appropriate locking types to the classified system calls.

Vmnt locking cannot be seen completely separately from vnode locking. The reason is that the unmounting process already fails if there are still in-use vnodes (i.e. vnodes with a positive FS reference count, and therefore a positive local reference count), which means that FS requests only involving in-use vnodes do not have to acquire a vmnt lock. On the other hand, the FS requests that do not involve in-use vnodes *do* have to acquire a vmnt lock.

This leads to a very important assertion: of the system calls that VFS offers to processes, the calls that involve a file descriptor, by definition, operate on an open file and thus on an in-use vnode (following the path fd à filp à vnode). *None* of those operations therefore need a vmnt lock. On the other hand, *all other* operations that make requests to a FS *always* need (at least) a vmnt lock.

We start by classifying all these calls in several distinct groups, including the pseudocode that outlines their operation in the original VFS. See table 3.7. For now, we ignore specific details that must be solved at another level.

| **Group** | **System calls** | **VFS pseudocode** |
|---|---|---|
| File open operations (non-create) | open(), exec(), chdir(), chroot() | `file_fs, file_ino = lookup(input_path, EAT_PATH);` |
| | | `if (!(vp = find_vnode(file_fs, file_ino))) {` |
| | | `        res = fs_open(file_fs, file_ino);` |
| | | `        vp = create_vnode(file_fs, file_ino, res);` |
| | | `}` |
| | | `/* ‘vp’ represents an in-use vnode here */` |
| File create-and-open operations (may open only instead) | open(O_CREATE), creat() | `fs, ino, err, file_name =` |
| | | `        lookup(input_path, EAT_PATH);` |
| | | `if (err == does not exist) {` |
| | | `        /* the returned fs, ino are the last dir */` |
| | | `        fs, ino, res = fs_create(fs, ino, file_name);` |
| | | `        vp = create_vnode(fs, ino, res);` |
| | | `}` |
| | | `else /* like open() above, using ‘fs’ and ‘ino’ */;` |
| File create-unique-and-open operations | pipe() | `file_fs, file_ino, res = fs_create_unique();` |
| | | `vp = create_vnode(file_fs, file_ino, res);` |
| File create-only | mkdir(), | `dir_fs, dir_ino, file_name =` |
| | | `        lookup(input_path, LAST_DIR);` |

| | | |
|---|---|---|
| operations | mknod(), slink() | **fs_call**(dir_fs, dir_ino, file_name); |
| File information retrieval or modification (not replicated) | stat(), lstat(), access(), rdlink(), utime() | file_fs, file_ino = **lookup**(input_path, EAT_PATH); **fs_call**(file_fs, file_ino); |
| File modification (possibly replicated) | chmod(), chown(), trunc() | file_fs, file_ino = **lookup**(input_path, EAT_PATH); res = **fs_call**(file_fs, file_ino); if ((vp = find_vnode(file_fs, file_ino))) modify_vnode(vp, res); |
| File link operations | link() | src_fs, src_ino = **lookup**(src_input_path, EAT_PATH); dst_dir_fs, dst_dir_ino, dst_file_name = **lookup**(dst_input_path, LAST_DIR); */* make sure that src_fs equals dst_dir_fs */* **fs_call**(src_fs, src_ino, dst_dir_ino, dst_file_name); |
| File unlink operations | unlink(), rmdir() | file_fs, file_ino = **lookup**(input_path, EAT_PATH); if ((vp = find_vnode(file_fs, file_ino))) */* perform certain checks on the vnode */*; dir_fs, dir_ino, file_name = **lookup**(input_path, LAST_DIR); **fs_call**(dir_fs, dir_ino, file_name); |
| File rename operations | rename() | old_dir_fs, old_dir_ino, old_file_name = **lookup**(old_input_path, LAST_DIR); new_file_fs, new_file_ino = **lookup**(new_input_path, EAT_PATH); if (*new file already exists* && (vp = find_vnode(new_file_fs, new_file_ino))) */* perform certain checks on the vnode */*; new_dir_fs, new_dir_ino, new_file_name = **lookup**(new_input_path, LAST_DIR); */* make sure old_dir_fs == new_dir_fs */* **fs_call**(old_dir_fs, old_dir_ino, old_file_name, new_dir_ino, new_file_name); |
| Non-file operations | sync(), stime() | **fs_call**(); |

*Table 3.7: classification of non file descriptor based process calls in VFS*

The calls that actually involve FS requests are marked in bold. For clarity's sake, the above pseudocode uses different names for most calls than VFS does. For example, 'fs_open' is actually called 'req_getnode' in VFS, and 'fs_create_unique' is actually 'req_pipe'.

The first thing to notice is that there is typically a "gap" between a path lookup and the use of the results of the lookup. During this gap, another system call may be issued that operates on the same path, and may perform an arbitrary number of operations in between. A prime example scenario: between the FS processing the ‘lookup’ and the subsequent ‘fs_open’ calls of a file open operation, the inode resulting from the lookup may be deleted and its inode number may be reused for another file created immediately after. The wrong file would then be opened. Only VFS can prevent this situation from occurring, for example by making entire file delete operations mutually exclusive from file open operations.

A second observation is that file system requests operate on one file system only. In fact most calls that VFS offers to user processes involve only one FS process at a time. An important exception is that in the case of the file link/unlink/rename operations, there are multiple FS lookups leading to a single FS call. During these operations, the results of the first lookup *must* remain valid until the last FS call finishes. Preserving the isolation of these operations while also preventing deadlock proves to be difficult, and that is the direct result of these multiple lookups. Lookups are an important aspect of most of the groups, and deserve special attention.

Path lookups take arbitrary (absolute or relative) paths as input. They may therefore visit arbitrary file systems in an arbitrary order, start and end at an arbitrary file system, and possibly visit the same file system more than once. Each file system resolves the next piece of the path, so the path itself determines the order in which the file systems are visited. VFS can never tell in advance at which FS the lookup will end. All this has the following crucial implications for the locking model:

- In the lookup process, only one file system must be locked at a time. When going from one FS to the next, the lookup process must unlock the last FS before locking the next. This prevents concurrent lookups deadlocking each other. Serializing all lookups globally would violate the first requirement for MMFS, and effectively disallow a multimedia process from immediately opening a multimedia file in its working directory just because other lookups are going on elsewhere. With networked file systems, the effect of this would be even worse.
- The lookup process must lock each visited file system with a lock type that is equal to, or can be upgraded to, the lock type desired by the caller for the destination file system. After all, it does not know whether the file system it queries is the destination file system. The lookup process may therefore not use a *concurrent* lock type if the desired lock type is *serialized* (see subsection 3.5.4).

- Despite locking, two lookups on the *same* path may result in different results each time. Although the first lookup may lock the destination file system, this lock does not prevent concurrent calls from making changes on other file systems responsible for resolving earlier parts of the path. In particular, after these changes, the second lookup may resolve to another file on the same destination file system. The second of two lookups for *one single* path may therefore *never* visit an unlocked file system.

The last point led us to restructure the file unlink and rename operations. Instead of performing two full lookups for a single pathname (one for the full path to perform certain vnode checks, one for the containing directory for the actual operation), it now uses a newly added "ADVANCE" lookup that continues on (and never leaves) the currently locked file system, starting from the containing directory that was the result of the first lookup. This results in the replacement scheme in table 3.8.

| **Group** | **System calls** | **VFS pseudocode** |
|---|---|---|
| File unlink operations | unlink(), rmdir() | `dir_fs, dir_ino, file_name =` `    lookup(input_path, LAST_DIR);` `file_fs, file_ino = lookup(file_name, ADVANCE,` `                          dir_fs, dir_ino);` `if ((vp = find_vnode(file_fs, file_ino)))` `    /* perform certain checks on the vnode */;` `fs_call(dir_fs, dir_ino, file_name);` |
| File rename operations | rename() | `old_dir_fs, old_dir_ino, old_file_name =` `    lookup(old_input_path, LAST_DIR);` `new_dir_fs, new_dir_ino, new_file_name =` `    lookup(new_input_path, LAST_DIR);` `/* make sure old_dir_fs == new_dir_fs */` `new_file_fs, new_file_ino =` `    lookup(file_name, ADVANCE, new_dir_fs,` `                               new_dir_ino);` `if (new file already exists &&` `    (vp = find_vnode(new_file_fs, new_file_ino)))` `    /* perform certain checks on the vnode */;` `fs_call(old_dir_fs, old_dir_ino, old_file_name,` `    new_dir_ino, new_file_name);` |

*Table 3.8: new pseudocode for rewritten unlink/rename operations*

Slightly getting ahead of ourselves, file locking will need to be of the *exclusive* type for these two groups of calls, to prevent the *open vs delete+recreate* scenario mentioned earlier. The *exclusive* lock will prevent files from being opened on the same FS at all, and that also prevents the scenario that a vnode is opened between the vnode checks done in these operations, and the final FS request of the operation.

The requirement for an *exclusive* lock for in particular rename() has implications for avoiding deadlocks. If a lookup is followed by another independent lookup, then the first lookup must never obtain an *exclusive* lock: two threads doing this concurrently would then deadlock on their second lookups. The first lookup must also not obtain a *concurrent* lock if the second lookup obtains a *serialized* lock which it then later upgrades to *exclusive*: this may similarly deadlock if another thread tries to upgrade its own resulting *serialized* lock to *exclusive* after the first thread's *concurrent* lookup has finished but before the *serialized* lookup has finished.

The only way to end up with a *serialized* or *exclusive* lock after multiple lookups, avoiding deadlocks altogether, is to let the first lookup obtain a *serialized* lock. After that, the remaining lookups are *concurrent*. Once the operation will not make any more lookups, the *serialized* lock can be upgraded to *exclusive*. However, the *concurrent*-type lookups must never be queued behind a *exclusive* lock request (this would still lead to deadlocks), so queued *exclusive* lock requests must not exist at all. All other system calls that require an *exclusive* vmnt lock, also first have to acquire a *serialized* lock, and then upgrade it to *exclusive* afterwards. In the cases where such upgrades are not explicit, we take care of this transparently in the new lock_vmnt() call in our implementation.

It is here that we have to make a relaxation from the "serializable" isolation model, as mentioned in subsection 3.5.1. Operations that involve two different paths (i.e. link and rename) are not necessarily atomic with respect to the two path lookups. Between the lookups, changes may take place that affect the second lookup. This concerns lookups on different paths only, and is allowed by POSIX.

This concludes the considerations with respect to path lookups. What remains is the assignment of a locking type to all groups. As starting point, it is desirable to let open() calls that will not create the file if it does not exist, run fully in parallel. In combination with the requirements already defined for MMFS, this means that two or more processes opening (the same or different) multimedia files in parallel would essentially take just as long as a single process opening a single file. This turns out to be possible. The direct implication is that an exclusive lock is needed for all other calls that potentially conflict with these open() calls on the vmnt level.

Table 3.9 lists the lock types for vmnts, based on the locking levels of the three-level lock.

| **Lock type** | **Mapped to** | **Used for** |
|---|---|---|
| VMNT_READ | TLL_CONCUR | Read-only operations and fully independent write operations. |
| VMNT_WRITE | TLL_SERIAL | Independent create and modify operations. |
| VMNT_EXCL | TLL_EXCL | Delete and dependent write operations. |

*Table 3.9: vmnt lock types*

Note that the "write" lock in this context differs from the "write" used in the definition of a readers/writer lock, because single simple write operations are allowed to take place in parallel with multiple read operations.

Table 3.10 summarizes the various lock types that we have applied to all operations in each of the groups, along with the main motivation why this lock type was chosen. Replication of state concerns vnodes, and will be discussed in the next subsection.

| **Group** | **Lock type** | **Motivation** |
|---|---|---|
| File open operations (non-create) | VMNT_READ | These operations do not interfere with each other, as vnodes can be opened concurrently, and open operations do not affect replicated state. |
| File create-and-open operations (may open only instead) | VMNT_EXCL for create; VMNT_WRITE for open. | As will be shown in the next subsection, file create operations require mutual exclusion from concurrent file open operations. If the file already existed, the VMNT_WRITE lock that is necessary for the lookup is not upgraded. |
| File create-unique-and-open operations | VMNT_READ | These create nameless "limbo" inodes which cannot be opened by means of a path. Their creation therefore does not interfere with anything else. |
| File create-only operations | VMNT_WRITE | These operations do not affect any VFS state (and are as such fully atomic), and can therefore take place concurrently with open operations. They must however |

| | | |
|---|---|---|
| | | not take place during the upgrade from the writelock of file create-and-open operations to exclusive, or the create may for example result in an "already exists" error even if the file should have been opened otherwise. |
| File information retrieval or modification (not replicated) | VMNT_READ | These operations do not interfere with each other, do not modify replicated state, and the actual FS operations after lookup are atomic. |
| File modification (possibly replicated) | VMNT_WRITE or VMNT_EXCL | These may conflict with file open operations due to seeing resulting replicated values out of order. An exclusive vmnt lock must be acquired if no vnode can be locked. See the next subsection. |
| File link operations | VMNT_WRITE | Other than the double path lookup, equal to "file create-only". |
| File unlink operations | VMNT_EXCL | These must not interfere with file open operations, in order to avoid the scenario where inode numbers could be reused immediately. A second reason is that these calls perform necessary checks on currently existing vnodes. |
| File rename operations | VMNT_EXCL | Same as for the "file unlink" operations. |
| Non-file operations | VMNT_READ | The update calls that make up this group, particularly sync(), are atomic at the file server itself, and therefore do not need to be protected from any other calls. |

*Table 3.10: vmnt lock types used for non file descriptor based process calls in VFS*

Following the lock type restrictions mentioned above to avoid deadlock, a VMNT_EXCL lock of a file operation always involves a VMNT_WRITE lookup followed by an upgrade. In the case of multiple lookups, the first one is always of type VMNT_WRITE and the subsequent ones are of type VMNT_READ, eventually then followed by an upgrade of the first VMNT_WRITE lock to VMNT_EXCL.

Table 3.10 only lists the main motivation for the lock type as resulting from our initial approach. If a different starting point were chosen, many other considerations could result. A simplifying aspect is that most of the actual FS calls are atomic themselves, which is why relatively few operations can interfere with eachother purely at the vmnt level. Apart from the path lookups, the isolation requirement is therefore fairly easily fulfilled.

Missing from the tables are mount and unmount operations. Although it appears that both involve two vmnt locks (for the parent and the child FS), this is not true. Mounting only involves locking one *existing* file system (the parent). Unmounting only involves locking the file system to be unmounted (the child), as notifying the parent of the unmount process merely involves releasing a vnode. Both are safe if a VMNT_EXCL lock is used on the parent (for mount) or the child (for unmount).

Besides changing the vmnt calls to adhere to the new object scheme described in subsection 3.5.2, we had to add the lock_vmnt(), unlock_vmnt() and upgrade_vmnt_lock() calls to the vmnt management. There are only twelve calls to lock_vmnt() in the resulting VFS code, because locking usually happens in the centralized path lookup function. There are eight upgrade_vmnt_lock() and 91 unlock_vmnt() calls, the latter mostly due to multiple exit paths within single calls as a result of error handling.

### *3.5.6. Vnode (open file) locking*

Vnodes, or VFS's representation of open files, are the next objects in VFS. In this subsection, we look at locking requirements with respect to field replication and creation/destruction of vnodes. We then define locking rules, and discuss opening, creation, and modification in more detail.

Vnodes are complicated from a threading point of view. Not only do our MMFS requirements impose a very flexible locking model on them (stating that read() calls must not be blocked by other read-only operations on that vnode), but vnodes replicate a whole number of file properties from the corresponding file system inode as well. As mentioned, these replicated fields must remain "in sync" with the file system at all times. The replicated fields are listed in table 3.11. The first five make up the so-called "node details" of a vnode.

| **Field** | **Description** |
|---|---|
| v_size | The file size. |
| v_mode | The file's access mode. This includes the file type (regular, directory, block special, character special, pipe). |
| v_uid | The user ID that owns the file. |

| | |
|---|---|
| v_gid | The group ID that owns the file. |
| v_dev | The device number, for block and character special files. |
| v_fs_count | The number of on-FS references of this file, that is, the number of times that this inode is opened according to the file server. |

*Table 3.11: vnode fields replicated from FS processes*

However, compared to the vmnt locking, the strategy of vnode locking can be fairly straightforward: all read-only access that merely uses the vnode and its fields is allowed to be *concurrent*, and from this follows that all access that can modify any of the vnode's replicated fields must be *exclusive*. That only leaves the creation and destruction of vnodes, and modification of its reference counts: both the number of times that the file is open on VFS ('v_ref_count'), and the replicated on-FS reference count ('v_fs_count').

As we already mentioned in subsection 3.5.2, the first process that creates or opens a file, creates a vnode object. Subsequent processes that also open the file, merely increase the vnode's reference count. When processes close the file, the reference count is decreased. When the reference count hits zero, the vnode object is destroyed.

Creation and destruction of vnode objects cannot interfere with other access, because of one important fact: one process can never have its files closed by another process. Or, restated from the point of view of individual processes, any filp or vnode that has been opened by a process will remain open (i.e., in use) *at least* until that same process closes it. This means that once a process has opened a file, its VFS threads never have to perform checks later to make sure that the corresponding vnode object still exists.

To stay close to the original "the vnode reference count is nonzero iff the on-FS inode reference count is nonzero" assertion from subsection 3.5.2, we serialize all vnode open/close operations (and thereby creation and destruction as well). This makes the reference changes atomic, so that they never interfere with each other. In particular, this means that the scheme from subsection 3.5.3 can be used. No special attention needs to be paid to the order of increasing or decreasing the reference counts, before or after making an actual FS request that changes the on-FS reference count.

With the three-level lock type, we can use the *serialized* locking type to serialize all such actions. That allows *concurrent* (i.e., read-only) actions to take place in parallel. This is safe because as mentioned, a thread that performs a read-only action on a vnode, can be assured the vnode will have, and keep, a nonzero reference count during the operation. The complete independence is also necessary, because our MMFS requires that read-only open and close operations on files on disk do not interfere with basic read operations on those files.

As a result, we have used the three locking types listed in table 3.12 to implement vnode locking in VFS.

| **Lock type** | **Mapped to** | **Used for** |
|---|---|---|
| VNODE_READ | TLL_CONCUR | Read access to previously opened vnodes. |
| VNODE_OPCL | TLL_SERIAL | Creation, opening, closing and destruction of vnodes. |
| VNODE_WRITE | TLL_EXCL | Write access to previously opened vnodes . |

*Table 3.12: vnode lock types*

This time, “reads” and “writes” *are* indeed mutually exclusive. Write operations consist of all operations that can modify the replicated fields listed in table 3.11 (except ‘v_fs_count’). For example, this includes the basic write() call because it may end up increasing the file size. Although the calls involving vnodes are too diverse to be grouped, the above scheme is sufficient to determine the required locking type for all system calls involving vnodes. There are no operations that require that two vnodes be locked at the same time, so avoiding deadlock is trivial in this case.

It can now be shown that concurrent opening of files is safe, and ends up making only one vnode object in any case. Opening of a file always starts with a lookup performed on a path, which results in a <*file system*, *inode number*> identity. The first thread opening the file finds no vnode object for this identity so it creates one, and locks it while it opens the file on the FS (retrieving the details of the node at the same time, using the REQ_GETNODE FS request – we called this “fs_open” in the last subsection). Another thread opening the same file performs the same procedure, but it will find the newly created vnode object, and be suspended acquiring a lock on it. Once the original thread finishes retrieving the details (and completes the operation on the vnode), it will unlock the vnode. The suspended thread then resumes, and finds that the vnode (to which it now holds a lock) has a positive reference count already. It does not have to perform the REQ_GETNODE FS query itself, it can just increase the in-VFS vnode reference count and continue immediately.

*Creation* of a file (on the FS, not just the vnode) is slightly different. The REQ_CREATE FS request (referred to as “fs_create” in table 3.7) creates a new inode on the FS, opens it, and returns its details all in one go. Therefore, no vnode can be locked in advance of the create request, because the full identity is not known beforehand. This explains why creating a file requires an exclusive lock on the *vmnt*: between the create call being performed by the FS, and the subsequent reply arriving at VFS, another thread might open the newly created file, change some of its details (e.g. write to it, increasing its file size), and close it. Once the delayed REQ_CREATE reply is received and handled, the node details from this reply are then not up to date any more. This is prevented by making sure that during the creation process, no open takes place. Opening files only takes VMNT_READ access, so creating files is set to require VMNT_EXCL access. An alternative solution would be to lock the created vnode afterwards and then perform another REQ_GETNODE call to get the latest details (throwing away the node details from the REQ_CREATE reply), but this involves more message overhead even in the best case.

A somewhat similar problem is that there are certain system calls that can make changes to files based on their path rather than on their vnode. These calls make up the “file modification (possibly replicated)” group from table 3.7. In these cases, the modifications still have to be replicated properly in VFS. In the original VFS implementation, these calls would just look for a vnode and update the vnode details if one was found. This approach now poses a risk with respect to consistency: modifications that are made based on a file descriptor (e.g., fchown()) only lock the vnode and not the vmnt during the actual REQ_CHOWN FS call, and modifications based on a path (e.g., chown()) only lock the vmnt and not the vnode. This means the modifications are not serialized at the VFS level, so that the replies to their FS requests may cross each other, resulting in a wrongly replicated value.

This has been solved by changing the trunc/chown/chmod calls to find a vnode for the file that is about to be modified, based on the lookup results. If no vnode is found at first, the file system is locked exclusively rather than just writelocked. After acquiring the exclusive lock, the operations try to find the same vnode again. If the vnode now exists, it can be locked. If it does not, then no vnode will be created for this inode until the whole modification operation completes, because the exclusive lock prevents files from being opened in parallel. An alternative here would be to open the file on the FS when making path-based modifications, and and to close it when done. This would again introduce more message overhead.

There is one last issue in the interaction between vmnts and vnodes, which is that vnodes are also used for working and root directories of processes. There are several places in VFS that iterate over all processes, performing a check on their working and root directory. For example, the unlink() call fails if someone's working/root directory is being deleted. Such checks must always be mutually exclusive to processes changing their own working/root directory. That is, from the point of view of other processes, every process must have valid and up-to-date a working and root directory at all times. This required small changes to various pieces of code.

We have now covered vnode creation, destruction, and modification, as well as vmnt/vnode interaction, for vnodes that represent normal files and directories. For pipes (both anonymous and named), character-special files and block-special files, additional considerations and restrictions apply. These will be covered bit by bit in later subsections.

For practical reasons, we have implemented different vnode calls for opening and creating files: 'get_vnode()' and 'attach_vnode()'. The former requests the inode's details using a REQ_GETNODE call if the vnode object is created for the first time. The latter takes the inode details as input, and assumes that no vnode exists for the newly created file. The final implementation has five 'get_vnode()' and six 'attach_vnode()' calls. Besides these two new calls, there are 'lock_vnode()', 'unlock_vnode()' and 'upgrade_vnode_lock()'. The last one is only used for pipes in the open() implementation. There are only five calls to 'lock_vnode()'. This is because of centralized locking of vnodes and filps, which will be discussed in the next subsection. There are 40 calls to 'unlock_vnode()'.

### *3.5.7. Filp (file position) locking*

The third object type is the 'filp' object type. The main fields of a 'filp' object that are shared between various processes (and therefore, between threads) and can change after the creation of the object, are 'filp_count' and 'filp_pos'. The former is the reference count of the filp object. The latter is the position within the file that the filp refers to. The file reference is stored in ‘filp_vno’ as a pointer to a vnode object. As stated before, like vnodes, filp objects cannot disappear if the current thread’s process opened them.

Changes to, and reads from, the ‘filp_pos’ must be mutually exclusive, as every call must see the latest version even across FS calls. The calls that change ‘filp_pos’ include the basic read() call, as this call advances the file position. This means that two read() calls that share the same filp object must not take place concurrently. It is therefore not useful to introduce multiple locking levels for filp objects. Instead, we use a simple mutex (as offered by the threading library) to make *all* access to filp objects mutually exclusive.

System calls that involve a file descriptor most often access both the filp that the file descriptor links to, and the vnode that that filp links to. The locking order imposes that vnodes be locked before filps. We take care of this at the filp level. Whenever a filp is obtained based one of a process’s own file descriptors, the corresponding vnode is locked with a certain requested vnode locking level first.

This has all been hidden away behind a single ‘get_filp()’ call. This call takes a file descriptor number and vnode locking type as input. It finds the filp object based on the file descriptor number and the current process’s ‘fproc’ structure. If the file descriptor was indeed valid, the call locks the filp’s associated vnode with the desired locking type, and locks and returns the filp. The filp can be unlocked later, at which point the vnode is unlocked as well, if requested. Our analysis indicated that locking the filp and not the corresponding vnode was not desirable, so this is not possible either.

The last point has an additional advantage: whenever a vnode is locked exclusively (that is, locked with the VNODE_WRITE lock type), all corresponding filps are implicitly locked as well. This is particularly useful because multiple filps may need to be locked at the same time, namely in the case of pipes:

- When opening a named pipe, VFS must make sure that there is at most one filp for the reader end and one filp for the writer end.
- Pipe readers and writers must be suspended in the absence of (respectively) writers and readers.

- To keep the file size of pipes low, the file position of both the reader filp and writer filp is reset to zero whenever possible. This can in fact happen after a read().

In all three cases, both the reader and the writer filp may need to be locked in an arbitrary order. However, these two filps are always linked to the same vnode (after all, they are for the same pipe). Therefore, deadlocks are avoided by always exclusively locking the corresponding vnode first.

This does mean that even during read() calls, vnodes for pipes have to be locked with the VNODE_WRITE type. To do this semi-transparently, the 'get_filp()' call supports a special flag indicating that if a VNODE_READ or VNODE_OPCL lock is requested on the vnode, and the vnode is a pipe, a VNODE_WRITE lock is acquired instead. This flag is used in the implementations of the open(), read() and close() calls.

The resulting VFS code has 20 calls to the extended implementation of 'get_filp()'. The newly added 'lock_filp()' code has only four calls to it, but one of them is from 'get_filp()'. There are 51 calls to the new 'unlock_filp()'.

### *3.5.8. Lock locking*

The remaining object types listed in subsection 3.5.2 are the 'lock', 'select' and 'dmap' structures.

The 'lock' structure keeps information about locking of file regions; they are not to be confused with the threading type of locking. For the 'lock' objects, it was determined that no mutual exclusion was required to maintain object integrity: none of the functions that access these objects make calls that can block the executing thread anywhere. As our threading model is nonpreemptive, this means that all access to those structures is already fully atomic.

The other two structures do not share this property, and we will discuss these in the following two subsections.

### *3.5.9. Select locking*

A single select() call may request the status of multiple file descriptors, and therefore multiple filps, after each other at once. If a select() call cannot be fulfilled immediately, all information pertaining the select() call is stored in a ‘select’ structure to allow for resumption later. Callbacks from other parts of the VFS code may cause such resumption when the status of one of the filps changes or when a timeout occurs.

Providing mutual exclusion for the ‘select’ structure proved to be difficult. The individual filps can be locked one by one without problem. However, the structure of the select implementation makes it impossible to apply per-object locking of the ‘select’ objects: there are several routines that operate on one ‘select’ object and then make calls to routines that iterate over all other ‘select’ objects.

Short of a full rewrite of the select code, the only solution is to use one global mutex covering all of the select code. The select code requires access to filp structures, so this select mutex must always be locked before filps, that is, it is higher up in the locking order:

select > filp

The next problem is that VFS makes certain callbacks to the select code while holding locked filp objects. To prevent deadlock, the select mutex must always be locked before a filp whenever a callback to the select code can be made. Fortunately, this problem is restricted to pipes. The only places from where such callbacks can be made, are also the places that can access both reader and writer filps for a single pipe. The writelock-vnode-if-pipe flag feature of ‘get_filp()’ from the last subsection was therefore extended to acquire the select mutex if the requested vnode is a pipe. An extra field, ‘filp_select_mutex’, was added to the filp structure. This field indicates whether a lock to the select mutex was acquired, so that the corresponding ‘unlock_filp()’ call can then unlock the select mutex.

Although this is a suboptimal solution, it only affects pipes. Pipe operations therefore do not benefit from our unblocking of VFS. Of course, it also does not make the current situation worse. The requirements for this project do not involve pipes at all, so we consider the solution to be acceptable.

### *3.5.10. Devices and drivers*

This leaves the ‘dmap’ objects, and for that matter, all code concerning devices and drivers in general. Communication with character device drivers was blocking before, and this was left as is. Unlike block device drivers, character device drivers already had the possibility to suspend a process performing a call on a character-special file. All other character device driver calls are expected to be short.

Unfortunately, the calls involving devices, drivers, and the ‘dmap’ structure are so diverse and widespread that no locking system could possibly adhere to the locking order established so far. To maintain object integrity anyhow, all blocking calls were eliminated from that code instead, resulting in all ‘dmap’ access being atomic. The blocking calls in this part of the code were limited to the ‘clone_opcl’ and ‘newdriver’ FS requests.

The ‘clone_opcl’ (REQ_CLONE_OPCL) request is made when opening certain character-special files. It creates an inode on the root device, with a new minor device number, as an instance of a device (e.g., a specific TCP socket as an instance of “/dev/tcp”). The call making this request turned out to be sufficiently detached from the rest of the device code that no special care had to be taken here. However, the implementation of this call had to be rewritten in such a way that it would not violate the locking order. We also had to make the assumption that a cloned device can not be reopened after a device crash. This is at least currently a valid assumption, as the new driver will not have the required state to do anything meaningful with the minor device number.

The ‘newdriver’ (REQ_NEWDRIVER) request is used for recovery from driver crashes. It notifies FS processes that there is a new driver endpoint for a block device. All data transfer from and to block device drivers goes through FS processes, whether the block device is mounted or not. If the device is mounted, the communication goes through that FS process; if not, it goes through the file server mainaining the root device (both using the REQ_BREADWRITE FS call). To make the ‘newdriver’ messages nonblocking, the driver update requests were turned from blocking FS calls into an update system that is integrated into the fs_sendrec() level as mentioned in subsection 3.4.1. This will be discussed in the next subsection.

### *3.5.11. Recovery from driver crashes*

Besides making the device code nonblocking, there is another argument for dealing with ‘newdriver’ messages at a lower level: driver updates are critical and as such deserve high priority. Whenever a crash occurs in a block device driver, the FS servers operating on any of its devices will cease to function as well. There is therefore no point in sending more requests to a FS server that is operating on a device with a crashed driver. At least, not until a new driver has been loaded and the FS server has been told about it.

Crash detection can come from two sides. The FS process itself could detect failure in communication with the driver, in which case it will reply to the corresponding request from VFS with special “driver died” error codes – at least in theory, this has not been fully implemented in MFS yet. Or, the MINIX Reincarnation Server (RS) could detect a crashed driver, launch a new one, and tell VFS about it.

It is therefore desirable for VFS to queue further requests to a FS server whenever that server has reported a driver crash. Once a new driver has been loaded and the FS server has been informed about it, it should resend all queued requests. Even if the FS server has not reported a crash yet, it is desirable that those ‘newdriver’ messages be sent with a higher priority than normal requests.

This has been implemented in the ‘fscom’ layer, on a per-FS basis. It operates right under the vmnt locking layer. The relevant state is saved as a separate structure within each vmnt structure. The threading approach allows alls its details to be hidden away from the layers above. Only the ‘fs_sendrec()’ and the ‘fs_newdriver()’ calls are exposed to worker threads. The former call will put the calling thread to sleep until the message has been sent *and* replied to. The latter call must not block, and merely queues the REQ_NEWDRIVER request for later sending. This queue has been implemented as a bitmap of major device numbers rather than a real queue. Only the latest driver endpoint for each major device number has to be sent, so there is no need to queue multiple *newdriver* requests for a single device.

The ‘fscom’ module then distinguishes between two types of requests to the FS process: basic requests, that is, all the normal FS requests, and ‘newdriver’ requests, i.e. the REQ_NEWDRIVER requests. The module limits the number of concurrent basic requests to the value specified by the FS (see subsection 3.4.4). It puts to sleep threads that cannot yet send their requests. Only one ‘newdriver’ request is allowed at a time, to prevent multiple updates for the same device from crossing each other. The single ‘newdriver’ requests are mutually exclusive from (potentially many concurrent) basic requests. No new basic requests are sent when there is a ‘newdriver’ request that has not been replied to yet.

If a basic request results in a “driver died” error code, the FS is marked as having no valid driver. All basic requests are then queued until a ‘newdriver’ request has been made, sent and acknowledged. The basic request that resulted in a “driver died” error code is simply put back into the queue of basic messages, and resent whenever a new driver has been installed this way. It is the file server’s responsibility to make sure that calls that fail with a “driver died” error can be resent later without any side effects.

Table 3.13 lists the fields used in the ‘struct fscom’ structure that is a part of every vmnt structure.

| **Field** | **Description** |
|---|---|
| int fsc_max_reqs; | The maximum number of concurrent requests allowed to this FS, as specified by the FS server (see subsection 3.4.4). |
| int fsc_nr_reqs; | The current number of outstanding requests to this FS. |
| int fsc_error; | Flag indicating whether a previous FS request has resulted in a “driver died” error, stopping any basic requests if set. |
| int fsc_driver_req; | Flag indicating whether there is a ‘newdriver’ request currently pending at the FS server, stopping any basic requests if set. |
| int fsc_driver_bits; | The number of ‘newdriver’ requests that have yet to be sent. |
| bitchunk_t fsc_bitmap [BITMAP_CHUNKS(NR_DEVICES)]; | The bitmap for major device numbers, each bit indicating whether a ‘newdriver’ request must be sent about that device’s driver. |
| struct queue fsc_queue; | The queue of processes that want to send a basic request. |

*Table 3.13: fields of ‘fscom’ vmnt substructure*

There is a high degree of similarity between this system and a basic readers/writer type lock, but to be able to accommodate for the specific needs of the fscom layer (the maximum on concurrent basic requests, and the special way the ‘newdriver’ requests are queued), we did not use the three-level lock here.

### *3.5.12. Block-special files*

As mentioned, the individual file servers are responsible for the block data transfer from and to the device that they are mounted on. This has another implication: mounting and unmounting may cause the FS endpoint handling the block transfer to a device to change. The FS server used for the block transfer might also be different from the FS that owns the vnode of the corresponding block-special file. As such, locking the vnode does not prevent concurrent unmounting of that FS. To prevent conflicts, the system calls involving the endpoint of the FS server responsible for that device (stored in the vnode as the ‘v_bfs_e’ field) must be mutually exclusive from mount and unmount operations.

A global ‘bfs’ mutex was added to serialize all operations involving block-special devices. This however presented an issue with respect to the locking order. The operations on a block-special file always take place while holding a vnode lock, whereas the mounting process involves creation and retrieval of vnodes. Fortunately, the latter never involves vnodes for block-special files. To maintain the locking order, an exception was made for the ‘bfs’ mutex, splitting in two the ‘vnode’ part of the locking order:

block-special vnode > bfs > non-block-special vnode

Although a finer-grained lock would be preferred, the use of block-special files is not that common. The complete mutual exclusion does have the side effect of serializing all mount and unmount requests as well. The handling of vmnts already was safe from concurrent operations, so this is not necessary. On the other hand, mounting and unmounting are not very common operations either.

### *3.5.13. Exec locking*

One final global lock had to be added to keep the stack usage of individual threads down to a minimum (see subsection 3.3.1). A single local variable was identified to be solely responsible for increasing the stack size requirement from about one kilobyte to over four kilobytes: the four-kilobyte ‘buf’ buffer put on the stack by the ‘patch_stack()’ function as part of the exec() implementation. This buffer was therefore made static, essentially making it a global variable. To protect multiple concurrent exec() calls from interfering with each other while using this buffer, an additional global ‘exec’ mutex was introduced. This mutex covers all of the exec() implementation code. It is therefore higher up in the locking order than any other object that can be locked by the exec code:

exec > vmnt > vnode

The result is that only one program execution can take place at a time.

## 3.6. Evaluation

Even though it took much more work than expected, the resulting multithreaded VFS locking implementation fulfills the stated requirements, provides the concurrency desired for the MMFS layer created on top of it, works perfectly in this regard, and leaves the actual VFS code fairly readable. In that sense, our efforts were successful, even though there are still a number of rough edges (for example, the global select and bfs locks).

However, due to time constraints we have been able to do only limited testing. In general, testing and debugging has become much harder with these changes. Needless to say, this is the downside of multithreading any program – even with nonpreemptive threading, the number of states that VFS can be in grows exponentially. It is no longer feasible to test all possible combinations of thread execution paths, even with just two threads. As such, we can not fully eliminate the possibility that we have introduced new bugs.

Also, the modifications have made VFS much more “read-only.” As we have shown, many of the decisions for the current locking model involved many simplifying assumptions. Although these were necessary to keep the unblocking effort feasible for this project at all, many of the assumptions may change as a result of future code changes and additions. This might subsequently require rethinking of the whole locking model. In addition, many subtleties of the current locking model are not at all apparent from the code. For example, the locking order is neither obvious nor enforced at the code level in any way. This way, even small modifications could have a large impact on the correct functioning of VFS as a whole.

# Chapter 4. MMFS part 1: the file system side

At this point, it makes sense to discuss the first part of the MMFS multimedia file server. This first part consists of the VFS-FS protocol implementation and file system aspects. First, we define the file system format as stored on disk (section 4.1). Then, we address how the MMFS implementation handles the requests from VFS, and we cover the practical aspects of creating and mounting a MMFS partition (section 4.2). Chapter 6 will cover how MMFS handles multimedia streams and communication with the disk driver.

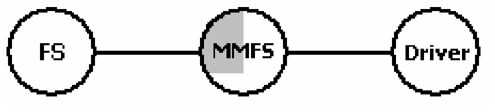


*Figure 4.1: the MMFS process, part 1*

The choice of making MMFS read-only simplifies many issues regarding the design of the file system format. This allows the actual format and the resulting implementation to be as simple and efficient as possible. The types of applications that could make use of the result are those that require no more than read-only access, for example local video serving applications, video-on-demand servers, and digital libraries.

## 4.1. The file system format

In the following subsections, we will establish the MMFS file system format. Simplifications and other practical aspects result from contiguous layout of files (subsection 4.1.1), independence from the internal format of files (4.1.2), and the lack of a need to store file metadata (4.1.3). Subsection 4.1.4 will then define the actual file system format.

### *4.1.1. Contiguous file layout*

Optimizing on data retrieval from hard disks has become harder as disks have become more complex over the years. In many cases, the phyiscal disk layout and parameters of modern disks are not known to the operating system at all any more. MINIX does not even try. Instead, the low-level communication protocol between the operating system and the disk makes use of Logical Block Addressing (LBA), where disk blocks are accessed by a single block number rather than by physical disk position. This allows the disk to define its own mapping from block numbers to physical disk locations.

We make the assumption that LBA blocks that have block numbers close together, will typically be physically close to each other on disk as well. This allows for optimization by grouping together requests with similar block numbers using an algorithm like SCAN.

On the disk, we therefore group related data close together. This minimizes the time it takes to retrieve a large chunk of such related data at once. With the file system being read-only during operation, the layout of files can be chosen fully in advance (i.e., “offline”). The optimal choice is then to lay out all files fully contiguously on disk.

Quite possibly, actual file access by end users will follow a Zipf-based distribution in practice [31]. This means that it is advantageous not only to have all blocks of a single file grouped close together, but also to have the most popular files grouped together. This leads to an optimal disk layout by following the “organ pipe” approach [31] if the relative popularity (i.e., relative frequency of use) of the files is known at file system creation time. In this approach, the files are sorted by popularity. The first, most popular file is placed in the middle of the disk, after which the second and third files are placed on the left and right side of the first file, the next two on the left and right side of the second and third, and so on. Figure 4.2 illustrates this.

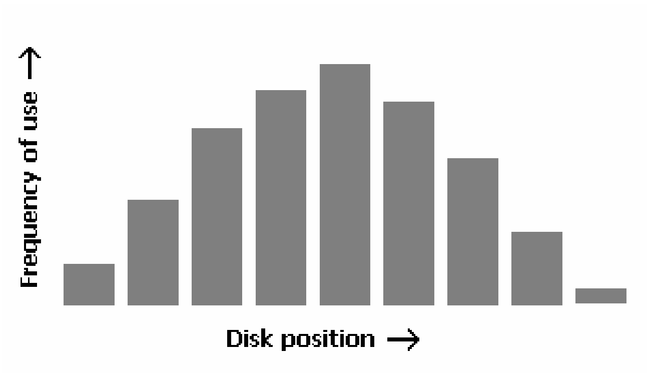


*Figure 4.2: “organ pipe” file positioning*

### *4.1.2. Independence from file formats*

We have made the file system format, and subsequently the implementation of the MMFS server, completely independent from the actual internal file format of the files used. This gives maximum flexibility in the actual use of the file system. The next chapter will show that frame boundaries inside multimedia files become irrelevant with sufficiently coarse-grained prefetching. We therefore simply ignore frame boundaries at the file system level, at the cost of more memory consumption by MMFS.

One side effect of this approach is that implementing high-level video functionality in MMFS like fast-forwarding in multimedia streams becomes impossible. Without knowledge about individual frames, no effort can be made to retrieve a small subset of those frames at a higher rate. One possible solution, should one be desired, is to generate a separate fast-forward file for each multimedia file in advance. That file contains only the frames necessary for fast-forward. This avoid the problem altogether, at the cost of having each multimedia stream require more disk space [5].

### *4.1.3. Absence of file metadata*

The actual file system will always be only a part of a larger system providing multimedia streams to interested parties, so there is no need for the file system to be fully self-contained. We assume that any file-specific metadata of the multimedia files is available elsewhere, for example on the same partition as the applications that read from the multimedia files.

File names are therefore not necessary at all: the file system can use a simple numbering scheme for all file nodes, and base the names on the node numbers. The file system does not have to store the file names anywhere, and the path resolution in MMFS can be simplified as well. This scheme has been implemented in MMFS in the form of strings containing six-digit decimal numbers for all file names, i.e. “000000” for the first file, “000001” for the second, and so on. Note that in this context the first file is the first file as sorted by frequency of use, not as sorted by its position on disk (see also subsection 4.2.2).

The file system does not hold information such as stream rates either. Picking a stream rate is left up to the applications reading from streams, and that allows several applications to pick different stream rates for a single file. As we will show in chapter 6, this is useful for our approach to variable-bitrate files.

### *4.1.4. Resulting file system format*

We can now define the resulting file system format. The first block of partition contains a small ‘superblock’ structure at the very start. This structure contains information about the whole file system: a unique identifier for the file system, followed by the number of file nodes and the total partition size (table 4.1). Directly following this structure is an array with information about all the ‘sb_nodes’ nodes. Each array element contains the position and size (both specified in bytes) of a single file (table 4.2). The file naming/numbering scheme is implicit: the node number and name of each file are based on its index into the array.

| **Field** | **Size** | **Description** |
|---|---|---|
| sb_magic | 32-bit | File system identifier, always set to 0x4d4d3141 (“MM1A”). |
| sb_nodes | 32-bit | The number of nodes (files) on the file system. |
| sb_size | 64-bit | The total size of the partition, in bytes. |

*Table 4.1: the superblock structure*

| **Field** | **Size** | **Description** |
|---|---|---|
| fn_pos | 64-bit | The position of the start of the file on the partition, in bytes. |
| fn_size | 64-bit | The total size of the file, in bytes. |

*Table 4.2: the file node structure*

It can be expected that large hard disk and partition sizes will be used to keep multimedia files on. For this reason, both the design and the implementation fully support 64-bit disk and file sizes. All values are stored in big-endian format. In our implementation code, the 64-bit values are stored as two 32-bit values, with the upper 32 bits first, followed by the lower 32 bits.

The format deliberately makes no statements or assumptions about block sizes. This is not necessary: any implementation reading from the file system can make its own decision regarding how to map bytes to blocks (if at all). With contiguous file layout as a given, this is not a file system format issue. In principle, the file system format even allows that the end of one file and the start of another file fall within the same disk sector. MMFS does assume that all files are aligned to the block size that MMFS uses internally, but that is merely for implementation simplicity.

## 4.2. Implementation

The following two subsections discuss the implementation of the VFS side of MMFS (subsection 4.2.1) and the way MMFS partitions can be created and mounted/unmounted (subsection 4.2.2).

### *4.2.1. Handling of VFS calls*

The simplicity of the file system format allows all node data to be stored in memory. Upon receiving the initial REQ_READSUPER request from VFS, MMFS fetches the superblock and node data from the start of the disk, validates it, and stores the list of files in an array. With two exceptions, all subsequent calls can be processed immediately and without having to block the calling user process due to required interaction with the disk driver.

The exception are REQ_READ and REQ_QUERYFS. These two calls will be covered in more depth in chapter 6. For now, it is relevant that saving and resuming state for these calls in MMFS is relatively simple. The use of threads was therefore not deemed useful for MMFS. A form of continuations is used instead: the state of pending calls is stored in “stream” objects (see chapter 6). This allows MMFS to continue serving other requests in the meantime.

Many of the VFS requests are write operations which fail right away. Some other operations are not implemented because they are not relevant. Access control was not of any concern in this project. The root directory is readable and searchable by everyone (access mode 555 octal), and all files are readable by everyone (access mode 444 octal). The root directory and all files are owned by MMFS’ own process user ID (typically root). All file times are set to zero.

Table 4.3 summarizes the MMFS implementation of the various VFS requests.

| **Request** | **Action** |
|---|---|
| REQ_GETNODE | Increase the given node’s reference count and return its node details. |
| REQ_PUTNODE | Decreases the given node’s reference count. |

| | |
|---|---|
| REQ_PIPE | Return the ‘ENOSYS’ error. MMFS must not be the root device and therefore does not need to support creation of anonymous pipes. |
| REQ_READ | Fulfill the given read request, suspending the call until data is available if necessary. See chapter 6. |
| REQ_WRITE | Return the ‘EROFS’ error, as this is a write operation. |
| REQ_CLONE_OPCL | Return the ‘ENOSYS’ error, MMFS may not be the root device and therefore does not need to support virtual device cloning. |
| REQ_FTRUNC | Return the ‘EROFS’ error, as this is a write operation. |
| REQ_CHMOD | Return the ‘EROFS’ error, as this is a write operation. |
| REQ_CHOWN | Return the ‘EROFS’ error, as this is a write operation. |
| REQ_ACCESS | Test the given access mask against the given node’s generated access mode, and return ‘OK’ or an error code based on the outcome. |
| REQ_MKNOD | Return the ‘EROFS’ error, as this is a write operation. |
| REQ_MKDIR | Return the ‘EROFS’ error, as this is a write operation. |
| REQ_INHIBREAD | Return ‘OK’, this call is on node basis rather than file descriptor basis, so nothing useful can be done as a result of it. |
| REQ_STAT | Fill a buffer with basic information about the given node and copy the result to the user process’s address space. Of the fields filled in, only the ‘st_dev’, ‘st_ino’, ‘st_mode’ and ‘st_size’ fields do not contain hardcoded values. |
| REQ_CREATE | Return the ‘EROFS’ error, as this is a write operation. |
| REQ_UNLINK | Return the ‘EROFS’ error, as this is a write operation. |
| REQ_RMDIR | Return the ‘EROFS’ error, as this is a write operation. |
| REQ_UTIME | Return the ‘EROFS’ error, as this is a write operation. |
| REQ_QUERYFS | If the subcall is FSTATFS, copy out the block size used to the user process’s address space. If the subcall is STREAMCTL, pass it on to the stream module. See chapter 6. |
| REQ_LINK | Return the ‘EROFS’ error, as this is a write operation. |
| REQ_SLINK | Return the ‘EROFS’ error, as this is a write operation. |
| REQ_RDLINK | Return the ‘ENOSYS’ error, as symbolic links are not supported. |
| REQ_RENAME | Return the ‘EROFS’ error, as this is a write operation. |
| REQ_MOUNTPOINT | Return the ‘ENOSYS’ error, as mounting on top of MMFS is not supported. |
| REQ_READSUPER | Read the superblock data from disk. When the reply arrives, validate and store its contents, and reply to the VFS call. |
| REQ_UNMOUNT | Refuse to unmount if any of the nodes still have a positive reference count. Otherwise, simply return ‘OK’. |
| REQ_TRUNC | Return the ‘EROFS’ error, as this is a write operation. |

| | |
|---|---|
| REQ_SYNC | Return ‘OK’, nothing needs to be done. |
| REQ_LOOKUP | Process ‘/’-delimited path components, allowing each component to be “.”, “..” or a string representation of a node number. Return the details of the resulting node or the root directory, or an appropriate error or pseudo-error code if the lookup fails or continues in the parent file system. |
| REQ_STIME | Return ‘OK’, nothing needs to be done. |
| REQ_NEWDRIVER | Driver crash recovery. See chapter 6. |
| REQ_BREAD | Return the ‘ENOSYS’ error, raw block reads are not supported at this time. |
| REQ_BWRITE | Return the ‘ENOSYS’ error, raw block writes are not supported at this time. |
| REQ_GETDENTS | Fill the requested portion of the buffer with ‘dirent’ structures, starting with “.” and “..”, followed by the string representations of all node numbers. |
| REQ_FLUSH | Return ‘OK’, nothing needs to be done. |

*Table 4.3: MMFS implementation of VFS requests*

### *4.2.2. File system creation and mounting*

To create a MMFS file system on a disk partition, a special ‘mkfs’ utility was written:

mkfs-mmfs <*device*> <*listfile*> <*blocksize*>

It takes a partition or subpartition device path (e.g. “/dev/c0d0p1s0”), and will automatically determine the partition size from this device. To determine which files to put on the partition, it also takes a file containing a list of file paths (one per line). This list is assumed to be sorted by relative frequency of use, most popular first. The mkfs utility uses the organ pipe approach to position as many files on the partition as can fit. Once mounted by MMFS, the resulting file numbering scheme of the partition is in the same order as this input file. The given block size is used for alignment of the start of files to blocks, which is expected by our MMFS implementation.

To mount and unmount a MMFS file system, another pair of utilities was added:

mount-mmfs <*device*> <*path*>
umount-mmfs <*device*>

The mount utility mounts the (sub)partition identified by the given device on the given path and launches a MMFS instance to operate on it in read-only mode. The umount utility unmounts such a mounted MMFS instance. The executable path and label used for mounting and unmounting file systems is currently hardcoded to ‘MFS’ in the mount() and umount() calls that are part of the MINIX C library. These utilities are therefore a simple adaption of that code that replaces the path and label of MFS with MMFS. Extension of mount() and umount() to accept arbitrary file server executables and labels would be subject of future work.

# Chapter 5. A driver that supports multimedia

As stated in the introduction, applications must be able to read data from a multimedia file at a specified rate, without spending time in the individual read() calls for longer than strictly necessary. The VFS changes in chapter 3 already make sure that read() calls can not be blocked by other VFS operations. This fulfills in part the first and third goal of this project.

On top of VFS, MMFS minimizes read() latency by prefetching all data before it is read by the multimedia application. As such, MMFS acts as a buffer between the application and the disk, in a better way than the application itself ever could. This fulfills the remaining part of the first goal.

To let MMFS provide guarantees to applications, the prefetching itself must be guaranteed to finish in time in all cases. MMFS is then *work-ahead augmenting*. MMFS will not be the only file server on the system, so this requires support from the disk driver. The prefetching approach also has an advantage: the prefetch requests are available ahead of time, and that allows the driver to reorder the retrieval requests for minimal disk latency. This in turn allows more multimedia applications to be active at the same time, fulfilling the second goal.

Section 5.1 presents the basic model that we will use for MMFS and the disk driver. It establishes a division of tasks between MMFS and the driver, and describes the implications of this division. Section 5.2 translates the results into a set of FS/driver protocol extensions. Section 5.3 describes the implementation of these extensions in the disk driver. The MMFS implementation will be the subject of chapter 6.

## 5.1. Architectural decisions for the MMFS – driver protocol

To guarantee timely prefetching and minimize disk latency, we use the well-established model of data retrieval in *rounds*, based on the SCAN-EDF principle. In subsection 5.1.1 and 5.1.2, we elaborate on the model's general explanation of chapter 2 with details relevant for our case. Subsection 5.1.3 then divides the subtasks of this model between MMFS and the disk driver. Subsections 5.1.4 to 5.1.7 describe various requirements resulting from this particular division. These requirements form the base for the actual FS/driver protocol extension which will be described in section 5.2 later. Subsections 5.1.8 to 5.1.11 describe the conceptual implications of the split for MMFS. Finally, subsection 5.1.12 discusses alternative task divisions that we rejected.

### *5.1.1. Mapping from streams to requests*

We define a stream as a continuous (forward) flow of data from a file on the disk, to a user process (the "stream reader") that is interested in retrieving consecutive data from the file at a certain specified guaranteed rate. Each stream has a window of data that has been prefetched from the disk, but not yet read by the stream reader. Prefetching is not instant, so each stream also has a window of data that has been requested but is not yet retrieved. Combining these two windows, we get a larger window with a *split*. The head of the whole window moves forward as new prefetch data is requested from the disk. The split of the window moves forward as requested data arrives. The tail moves forward as the stream reader makes read() calls.

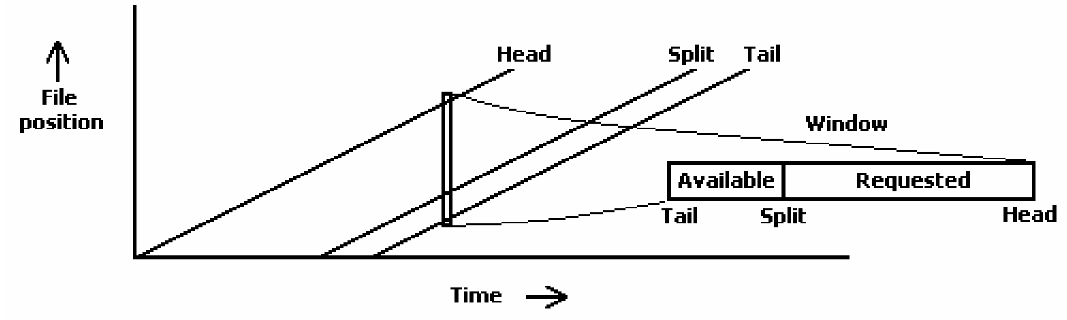


*Figure 5.1: basic stream progress*

Contrary to what figure 5.1 shows, moving forward of the stream's head, split and tail is not continuous. A stream reader will typically read one multimedia frame at a time, and the frames are usually not all of equal size. The prefetching process is not truly continuous either, as retrieval of data from disk can only be done in blocks. The prefetching for a stream is therefore broken up into individual *block requests*, one for each whole block. Figure 5.2 shows this.

Each block request carries the disk position of the block, the block size, and a destination buffer address at the very minimum. We are concerned with prefetching, so the destination buffer cannot be in the actual stream reader's address space. Either MMFS or the driver will have to maintain these buffers.

The rate of the stream is guaranteed if the split of the stream stays ahead of the tail at all times. In that case, read() calls need never be delayed until the data is available. This gives all block requests a *deadline*: the time at which the stream reader is expected to read the data.

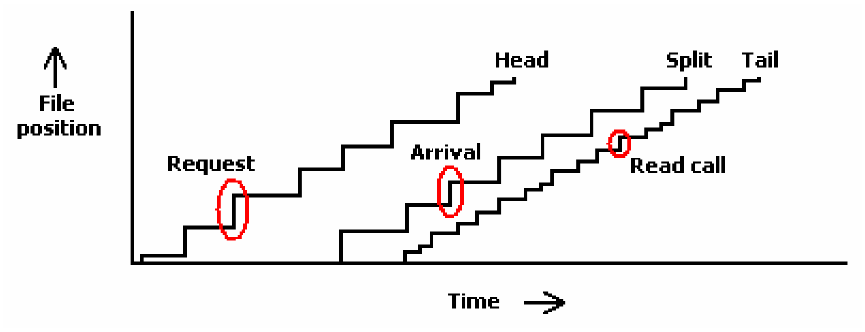


*Figure 5.2: realistic stream progress*

*5.1.2. Mapping from requests to rounds*

Our own tests (chapter 8) show that most speed gain at the disk can be obtained from retrieving disk blocks that are as large as possible. The maximum block size is bounded merely by the minimum number of streams that are to be supported. Sorting block requests in SCAN order then allows for further reduction of the total retrieval time.

To allow for SCAN sorting, we use the concept of rounds. The prefetch requests of all streams are gathered in batches. Within one round, all the requests all have their deadlines at or after the end of that round. That way, reordering will not make any request miss its deadline. EDF is used to determine which set of requests to process in the next round. Within each round, the requests are sorted in SCAN order.

There may also be nonmultimedia requests, originating from MFS instances. In the old situation, these are processed on in first-come-first-serve (FCFS) order. That is also applicable to the new model. A fraction of each round is reserved for nonmultimedia requests. This prevents starvation of such requests.

The basic model then consists of multimedia streams that are broken up into single requests that each have a deadline. These requests are gathered into rounds, based on their deadlines by using EDF, along with other nonmultimedia requests on FCFS basis. Each resulting round is SCAN-sorted, and all requests in that round are processed by the disk driver in the SCAN order.

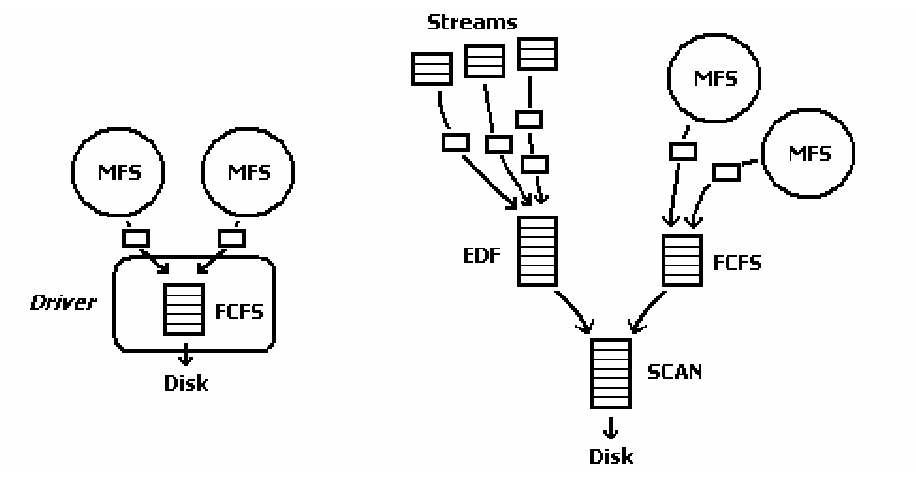


*Figure 5.3: the old (left) and new (right) situation*

Figure 5.3 shows the old and the new situation in this regard, limited to the file servers and the disk driver. The figure of the new situation deliberately leaves open the division of tasks. Combined, MMFS and the driver will have to maintain streams, buffers, requests, queues and rounds. The next step is the division of tasks and responsibilities between MMFS and the disk driver.

### *5.1.3. Division of tasks*

In a microkernel operating system like MINIX, the file servers and the disk driver are different processes that can communicate only through message passing. Needless to say, the division of tasks has a large impact on the message protocol used between those processes, as well as the implementation of the processes.

As a start, it is clear that at one end, MMFS instances must be in charge of determining where on disk the blocks are. File placement is an intrinsical part of a file system, and we have already defined this in chapter 4. The disk driver is not aware of the contiguous layout of files on disk, so MMFS must inform the disk driver of the position of individual blocks.

At the other end, the disk driver must be in charge of receiving both the multimedia requests and other requests originating from nonmultimedia file servers. Only at the disk driver level, requests from multiple file servers for one disk are gathered at all. Thus, only at this level can serialization of all requests into a desired order take place.

To fill in the parts in between, we have chosen the most clean separation: the MMFS file server is responsible for managing streams and block buffers and for issuing individual requests on behalf of streams on a periodic basis, whereas the disk driver manages the request queues, rounds, and the various sorting levels. Figure 5.4 illustrates this.

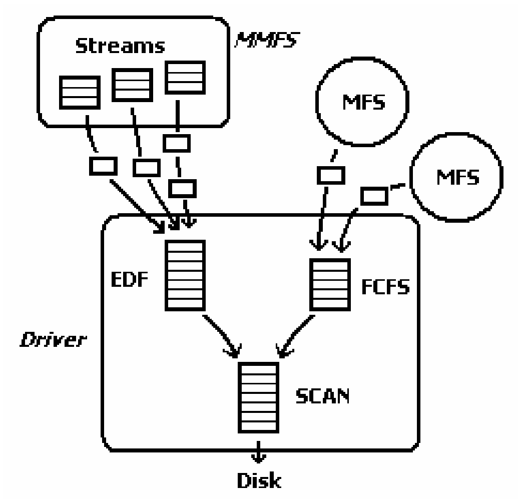


*Figure 5.4: division of tasks*

The chosen approach implies that MMFS is not aware of rounds, round boundaries and disk speed parameters. The driver manages all of this. The driver is also in charge of admission control. MMFS has to query the driver whenever it wants to add or delete a multimedia stream. The driver decides how many streams (and with which rates) can be actually admitted, making sure that the deadlines of all admitted streams can be met.

Other architectural approaches are possible, subsection 5.1.12 lists some alternatives that we have considered.

The following four subsections discuss the implications of the chosen division. The next subsection (5.1.4) looks at the practical aspect of round durations, with as result that static block sizes will be used. Static block sizes form the base for determining how far ahead of their deadline MMFS has to issue all requests (subsection 5.1.5). In turn, that defines how admission control must be done, and how far apart request deadlines for a stream must be (5.1.6). With these results, we can guarantee that all deadlines are met (5.1.7).

### *5.1.4. Round durations and block sizes*

With the test results in mind, we pick a round duration in the order of magnitude of one second. This allows the driver to get a substantial gain out of SCAN sorting. With one-second rounds, each round has to cover a full second worth of data (in the form of block requests) for each of the active streams.

Such a coarse granularity leads to significant buffering of prefetched data within MMFS. This saves MMFS from having to take into account the sizes of individual frames as they are read by the stream reader. For example, a stream reader may want to retrieve 25 frames per second, with differences in frame sizes of up to a factor 10. If the data for the stream reader is prefetched on a per-second granularity, then the factor 10 difference of individual frames is negligible as long as the maximum retrieval rate *per second* (i.e., the maximum sum of all frame sizes retrieved in any arbitrary period of one second) does not exceed the average retrieval rate.

This is a simple form of smoothing. True VBR streams can deviate from the average stream rate much more heavily, and require prefetching beyond one second. See subsection 5.1.10 for more on this.

With the prefetching granularity decoupled completely from the actual reading granularity of the stream reader, we can simplify by using a large static block size for all block requests of the stream, typically in the order of magnitude of hundreds of kilobytes. MMFS will use one static block size for *all* streams, to facilitate buffer assignment and sharing, but that is not a requirement of the protocol.

### *5.1.5. Request deadlines and the work-ahead time*

To make sure all the data is prefetched in time, MMFS has to send all the prefetch requests to the driver sufficiently far ahead of time. As mentioned in subsection 2.1.1, the maximum time it can take for a request to be served in the SCAN-sorted rounds model, is two times the maximum round time. Figure 2.1 already illustrated this: a request from MMFS may arrive at the driver just after a round has started, and must therefore be processed in the next round at the earliest. Because of SCAN-sorting, the result may come in only at the end of that round. The time difference between the arrival of the request and the result being delivered to MMFS, is equal to the duration of two full rounds.

For MINIX, we have to add a small epsilon value to account for communication overhead. Let us call the sum of two rounds and this epsilon value the *work-ahead time*, because that is the minimum time that all requests have to be made in advance. Requests with a deadline nearer in the future than the work-ahead time, are not *feasible*, and may not be served before their deadline. Such requests must therefore not be sent. To this end, the driver must inform MMFS of the work-ahead time; that happens upon stream admission.

This requirement automatically implies that all streams need a startup time of this same work-ahead time. Each application opening a stream must wait this startup time before it can start reading from the stream. Figure 5.5 illustrates this. With rounds of one second, this is a two-second delay. Subsection 5.1.9 will describe how this delay can be reduced under certain circumstances.

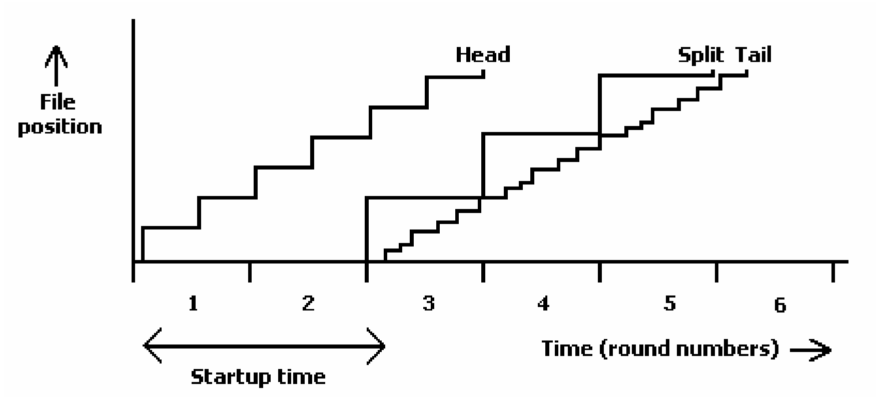


*Figure 5.5: startup time and worst-case arrival*

### *5.1.6. Stream rate granularity and admission control*

For the following discussion, we assume a deterministic admission controller. Like EDF scheduling in general [15], round-based EDF-scheduling is only guaranteed to work if the total utilization is less than 1.0 per round. Or, in practical terms, if all blocks in a round can be retrieved within the round's maximum duration. The admission controller therefore assigns a fraction of each round to each admitted stream. This fraction is based on the requested stream rate. If a new stream requests admission, and the sum of all current streams' fractions plus the fraction of the new stream exceeds 1.0, then the stream is rejected. In practice, a value just under 1.0 is used to allow for nonmultimedia requests.

Subsequently, the deadlines of requests of a single stream must never *force* the driver to exceed the stream's fraction *in any round*. This has two key implications:

1. For admission control, each stream rate used must be rounded up to the lowest matching per-round fraction.
2. The deadlines of each two requests for a stream must be sufficiently far apart, to guarantee that EDF will not cause too many requests to be included in any round.

Remember that all requests are for single blocks, all with the same size for each stream. As an example of the first point: a stream that has a user-requested stream rate equivalent to one block per three rounds, must be admitted with a stream rate equivalent to one block per round. Otherwise, the per-round stream fraction (1 block per round) may exceed the admitted overall stream fraction (1/3 block per round). Figure 5.6 shows this.

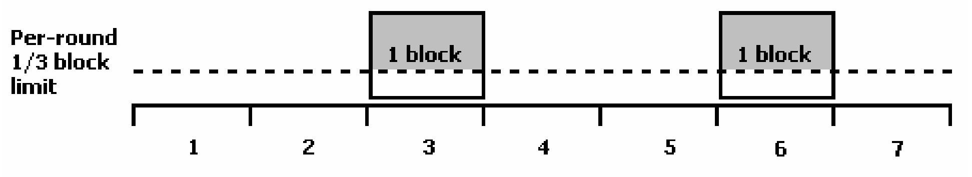


*Figure 5.6: possible per-round overutilization by one stream (1)*

The second point implies that if a stream is admitted with one block per round, no two requests must have their deadlines apart by less than a full round duration. Otherwise, the driver may be forced to include both requests into one single round (as EDF is used), and exceed the per-round stream fraction of 1 block per round. Figure 5.7 shows this: with no minimum time distance of a full round between deadlines, a round may end up including two requests for the stream. MMFS is not aware of round boundaries, so it cannot tune the deadlines to round boundaries either.

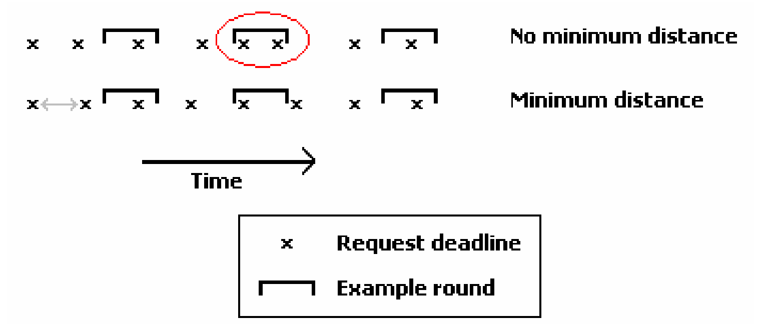


*Figure 5.7: possible per-round overutilization by one stream (2)*

Generalized, streams can only be admitted with a stream rate rounded up to “*N * block_size / round_duration*” bytes per second, where N is a nonzero positive integer. All requests’ deadlines must then be apart by at least “*round_duration / N*” seconds. The stream’s fraction can be expressed as N blocks per round. A practical restriction on the possible values of N comes from the MINIX clock frequency, but that will be discussed in subsection 5.2 when these points are put in practice.

The block size is chosen by MMFS, and the round duration is chosen by the driver. The driver performs the admission control, so MMFS must inform the driver of the block size when it requests a stream’s admission. MMFS generates the block requests, so after admission, the driver must inform MMFS about the minimum time between each two request deadlines, in addition to the work-ahead time (see the last subsection).

### *5.1.7. Guarantees*

Based on the previous subsections, we can now establish that a deterministic admission controller can guarantee that all deadlines of all requests be met. That follows from these points:

- The current round must include all requests that have a deadline before the (worst-case) end of the round after it. The deadline, after all, represents the time at which the requested block should be available to the stream reader.

- Recursively, we can infer that there are no deadlines that are before the (worst-case) end of the current round. This holds as long as both the previous point is followed, and the deadline of every newly submitted request is at least two round times ahead (the work-ahead time) of its submission time. The requests from the previous point are therefore the requests with the earliest deadlines, meaning they can be found with EDF sorting.
- Consequently, the minimum set of requests that have to be included in a round consists of those that have a deadline between the (worst-case) end of that round and the (worst-case) end of the next round after it.
- All deadlines are met if, in each round, the total time required to process the requests included in the round, is less than the (maximum) round time. All requests are then served before the end of this round, and the next round will start in time.
- The driver's admission controller admits a stream with two parameters: the block size used, and the requested stream rate. Based on this the driver determines maximum number of requested blocks per round, rounded up to an integer. Subsequently, the driver determines the (maximum) time fraction that it takes to retrieve these blocks in a round.
- The admission controller makes sure that the total sum of all such time fractions of all admitted streams plus the new stream never exceeds the predetermined maximum round duration. This is the actual admission test.
- The predetermined maximum round duration is divided by this number of blocks per round, to obtain a minimum time distance between the deadlines of each two requests for the stream that is tested for admission. The resulting time distance is sent to MMFS if the stream is admitted.
- MMFS then sends requests with deadlines that each have this minimum time distance apart. MMFS is not aware of round times, so can not align these requests to rounds.
- However, with this minimum time distance, in *any* randomly chosen time frame with the maximum duration of a round time, the number of requests for that stream with deadlines in that time frame, does not exceed the admitted number of blocks per round.
- Therefore, at *any* time that the driver starts a new round, the number of requests from one stream that have deadlines that fall between the (worst-case) end of that round and the (worst-case) end of the next round, does not exceed the admitted number of blocks per round for that stream.
- Therefore, retrieval of those blocks for that stream will never exceed the stream's admitted time fraction.

- Therefore, with all streams following the minimum deadline distance for their block requests, the total time duration of each round is never exceeded, since the sum of the fractions never exceeds one.
- Therefore, all deadlines are met.

This concludes the heart of the MMFS/driver communication model resulting from the chosen division of tasks. The next four subsections will discuss the model from MMFS' point of view, discussing block buffer requirements (subsection 5.1.8), an extension to reduce startup times (5.1.9), an extension to prefetch further than strictly necessary (5.1.10) and the implications of this "deep prefetching" for the protocol (5.1.11).

### *5.1.8. MMFS buffer requirements*

The number of blocks per round for a stream (N), together with the block size, represents an effective rate that is at least equal to the requested rate. Issuing requests at the same rate as the agreed-on time distance between deadlines is therefore sufficient for MMFS to keep the stream going at a continuous rate. The result is that issuing of requests starts to look like a perfect staircase. Each stair represents a single block, and each N stairs combined equal the maximum duration of one round (even though the requests are not aligned to round boundaries!). Again, stream reader's progress is not assumed to be as regular. Figure 5.5 already shows this perfect staircase, with N being two blocks per round.

From this, we can determine the minimum number of buffers that MMFS needs for each stream. With N still being the number of blocks that the stream requests during each round duration, the resulting minimum number of buffers that have to be available for the stream equals 3N: 2N requests may be pending at the driver at any time in the worst case, and the data resulting from fulfilled requests has to be available for reading by the stream reader for some time as well. Figure 5.8 shows how the perfect staircase of requests (assuming one block per round in the figure) translates into each buffer being in use for three round times at worst. This leads to the minimum of three buffers.

In terms of both driver admission control and MMFS buffer use, a single stream with N > 1 blocks per round is equivalent to multiple streams of 1 block per round. The staircase is simply steeper. The implication is that a MMFS instance that wants to support S streams, needs to have at least 3S block buffers in total.

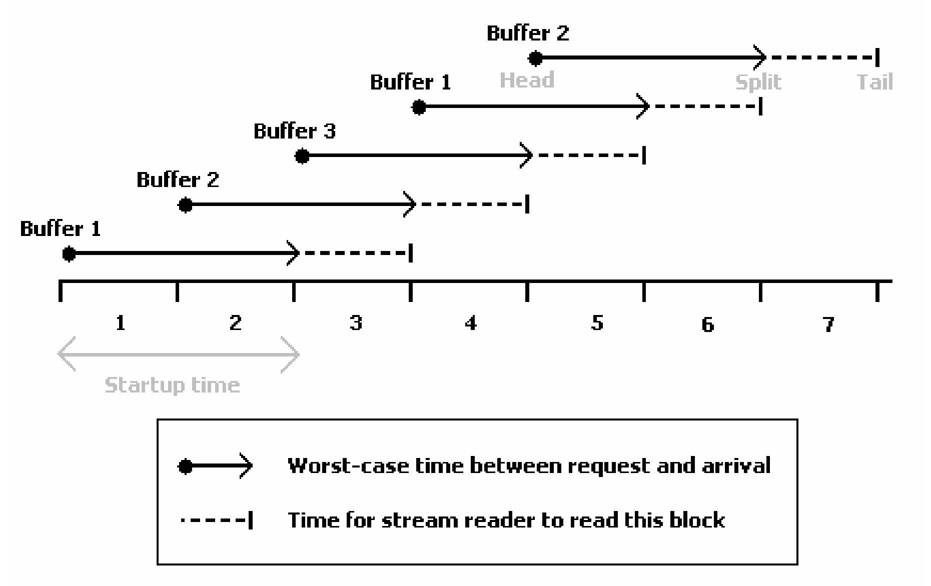

Buffer 2
Buffer 1
Head
Split
Tail
Buffer 3
Buffer 2
Buffer 1
1
2
3
4
5
6
7
Startup time
Worst-case time between request and arrival
Time for stream reader to read this block

*Figure 5.8: three buffers per stream*

Note in the figure, however, there is no overlap between the availability of consecutive blocks to the stream reader (the dashed lines). This means that the reading behavior of the stream reader must equal a similar perfect staircase. Otherwise, a buffer could be kept in use for too long, and that would prevent the next request from being issued in time.

Preferable would be 4S blocks in total: this allows the stream reader to make stream read requests that are not strictly aligned to the block size used in MMFS. Figure 5.9 shows the resulting progress; there is now overlap of the availability of filled buffers to the stream reader.

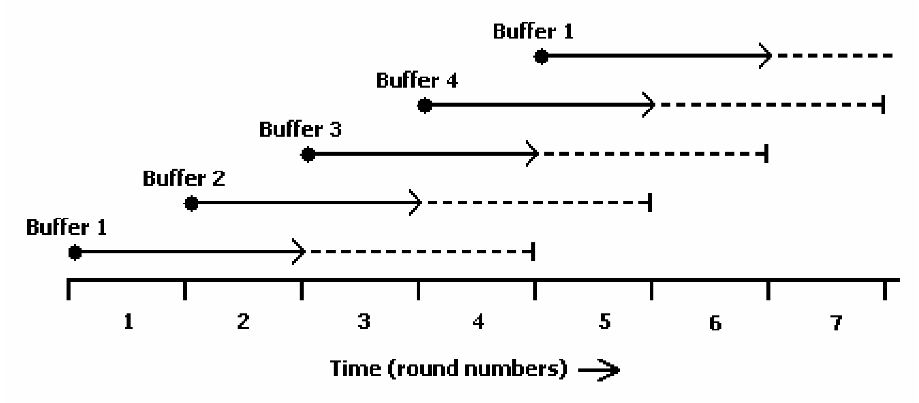


*Figure 5.9: four buffers per stream*

The increase by one block per stream from the requirements stated in subsection 2.1.3 is due to the fact that in our case, the requests have to be sent off far ahead of their deadline to match any potential round boundaries.

### *5.1.9. Reducing startup times*

Subsection 5.1.5 described that the startup time of a stream is in principle equal to the work-ahead time. If the disk is already being used to its maximum, this is unavoidable. However, in many cases the disk will be underutilized, and we can exploit that to reduce startup times.

To start off a stream, enough data has to be available to MMFS to serve the read requests of the stream reader during the retrieval time of the next requests. Waiting for the work-ahead time fulfills this requirement, but the requirement is also fulfilled if all the data is available before the work-ahead time.

Right after stream admission, MMFS can make requests for enough blocks to support the stream up to the work-ahead time. The deadlines of all but the first request will be further ahead than the work-ahead time. This gives the disk driver the opportunity to process all of the requests ahead of time, even though it does not *have* to. If the driver is fast enough in serving all the requests, we can reduce the stream startup time to a minimum.

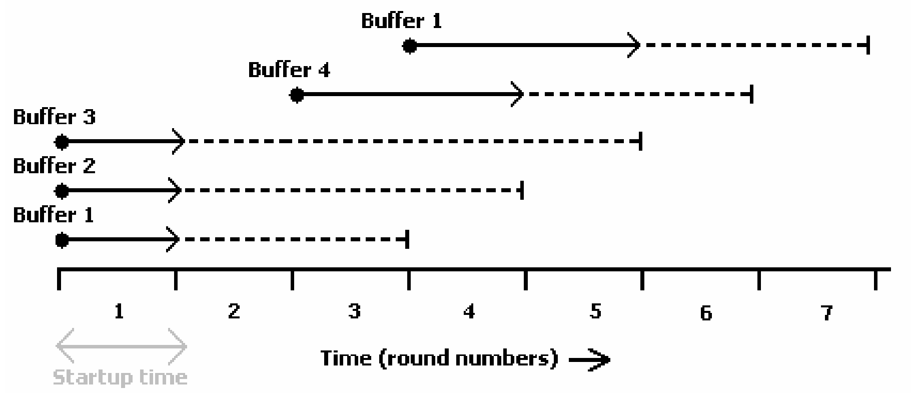


*Figure 5.10: example of reduced startup time*

Figure 5.10 shows how retrieval of the first three blocks within a little over one round’s time can lead to the stream exhibiting the same pattern as figure 5.9 one full round earlier (the arrows show the actual retrieval time in this case). In practice, it can be much shorter. On the other hand, if not *all* blocks arrive earlier than the work-ahead time, no optimization can be applied.

### *5.1.10. Deep prefetching*

In princple, our chosen model limits prefetching for each stream to a minimum. However, it is highly desirable to prefetch further ahead whenever more buffer space is available. This provides extra buffering that overcomes the effects of stream readers that unintendedly deviate from their stream rate.

Such “deep prefetching” is also the approach that we use to deal with true VBR streams. It essentially consists of smoothing the VBR stream so that we can still use constant-bitrate retrieval [2]. The result is a statistical take on VBR streams: stream readers can request a stream rate that is (far) below the peak rate of the VBR file, and then rely on deep prefetching to cope with the peaks in the data stream. The lower the chosen rate is, the more streams can be admitted, but the more those streams rely on prefetching. It is up to the application to decide on the lowest rate that still gives an acceptable loss probability.

To perform deep prefetching, MMFS instances can submit requests ahead of time. These requests then carry deadlines far in the future, and do not *have* to be served anytime soon by the disk driver. However, if the driver is not completely busy, the driver’s per-round EDF algorithm will automatically process requests relatively further ahead of their deadline.

### *5.1.11. Buffer reuse and overriding/cancelling requests*

MMFS maintains the block buffers, and it may submit requests with deadlines far ahead to facilitate deep prefetching. This combination leads to new issues that need to be addressed: those of buffer reuse and request cancellation.

With the deep prefetching described earlier, it is possible and even desirable that if a relatively small number of streams is active, these streams use up most available buffer space for prefetching. If the number of streams increases due to application demand, some of the buffer space has to be reallocated to these new streams. There might not be sufficient buffer space available to support all new streams otherwise.

The buffers most eligible for reuse are the ones that have the highest estimated use time distance (i.e. with the latest deadlines) for the originally active streams, as reusing those would limit the prefetch window of those streams by the smallest possible amount. Those buffers, however, are typically those that are waiting to be filled by the disk driver.

Figure 5.11 illustrates this. It shows how there are two streams at first (on the left). As three new streams are created (on the right), the first two streams have to give up their last buffers, including those that were requested from the disk driver. This effectively reduces their window at the head side, and requires that the outstanding requests be dealt with somehow.

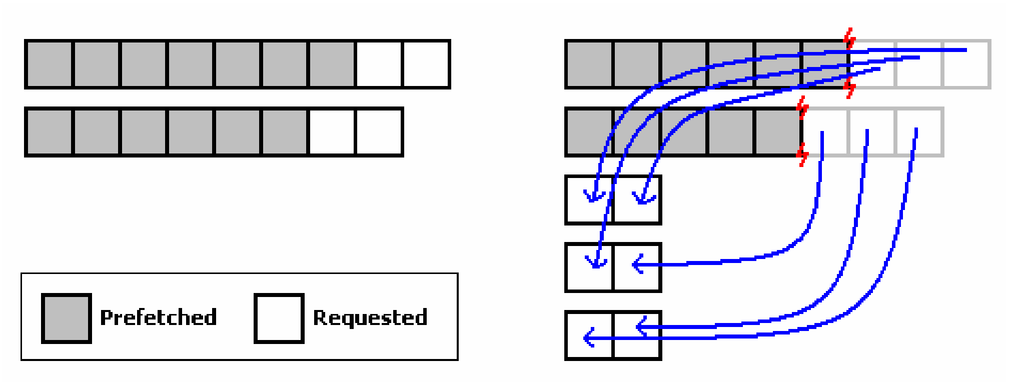


*Figure 5.11: buffer reallocation to new streams*

The deadline of the request for which a buffer will be reused, will always be earlier than the deadline of the original request involving that buffer. This ensures *fairness* of buffer usage between streams; we will elaborate on this in chapter 6. A race condition emerges if a new request is simply sent off without taking care of the old request: if the driver serves the new request and reports this to MMFS, and then serves the old request immediately after (storing the result in the same buffer), the buffer contents would be indeterminate from MMFS' perspective.

There are several ways to solve this. MMFS could wait until the request has been completed (i.e. the buffer has been filled) and only then reuse it, but that could literally take minutes if the given stream has built up a huge prefetch window. MMFS could also reserve an extra number of blocks to be able to support any streams that could possibly still be created, but this would be a huge waste of buffer space. A third option is to explicitly *cancel* pending requests at the driver with a deadline far ahead, and although this is a viable approach, there is a simpler alternative.

As section 5.2 will show, each request contains parameters that uniquely identify the destination buffer. The driver can therefore identify which requests are for the same buffer within the same MMFS instance. Each request also contains a deadline, so the driver can consider the one with the later deadline as *overridden* by the one with the earlier deadline. This is regardless of the order in which these two requests arrive. Reusing buffers is now trivial for MMFS, and the message overhead for buffer reuse is kept to a minimum.

Despite this request overriding system, explicit cancellation of block requests is still needed. When a stream is closing down on request of an application, all requests belonging to that stream must necessarily be fulfilled or cancelled before the driver can be told that the stream is deleted. If not, a new stream could be admitted in its place while requests for the closing stream are served in the same round, and that could lead to potential per-round overutilization. For this reason, the FS/driver protocol must support explicit request cancellation as well.

### *5.1.12. Rejected alternatives*

Finally, we will discuss some rejected alternative divisions of tasks. In particular, one approach that was also considered, involves a single MMFS process that repeatedly issues batches of multimedia requests to the driver. It is then the MMFS process that uses EDF to determine which requests are included in the next round. MMFS sends off the whole round to the driver at once, waits for all requests' replies (or perhaps one reply batch) to come in, and then determines and submits the contents of the next round. The result is shown in figure 5.12.

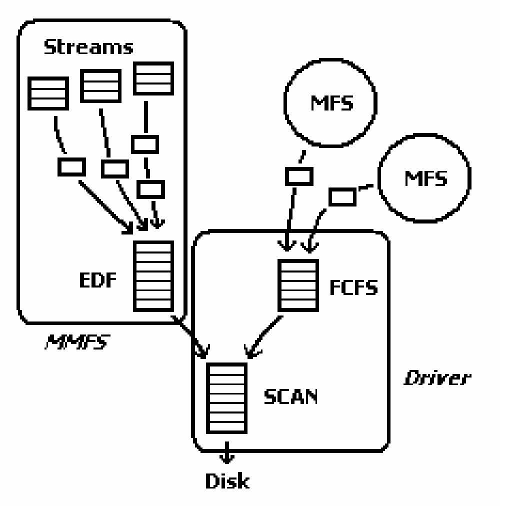


*Figure 5.12: alternative division of tasks*

This simplifies block buffer management issues, because MMFS is responsible for both the buffers and the deadline management. Issues like buffer reuse and stream cancellation become nonexistent. On the other hand, MMFS then determines the duration of each round. It must subsequently be in charge of defining which portion of the maximum round duration can be used by nonmultimedia requests, and perform admission control as well. This means it has to be aware of disk metrics – at the very least, it has to communicate about them with the driver.

Having a file server perform many tasks that are typically made by the disk driver, is unnatural to say the least. MMFS would essentially impose its model onto the disk driver, making the whole approach rather "bolted down" and inflexible. Also, having more than one multimedia-aware file server mounted at the same time would become impossible.

This model can be extended to support multiple file servers, by moving admission control back into the driver and letting each MMFS instance allocate a fraction of the (now again driver-controlled) rounds based on the streams the file server has. MMFS could increase and decrease its own fraction on demand, and submit a batch of requests of up to this fraction for each round.

There is however another issue that both these models suffer from: there is a communication gap between the last reply of the last round and the submission of the next round's batch. In the case of multiple MMFS processes, the driver would have to synchronize with all MMFS processes before it can start a new round. Otherwise, the round batch of one MMFS process could arrive after the start of the new round, and possibly cause it to miss its deadlines. The time needed for synchronization could lead to the disk going idle even in the case that requests are present. This makes this model *non-work-conserving*, as it does not use the disk to the fullest extent.

In addition, the above approach implies that activity in all MMFS processes and the driver now happen at roughly the same time, forcing the computational overhead to happen at peak moments rather than being more evenly spread out over time as in our model. Ultimately our current approach was preferred because in addition to all the above points, it inherently supports multiple MMFS instances rather than requiring explicit support for this.

As a completely different alternative, block buffers could theoretically be placed at the disk driver. MMFS processes would then notify the driver of requests and get a block buffer address in the disk driver’s address space containing the data as reply. Especially with a LRU cache replacement policy, a centralized cache could have advantages over several local caches in MFS processes. However, the file servers would have to communicate their caching policies to the disk driver if anything other than LRU is to be used. Also, given the potentially huge differences in block sizes between MFS and MMFS, this would make the choice of cache buffer size rather difficult. Finally, the file servers would have to let the disk driver know when they are done with a specific buffer, adding an extra layer of communication.

## 5.2. The MMFS – driver protocol

The concepts from section 5.1 are now implemented in an actual protocol extension between MMFS and the driver. Figure 5.11 shows the relevant extract from figure 5.3. The protocol revolves around the block prefetch requests from MMFS for each stream, linking MMFS streams to the driver’s EDF queue. Besides basic block request messages, the protocol must offer a way to override and cancel block requests (see subsection 5.1.11). For admission control, there must be stream addition and deletion messages. All requests are made by MMFS; all replies come from the driver.

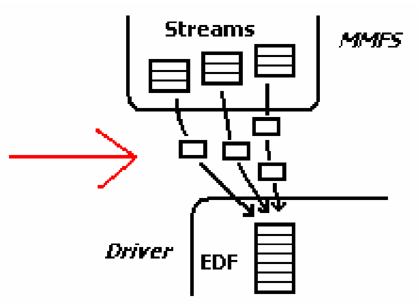


*Figure 5.11: the protocol between MMFS and the driver*

Subsections 5.2.1 and 5.2.2 describe the resulting protocol extension. Subsections 5.2.3 and 5.2.4 discuss deadlines and admission control, adding the practical aspect of MINIX’ clock frequency. Subsection 5.2.5 summarizes the model’s requirements into three basic protocol rules. If MMFS follows those rules, the driver can meet the deadlines of all block requests.

### *5.2.1. Extension of existing protocol*

The format of the data retrieval request and reply messages are fully based on the original basic driver read and write requests (DEV_READ_S and DEV_WRITE_S) and replies. The driver can then process multimedia requests as normal requests.

However, the original protocol assumes that there is always at most one best-effort request pending at the driver, and this is not the case with MMFS. To allow multiple pending requests and replies between a single MMFS instance and the driver, the new asynchronous communication primitive is used in both directions here too. Again, this implies a non-FIFO ordering of messages.

We then need to extend the basic messages with at least two things:

- The multimedia deadline of each request, so that the driver can perform EDF sorting.
- Some way to tell which replies and cancel requests belong to which requests.

To indicate that a deadline has been specified at all, we add the DEV_MM_READ_S and DEV_MM_WRITE_S message types for data transfer. The only difference between DEV_{READ|WRITE}_S and DEV_MM_{READ|WRITE}_S *requests* is the deadline in the request message. The difference between their *reply* messages is somewhat more elaborate.

New messages are introduced for request cancellation (DEV_MM_CANCEL_S), and for admission control (DEV_MM_ADD and DEV_MM_DEL), using a new field layout specific to the requests. All reply messages share a field that contains the original request type. The other fields that were added to the reply message are specific to the request type. The full message layout of all added requests and replies is listed in appendix C. The next three subsections will discuss several aspects in detail.

To retain the spirit of the original protocol and retain a “balanced” communication scheme, every request is always eventually replied to. Specifically, an overridden or cancelled request always results in an error, rather than being discarded silently.

The requests are not tied to streams directly. The driver can only link a request to a specific MMFS instance, not to a specific stream. The implications of this are discussed later.

### *5.2.2. Request identification and cancellation*

As it turns out, a system with request IDs as implemented for the new VFS/FS protocol (see subsection 3.4.4) is not needed, as there is already enough information present in each request for unique identification. Each request carries an I/O destination endpoint (the IO_ENDPT request message field) and an endpoint-dependent indicator that can uniquely identify the destination buffer in the form of an *I/O grant* (the IO_GRANT request message field). I/O grants are part of MINIX 3’s improved interprocess protection. One process tells the kernel to grant another process access to a specific buffer with a specific length, and hands over the resulting grant identifier to the other process. The other process is then limited to accessing only that buffer in the first process’s address space. In general, messages that use grants have a ‘_S’ suffix to their name.

As long as the MMFS process makes sure that there is a 1-1 mapping between I/O grants and buffers (which it can, by preallocating all grants), the combination of the I/O endpoint and I/O grant are sufficient to identify a specific *buffer* of a MMFS process. The combination of those two fields and the request’s deadline (the MM_DEADLINE field) are sufficient to identify a specific *request*.

It is now trivial for MMFS to override or cancel previous requests. MMFS can override a previous request with a new request for the same buffer but with a lower deadline, by sending off the new request with the same I/O endpoint and grant and the new lower deadline. A specific request can be cancelled by sending a DEV_MM_CANCEL_S request that carries the I/O endpoint, grant, and deadline of the request to cancel.

### *5.2.3. Request deadlines*

Deadlines are necessarily expressed in clock ticks – the smallest time unit available on MINIX 3. The only field that is still unused in the original DEV_{READ|WRITE}_S messages is the m2_s1 short integer field. This is not large enough to store clock values, which require a long integer field. Therefore, the deadline has to be fit into a short integer value. In practice this means going from 32-bit to 16-bit on 32-bit architectures.

It is attractive to use deadlines relative to the current time in this case, but this is risky. Even if passing a message always takes less than a clock tick in practice, exchange of a message may just happen to take place across a clock tick. The receiver would then decide on a different resulting clock value than the sender. This would make it impossible to use the deadline as part of the unique identifier for requests.

To avoid this, absolute deadlines are used. These have to be truncated; it is up to the receiver to determine the original deadline from the truncated value. To facilitate this, deadlines used in messages may not be further in the future than can be expressed by a *signed* short integer (15-bit, i.e. 32767). Combined with the fact that deadlines always have to be in the future at all (by at least the work-ahead time), the receiver can easily determine the original absolute deadline using the following C code. It takes the 32-bit current time as ‘cur_time’, and the 16-bit ‘deadline’ (MM_DEADLINE) value from the request, and returns the resulting 32-bit deadline:

```
if ((cur_time & 0xffff0000) + deadline >= cur_time) {
      return (cur_time + deadline);
} else {
      return (cur_time + deadline + 0x00010000);
}
```

Note that even 32-bit deadlines are not really absolute: with MINIX' standard clock frequency of 60 ticks per second, it takes about two years and three months overflow the counter. Our MMFS and driver implementations deal with this by subtracting 32-bit clock values for relative comparisons.

## *5.2.4. Admission control and timing restrictions*

Before a MMFS instance may start a stream, it has to send an admission request to the driver, in the form of a DEV_MM_ADD request. The request includes the stream rate in bytes per second (MM_RATE) and the block size in bytes (MM_BLOCK_SIZE). If the driver cannot admit the stream, it must send back an error code indicating the stream has not been admitted. If the stream *can* be admitted, then the driver must send back the work-ahead time and the minimum time distance between each pair of requests (see subsections 5.1.5 and 5.1.6).

These 'work-ahead time' (REP_MM_WAT) and 'ticks per block' (REP_MM_TICKS) values must be expressed in clock ticks as well. After all, timers can only be set on clock tick granularity. The 'ticks per block' value is the minimum number of *clock ticks* between each two request deadlines. To guarantee the admitted stream rate and to make sure no more than the admitted number of block requests have their deadline fall in one round, this value must necessarily be a clean divisor of the number of clock ticks per round.

Based on the discussion in subsections 5.1.6 and 5.1.7, this results in the following pseudocode logic to determine the effective stream rate and return the appropriate 'ticks per block' value along with the work-ahead time:

```
on receipt of DEV_MM_ADD(rate, block_size) {
     /* convert rate from bytes per second, rounding up
      * to a multiple of the given block size */
     blocks_per_second = divide_rounding_up(rate, block_size);

     /* convert the number of blocks per second to a fraction
      * of a round */
     blocks_per_round = blocks_per_second / TICKS_PER_ROUND;
```

```
        /* increase this fraction to the lowest higher
         * integer divisor of the number of ticks per round */
        blocks_per_round =
        round_up_to_divisor_of(blocks_per_round, TICKS_PER_ROUND);

        /* note that ((blocks_per_round * block_size) >= rate)
         * still holds. however, blocks_per_round is now an
         * integer divisor of TICKS_PER_ROUND, so we can divide
         * cleanly. the result is the number of ticks allowed
         * between each two block requests. */
        ticks_per_block = TICKS_PER_ROUND / blocks_per_round;

        /* do the actual admission control test, and return
         * the appropriate values if admitted */
        if admission_control_test(ticks_per_block, block_size)
                   return OK, ticks_per_block, work_ahead_time;
        else
                   return error code;
    }
```

This implies a whole lot of rounding up, although all of the rounding granularities are no more than predetermined configuration values (i.e. #define's): the MMFS block size, the round duration, and the clock frequency. The standard MINIX clock frequency of 60 ticks per second will already have to be changed if applications are to request frames at a rate of for example 25 frames per second. This is however not an issue for the driver or even MMFS, and we will look at this again in chapter 8.

Complementing the DEV_MM_ADD request, there is a DEV_MM_DEL request to tell the driver that a stream does not exist any more. These two admission control requests include a opaque "stream ID" field, allowing the driver to link delete requests to add requests. The stream ID is also echoed in both reply messages, allowing MMFS to link add/delete replies to requests. Actual stream requests are not linked to streams, so the file server may not issue the DEV_MM_DEL request until all of the stream's block requests have been fulfilled or cancelled.

### *5.2.5. Resulting contract with file servers*

We now summarize the requirements for actual block requests, to which MMFS instances must adhere, so that the driver can meet all deadlines (following subsection 5.1.7). These requirements make up the basic contract between the FS and the driver. For each stream, that is, each '*<block size*, *ticks per block>*' pair that the driver has admitted with a DEV_MM_ADD reply (and not later removed because of a corresponding DEV_MM_DEL request), the following must hold for all block requests made as part of that stream:

1. All request deadlines must be in the future by at least the number of ticks given by the driver as the work-ahead time. That is, two rounds plus a small epsilon value.
2. For each pair of requests, the deadlines of these requests (each of size 'block size') must be least 'ticks per block' clock ticks apart.
3. No request must be made with a deadline that is ahead of the current time by more ticks than the upper value of a signed short integer.

Requests that violate the first point are not *feasible*, as per subsection 5.1.5. Pairs of requests that violate the second point may cause per-round overutilization, as per subsection 5.1.6. Requests that violate the third point may lead to deadlines being extended incorrectly, as per subsection 5.2.3.

Stream requests are not linked to streams in the protocol, so the driver cannot enforce this contract for individual streams. As a result, one actively misbehaving MMFS instance could cause deadlines to be missed for *all* MMFS instances, not just for itself. One partial solution would be to let the driver maintain a per-FS, rather than a global, count of admitted stream bandwidth per round. It can then make sure that no more than the appropriate fraction of the (maximum) round time is dedicated to each of the file servers. This extra safeguard was not implemented in the driver for this project.

## 5.3. Driver implementation

In MINIX 3, the driver-independent aspects of drivers are implemented in a "libdriver" library. That cleanly separates the FS/driver communication protocol from the actual driver implementation. For this project, all changes made on the driver side are confined to “libdriver”. The changes therefore automatically work for all hard disk drivers, even though only the “at_wini” IDE driver is relevant to us in practice.

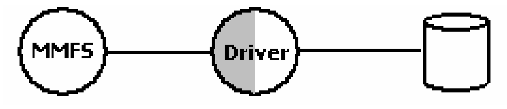


*Figure 5.12: the driver process*

For the most part, the driver implementation simply follows the protocol specification in the most straightforward way, but there are some aspects that are worth discussing. Subsection 5.3.1 describes the deterministic admission controller that we have implemented. Subsection 5.3.2 discusses the per-round combination of multimedia and nonmultimedia requests. Subsection 5.3.3 describes how rounds are implemented in a way that yields various advantages. Finally, subsection 5.3.4 describes the resulting logic that the driver uses to handle the EDF multimedia queue, the FCFS nonmultimedia queue, and the SCAN round queue.

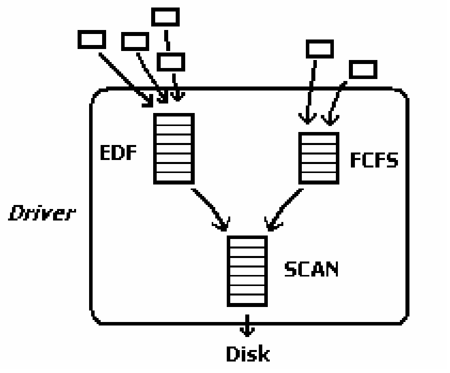


*Figure 5.13: the new driver archtecture*

## *5.3.1. Disk metrics and admission control*

The new multimedia-aware part of the driver library has been implemented with high modularity, in order to allow a more elaborate admission control and round management system to be dropped in with relative ease. However, the admission control system that we implemented is deterministic, and as such rather basic. The driver is assumed to know the worst-case maximum number of blocks (with a certain size) that the driver can fetch from the disk during a round, and it uses this value to determine how many streams can be admitted with which bandwidth. This rather simple approach requires the disk driver to pick a static block size to base these measurements on, rounding up and taking multiples of it as necessary.

In order to allow nonmultimedia requests to be processed in each round as well (see the next subsection), a static number of blocks per round is reserved for nonmultimedia requests, and admission control only allows streams to use the remaining number of blocks per round. A nonzero value ensures that nonmultimedia requests will never starve.

The actual disk speed parameters are computed “offline”, and currently statically defined at compilation time. With some changes, these values could be specified as environment parameters at boot time or even obtained statistically at load time, but that was not necessary for this project.

### *5.3.2. Combining multimedia and nonmultimedia requests*

If at least one nonmultimedia request is allowed per round, then the driver must know how to optimally merge multimedia and nonmultimedia requests into one single round. For this project, one important consideration is that the current implementation of the MINIX file server (MFS) – which was until recently fully integrated with the blocking VFS server – is also implemented as blocking. Each instance of MFS can make at most one request to the disk drive at a time, and waits until the disk driver has replied. Subsequently, the maximum number of nonmultimedia requests (all from MFS instances) that can arrive before the start of each disk scheduling round, is equal to the number of mounted MINIX partitions on the system. This number is typically very low: the standard MINIX 3 installation suggests three partitions ("/", "/usr" and "/home"), so in the worst case, three nonmultimedia requests are submitted to the disk driver for each round.

A second consideration is that partitions for multimedia and nonmultimedia file systems are completely separate. Each partition is laid out contigously on disk, so the blocks requested by MFS instance will be placed physically relatively distantly from blocks requested by MMFS. This implies that little advantage can be obtained from integrating multimedia and nonmultimedia requests in a way that minimizes disk arm movement. This is a significant difference from systems that integrate multimedia and nonmultimedia files on the same file system.

For these reasons it was decided to always handle the nonmultimedia requests at the beginning of the round, and then handle the multimedia requests. This ensures that while the multimedia requests will meet their deadlines in any case, pending nonmultimedia requests will be served as quickly as possible per round.

In practice, the sets of multimedia and nonmultimedia requests are both sorted in CSCAN (i.e. one-way SCAN) order, so that the resulting combination is CSCAN-ordered as well. Depending on the relative layout of the multimedia and nonmultimedia partitions, the complete CSCAN sorting may be over two rounds rather than within a single round.

### *5.3.3. Dynamic round times*

Like for example the MARS project [4], our implementation uses *dynamic round times*. Instead of using a periodic timer to start and end each round, rounds are started whenever multimedia requests are present while no round is active, and they end whenever the last request of the round has been processed. Rounds then only have an upper duration. Requests that arrive during a round, do not necessarily have to be queued for the duration of a full round before they are processed.

This approach is inherently *work-conserving*: the disk is not idle for the rest of a round if there is any work to do, and no special measures have to be taken to fall back on the basic FCFS policy for nonmultimedia requests if no multimedia requests are present.

Another advantage is that round times will be relatively short if relatively few multimedia streams are present. This means that nonmultimedia requests will automatically be served at a relatively lower latency, because the rounds are shorter. They will also be served at a faster rate, because there are relatively more rounds overall.

### *5.3.4. Actual implementation*

The sets of requests for the current round are kept in arrays to allow for easy sorting. Nonmultimedia I/O requests, as well as non-I/O requests, are kept in simple linked lists of structures that only contain messages (FCFS). Multimedia I/O requests are kept in a separate (doubly-linked) linked list that is sorted on (expanded) deadlines, lowest first (EDF). Whenever a new multimedia I/O request comes in, this list is iterated over, for two reasons:

- If another request with the same I/O endpoint and grant is found, then the request with the relatively highest deadline is considered to be overridden and removed from (or not added to) the list (and replied to with an ‘EINTR’ error code).
- The new request (if not overridden) is added at the right location into the list so that the list remains EDF-sorted.

Although this could be optimized by splitting these out into a hashtable and a list that is sorted on demand, the number of requests in the queue is in practice low enough not to substantially benefit from this.

A scheme similar to the first step is used when a cancel request (DEV_MM_CANCEL) from MMFS is processed. First the queue is examined. If a matching request is there, it is cancelled (i.e. removed and replied to with ‘EINTR’) and the cancel operation finishes successfully (‘OK’). If no matching request is present in the queue, the remaining multimedia part of the current round is examined, and an appropriate error code is returned based on whether the request-to-cancel was found in the current round (‘EBUSY’) or not (‘ENOENT’).

Every time the actual driver is ready to perform new work, either at startup or because it has finished processing the last request, the driver library takes the following steps:
- If a round is currently in progress and not yet finished, process the next request in the round.
- Otherwise, if the multimedia I/O queue is not empty, start a new round, and process the first request in this new round.
- Otherwise, if there are nonmultimedia I/O requests or non-I/O requests queued, dequeue and process the next one.
- Otherwise, receive a message, as there is no work queued.

The easiest way to model the receipt of a message is that the message is simpled queued in the appropriate queue upon receipt, after which the steps above are taken again if the driver is idle (i.e., not currently processing a request). In practice, if the driver is idle, nonmultimedia requests are not queued and instead processed directly upon receipt. If the driver is not idle (typically waiting for DMA transfer to complete, receiving messages until an interrupt notification comes in), messages are always queued.

When a new round is started, up to as many requests are taken from the head of the EDF-sorted multimedia I/O queue as allowed by the defined fraction of the defined per-round maximum. After that, the rest of the round is filled with nonmultimedia I/O requests. Both parts are stored in arrays, sorted on disk position (CSCAN), and the first request is processed. As indicated in subsection 5.3.2, requests are first taken from the nonmultimedia I/O queue, and then from the multimedia I/O queue.

Note that the driver will never go idle as long as any message is queued, so this entire approach is indeed *work-conserving* and implies the aforementioned dynamic round times. No timers are needed.

# Chapter 6. MMFS part 2: the driver side

This chapter describes the implementation of the second and last part of the MMFS multimedia file server. This part manages the data streams, block buffers, cache, and the communication to the driver using the new protocol extensions described in chapter 5.

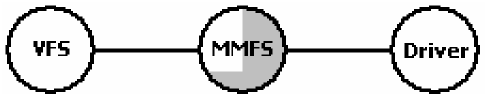


*Figure 6.1: the MMFS process, part 2*

Section 6.1 discusses the main objects in MMFS: buffers and streams. It describes the mapping between these two objects, and the procedures used to move streams forward. On top of that, section 6.2 outlines the policies of deadline and request management, and shows how MMFS fulfills the FS/driver protocol contract from section 5.2.

A stream is always opened, read from, and closed by a single user process. This is the subject of section 6.3, which discusses the user process API, and with that the MMFS implementation of the VFS requests not yet discussed in chapter 4. Section 6.4 makes some final notes.

## 6.1. Buffers and streams

MMFS streams are a direct implementation of the stream concept from subsection 5.1.1. Each active stream has a head, a split, and a tail. All data between the head and the split has been requested from the disk driver but not yet arrived. All data between the split and the tail of the stream has arrived and is available for reading by the user process.

MMFS uses a static block size. To store prefetched data in, it maintains a pool of buffers which are all of that block size. Buffers are dynamically assigned to blocks as needed. The window of each stream consists of a number of blocks, and therefore of a number of buffers. The head, split and tail of a stream move forward by whole blocks at a time, regardless of actual frame sizes.

Subsection 6.1.1 and 6.1.2 elaborate on the basic principles by defining the buffer object and stream windows. Subsection 6.1.3 to 6.1.6 describe the mechanisms employed to advance the head, split, and tail of a stream window. Subsection 6.1.7 describes MMFS' caching policy.

### *6.1.1 The buffer object*

Each buffer can be dynamically assigned to an arbitrary block on disk. This block is identified by a *block position number*. The block start (in bytes) into the partition is always a multiple of the static block size. Several streams may *overlap*, that is, cover (in part) the same section of a single file. For optimal buffer usage, only a single buffer can be assigned to a single block position at any time. A single buffer may therefore be part of the window of more than one stream. Buffers can be found by block position number using a position-based hashtable.

A buffer is *in use* by a stream if it is currently part of the window of that stream. It is then assigned to a block position number that falls between the head and tail of the stream. A buffer is *filled* if the block data for the buffer's block position has been retrieved from disk. A buffer is always in one of the following five states:

- **Free**. The initial state of all buffers. The buffer is not assigned to a block position and not in use by any stream. All buffers in this state are part of the *free list*.
- **Requested**. The buffer has been assigned to a block position, but is not yet filled. A (DEV_MM_READ) request for it is pending at the driver. The buffer is in use by one or more streams.
- **Prefetched**. The buffer has been assigned to a block position, and has been filled by the driver. The buffer is in use by one or more streams.
- **Cached**. The buffer is assigned to a block position and is filled, but it is not in use by any stream any more. It is therefore in the *cache*. MMFS' caching policy is described in subsection 6.1.7.
- **Cancelling**. In section 6.3 we show that this state is needed for buffers in use by closing streams. It indicates that the block was in *requested* state, and a DEV_MM_CANCEL message has been sent to the driver to cancel the DEV_MM_READ block request.

Figure 6.2 shows the state transitions. The remaining part of the chapter uses the transition numbers in the figure to indicate when certain transitions take place.

A buffer that is in use by one or more streams (i.e. in the state *requested*, *filled* or *cancelling*), always has an associated deadline. This is always the earliest stream deadline for the corresponding block. If the buffer is in state *requested*, then the latest request pending at the driver to fill this buffer always has that earliest deadline. Deadline computation will be covered in section 6.2.

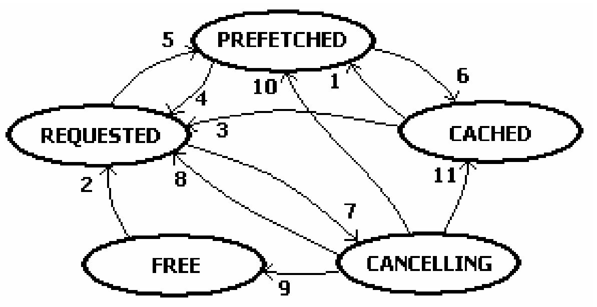


*Figure 6.2: buffer state transitions*

A buffer that is in use, also has a stream as *owner*. The buffer’s owner is always the stream that has the earliest deadline for the buffer’s block. If there are multiple streams with the same earliest deadline for this block, the buffer’s owner is any one of those.

It is important to note that several streams may operate on a single file with different stream rates, so no assumptions can be made about the ordering of buffer ownership between streams. That is, a buffer’s deadline and ownership may have to be recomputed whenever a stream advances its head or tail.

As explained in subsection 5.2.2, each buffer has an I/O grant associated with it. MMFS allocates these grants on startup. The FS/driver protocol message replies supply the grant as identifier for the buffer that has been filled or cancelled. For fast lookup, MMFS uses a grant-based hashtable.

Table 6.1 shows the ‘struct buffer’ structure fields.

| **Field** | **Description** |
|---|---|
| int b_state; | One of the five states *free, requested, prefetched, cached, cancelling*. |
| bpos_t b_pos; | Block position number currently associated with this buffer. |

| | |
|---|---|
| clock_t b_deadline; | The earliest deadline of all streams with this block in their window. |
| struct stream *b_owner; | The stream that needs the associated block the soonest of all streams. |
| char *b_data; | Pointer to the actual buffer data area in MMFS. Only set once. |
| cp_grant_id_t b_grant; | Grant to the driver associated with the data. Only set once per driver. |
| struct bufferlist *b_list; | Linked list pointers (prev, next) for the free list and the cache. |
| struct bufferlist *b_plist; | Linked list pointers (prev, next) for the position-based hashtable. |
| struct bufferlist *b_glist; | Linked list pointers (prev, next) for the grant-based hashtable. |

*Table 6.1: the buffer structure*

MMFS must have enough buffers to support the required number of streams. The memory consumption of MMFS dominated by the product of the block size and the number of buffers in the buffer pool. A bigger block size therefore leads to higher independence from frame size variations (see chapter 5) and faster data retrieval overall (see chapter 8) at the cost of a higher minimum stream rate (chapter 5) and more MMFS memory consumption.

### *6.1.2. Stream windows*

The head, split and tail of a stream's window are expressed in block position numbers. The file system design guarantees that all files are laid out consecutively on disk, so each next block position number can be found by adding one to the last block position number. Figure 6.3 shows this. Note that there is no real list or array of buffers for each stream; rather, all buffers are obtained from the position-based buffer hashtable based on the stream's head/split/tail block numbers.

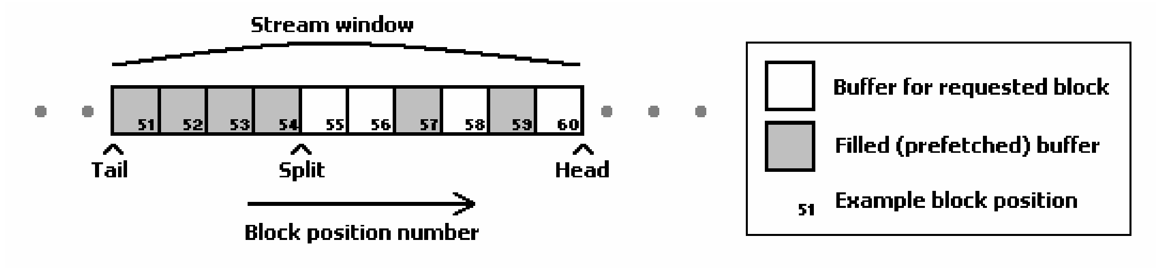


*Figure 6.3: a typical stream window*

The tail block is the first block that the user process has not (fully) read yet, and therefore the first block that needs to be available to the user process. The split block is the first block that has not yet been filled. The head block is the first block that is not part of the stream's window any more. If the tail block equals the split block, the stream has no prefetched data. If the split block equals the head block, the stream has no data requests outstanding to the driver.

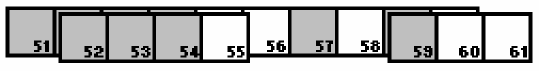


*Figure 6.4: buffer sharing of three streams*

One very important invariant is that a buffer has been assigned to each block in between the tail and the head. Obtaining a buffer based on a block position that is part of *any* stream's window, will therefore always succeed.

All buffers associated to blocks between the tail and the split are in state *prefetched* (with the exception of *cancelled* buffers, see section 6.3). Typically, the buffers between the split and head are all in state *requested*, but this part of the window may contain *prefetched* buffers as well, for one of the following reasons:

- The buffer was already prefetched for another stream that has this block in its window.
- The buffer was obtained from the cache.
- The corresponding block request was fulfilled relatively early by the driver.

In principle, the head, split, and tail block numbers of a stream will only increase. However, buffer stealing may cause a stream's head and split to be decreased. The relevant procedures are described in the next four subsections.

### 6.1.3. Advancing the window head

The procedure for advancing the window head is not as straightforward as sending a request to the driver. A buffer structure must be available to store the resulting data in, but it may turn out that no buffer can be obtained for this request. It is also possible that a buffer has been prefetched or requested for this block position already. This subsection and the next describe the exact procedure taken by MMFS.

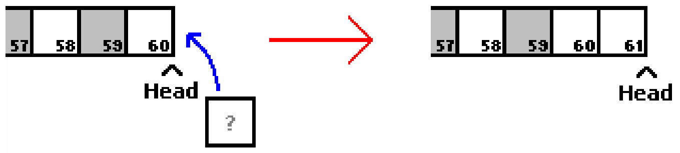


*Figure 6.5: advancing the window head*

Each attempt to advance a stream's head by a single block involves the desired block position number and a newly computed deadline for that block for this stream. Computation of the deadline is subject of section 6.2.

First, the position-based hashtable of buffers is checked. If a buffer structure is already present for this block number, the action taken depends on the state the buffer was in:

- If the buffer was *cached*, it is removed from the cache and marked as *prefeteched* (buffer state transition 1). It adopts the requesting stream as owner and takes on the newly computed deadline.
- If the buffer was *requested* or *prefetched*, the buffer's deadline is compared to the newly computed deadline. If the latter is earlier, the buffer changes owners to this stream and adopts this new deadline. In that case, if the buffer was in *requested* state, a DEV_MM_READ request for the block data is resent to the driver, but with the new, earlier deadline. This will *override* the old request at the driver, so that the data is retrieved sooner.

Nothing else needs to be done in this case.

On the other hand, if no buffer is present for this block position number yet, a buffer has to be obtained from elsewhere. If the free list is not empty, a buffer is taken from there (buffer state transition 2). Otherwise, if the cache is not empty, a buffer is obtained from there (transition 3). If both lists are empty, a buffer has to be stolen from another stream. The procedure for stealing buffers is described in the next subsection. This might not succeed, but if it does succeed, the resulting buffer was previously in state *requested* or *prefetched*. In the latter case, buffer state transition 4 takes place.

When a buffer is found for the free list or cache, or stolen successfully, then it is given the new block position, (re)inserted into the position-based hashtable, marked as owned by the requesting stream with the requested deadline, and put in *requested* state. A DEV_MM_READ request for it is sent to the driver, which possibly overrides a previous request for this buffer.

### *6.1.4. Stealing buffers*

When stealing a buffer, MMFS always has to find the “worst” buffer currently in use, that is, the buffer that has the latest deadline of all buffers in use by any stream. In no case must a buffer be stolen from a stream that needs the buffer more (i.e., that has an earlier deadline for it). This guarantees that with a limited number of buffers available, the number of buffers in use by each stream is automatically *fair* and in proportion to its stream rate. Or, to put it in the terms from subsection 5.1.7, with 4S buffers available in total, each stream gets its minimum of 4B buffers at all times. See also the discussion in subsection 5.1.11.

The worst buffer of all streams is found by looping over all the streams, and going from head to tail over each stream’s window, looking for the last buffer in the stream’s window that is owned by that stream. This buffer will have the latest deadline of all buffers owned by that stream, provided the stream owns any buffers at all. If there is no overlap in stream windows, the buffer will always be the last buffer in the window, that is, the first buffer that is checked for that stream.

If an owned buffer is found for the stream this way, its deadline is compared to the deadline of the worst buffer found so far (if any), and if this buffer’s deadline is further in the future, it will be the new worst buffer. At the end of the loop over all the streams, the result is the worst buffer overall. If this buffer has a deadline earlier than the deadline of the request for which the buffer is to be reused, then the buffer may not be stolen, and in that case the stealing stream can not advance its window head at that time.

If a buffer is actually stolen (i.e., the worst buffer has a later deadline than the new request’s deadline), this buffer is going to be used for a new block position number, so it is no longer available to *any* stream with this buffer’s old block position in its window. This includes, but may not be limited to, the stream from which it is stolen. Therefore, whenever a buffer is stolen, all streams are checked and their window head and split are reduced if necessary, in order not to include this buffer. The choice for the overall worst buffer implies that none of the streams of which the window is reduced, hold ownership of blocks in the part of the their window that is clipped off. Other buffers in this part therefore need not be checked.

### *6.1.5. Advancing the window split*

Whenever the driver notifies MMFS that a block has been retrieved from disk (with a DEV_MM_READ reply), MMFS looks up the corresponding buffer. If the buffer state is still *requested*, it is changed to *prefetched* (buffer state transition 5). All streams are checked to see if this buffer was the first buffer after the split. If so, the split block position is advanced to the next buffer in state *requested*, up to the head of the window. If enough data is now available to fulfill a suspended read() request from the user application, the read() call is resumed.

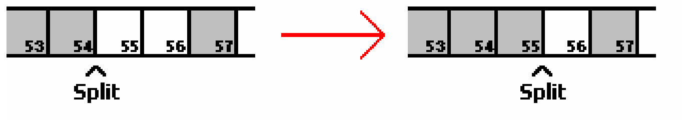


*Figure 6.6: advancing the window split*

*6.1.6. Advancing the window tail*

A stream's tail is moved up whenever the stream's reader position is at least one whole block ahead of the current tail block's base position. This means that one or more buffers cease being part of this stream's window. These buffers are always in the state *prefetched*.

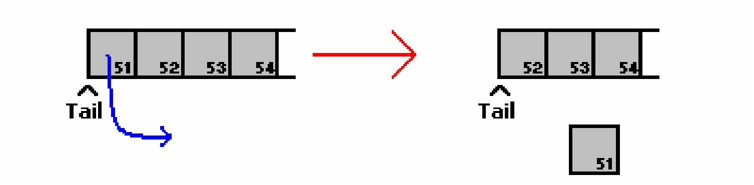


*Figure 6.7: advancing the window tail*

Of those buffers, the buffers that have this stream as owner will have to change ownership as a result. A new owner is searched for by looping over all streams, checking whether any stream has the buffer in its window, and if so, which stream now has the earliest deadline for this buffer. If such a stream is found, the buffer changes ownership to that stream, with that deadline. If no such stream could be found, the buffer is put into the cache (transition 6).

*6.1.7. The buffer cache*

Buffers that are filled but not in use are in the cache. MMFS implements a distance-based caching policy [22], organizing the cache in such a way that the buffers of which the contents have the lowest probability of being reused soon, will be the first to be reused for different content (i.e., for another block position).

If a buffer is added to the cache, it is added because no new owner could be found for the block. That is, no stream turned out to have this block in its window. That means that before a buffer is added to the cache, all streams are checked first. During this check, a hypothetical deadline distance to the current time is computed for the buffer's block position for each stream, even though the buffer is not part of that stream's window. If the block is ahead of the stream's tail, the distance can be computed based on the block's deadline *as if* the block were part of the stream window. If the block is behind the stream's tail, a default maximum upper value is used.

The lowest deadline distance of all streams forms an estimate of how soon this buffer will be reused. The buffer is handed over to the cache along with this distance. The buffers in the cache are stored in a doubly-linked list that is sorted on these distances, highest distance first. Buffers are taken from the front of this list for reuse. To reduce computational overhead, the deadline distances are computed only once.

## 6.2. Deadline and request management

The following subsections discuss issues related to timing: subsection 6.2.1 discusses how streams' block deadlines are computed, subsection 6.2.2 outlines how MMFS determines when to issue the next request for each stream, and subsection 6.2.3 discusses what happens when a stream reader is too fast or too slow.

### *6.2.1. Deadlines*

All block deadlines for a stream are based on the stream's "base time," which is the deadline (i.e., expected reader retrieval time) of the tail block of the stream's window. These deadlines play a role not only while making requests, but also to determine the importance of the current use of a buffer (see subsection 6.1.4 on block stealing).

A block buffer in the stream window that is N blocks ahead of the stream tail has a deadline that is simply equal to "*base time + N * ticks per block*". Remember that the 'ticks per block' value is specified by the driver after admission control. When sending block requests, the resulting deadline is increased to be ahead of the current time by at least the work-ahead time. This fulfills the first point of the FS/driver contract. However, having to do so indicates a potential deadline miss, and should not occur if everything goes well.

In the normal case where deadlines are not missed, the deadlines of all requests are already predetermined by the values returned by the disk driver after admission control according to the formula above. All MMFS does is manage the number of outstanding requests to the driver.

When advancing a stream’s tail, the stream’s base time is increased by the number of tail buffers removed, times ‘ticks per block’, to reflect the new tail buffer’s deadline. By only ever increasing the base time, by increasing it only by a multiple of ‘ticks per block’ whenever the stream advances its tail, and by only issuing requests when advancing the head of the stream, MMFS is able to guarantee the second point of the FS/driver contract.

### *6.2.2. Issuing of requests*

In principle, MMFS follows the effective stream rate as specified by the driver, advancing the stream head once every ‘ticks per block’ clock ticks. The time between issued requests is then equal to the time between these requests’ deadlines, but the two are maintained separately. A ‘next request time’ variable is kept with each stream, and MMFS tries to advance the window head for this stream when this time is reached, increasing the ‘next request time’ variable by the ‘ticks per block’ value.

That is sufficient to keep the stream going, but it does not allow for deep prefetching (see subsection 5.1.10). To facilitate this, an attempt to advance the window head is also made whenever the reply of a previous request comes in. Requests may then be issued at a faster rate than the requests’ deadlines are apart. A check is made to only advance the window head if the resulting deadlines are not too far ahead, in accordance with the third point of the FS/driver contract.

The deep prefetching attempt fails if no buffer is available to extend the stream head. If successful, however, the ‘next request time’ variable is increased accordingly.

### *6.2.3. Stream reader rate deviation*

The deadlines of requests are not affected if the stream reader user process is reading faster from the stream than with the rate it specified. Readers can attempt to read faster than the stream rate, but even if the data is already available, this will advance the base time accordingly. Actual driver requests then still have the minimum ‘ticks per block’ time between deadlines. If the data is not available, the read request is suspended.

The stream reader can also be slower. This poses a risk if not considered. If the stream’s base time was purely based on the reader’s expected read time, and the reader was slower, then the resulting base time (i.e. the deadline of the tail block) would end up being below the current time. After all, the tail block is not retrieved by the stream reader at the intended time, so it has to be kept around. All subsequent blocks’ deadlines are based on the base time, so a stream might then be able to acquire a disproportionately large number of buffers. The buffers’ near deadlines would prevent other streams from stealing them, and those streams would then miss their deadlines as well.

To avoid this, every stream’s base time is always kept at the current time as a minimum. This pushes back the deadlines of all new requests, forcing each stream to keep its buffer usage in line with that of other streams.

This approach isolates streams from each other, and has two other advantages. First, streams are prefetching at an effective rate that is typically higher than the requested rate, due to the rounding up of driver admission control. The reader will ultimately fall behind, as it reads at its own requested rate rather than at the effective rate. The base time correction automatically compensates for this by delaying requests accordingly. And second, the approach provides a very simple implementation of stream “pause” functionality: a stream reader can simply stop reading from the file for a while, and continue at any later time at no extra cost.

## 6.3. Remaining VFS requests

This section describes the implementation of the VFS calls that directly affect streams. Subsection 6.3.1 discusses the basics of read requests. Subsections 6.3.2 to 6.3.4 describe how streams can be opened and closed, and how MMFS handles these operations. Subsection 6.3.5 then defines the stream object as used in MMFS. Subsection 6.3.6 describes how driver crash recovery is implemented.

### *6.3.1. Reading from streams*

The basic stream reading is done using simple read() calls on a file descriptor. This automatically lets VFS take care of many aspects of the reading process.

The read calls arrive as REQ_READ requests at MMFS. The procedure for handling REQ_READ requests is simple. If the requested data has been prefetched for the associated stream, the request is satisfied immediately. If not, the request is suspended, and its state is stored in the stream object until it can be satisfied. Table 6.2 shows the structure containing the state (‘struct readreq’); note that REQ_READ VFS requests do not (yet) use buffer grants.

| **Field** | **Description** |
|---|---|
| u64_t rr_pos; | Current read byte position. Always falls within the stream’s tail block. |
| size_t rr_bytes; | Number of bytes left to copy out. Result might be less at end of file. |
| size_t rr_cum_io; | Number of bytes copied out so far. |
| char *rr_ubuf; | User-space buffer address to copy data to. |
| int rr_useg; | User-space buffer segment to copy data to. |

*Table 6.2: the readreq structure*

If a suspended data request spans multiple blocks, each part is copied out whenever it is available, so that the associated buffers can be freed up by advancing the window tail as soon as possible. The actual REQ_READ reply is only sent whenever the whole request has been satisfied, unless the end of file is reached first.

### *6.3.2. The queryfs API*

MMFS allows stream readers to specify the rate at which they want to read the given stream, so the stream reader must have a way to pass this value to MMFS directly. In addition, seek and close operations on the file descriptor are not passed by VFS to MMFS in such a way that they allow MMFS to map these requests to streams, so explicit requests are required there as well. Note that due to similar lack of information in the VFS/FS protocol, only one stream is allowed per user process per file. This allows MMFS to at least reliably map read (REQ_READ) requests to streams, as these include a process identifier, unlike seek (REQ_INHIBREAD) and close (REQ_PUTNODE) requests.

MMFS uses the new queryfs() interface and REQ_QUERYFS VFS call to let processes communicate with it (see subsection 3.4.6). The definitions needed for user processes are stored in the ‘sys/streamctl.h’ header, which is included as appendix A.3. Three stream-specific calls are supported to control the stream: STREAMCTL_OPEN to open a stream, STREAMCTL_CLOSE to close a stream, and STREAMCTL_SEEK to seek within a stream.

The seek operation is currently simply equivalent to closing and then reopening the stream in terms of startup and closedown latencies. The only advantage of the seek operation is that it avoids a race condition during the independent close and open calls, where a reopening stream is denied because another stream has been admitted in its place in the meantime.

### *6.3.3. Opening streams*

The STREAMCTL_OPEN call opens a new stream for the file descriptor it operates on, with a specified bytes-per-second rate. The stream open request also includes a position, so that streams can start at arbitrary locations within the file without having to call STREAMCTL_SEEK immediately after. However, it should be mentioned that VFS maintains file descriptor positions and MMFS has no knowledge about or control over those, while VFS does not look at the stream requests, so the user process has to perform an explicit lseek() call on the file descriptor as well. This cannot be avoided without making very specific modifications to the VFS/FS protocol.

When processing a STREALCTL_OPEN command, MMFS will first send an admission request (DEV_MM_ADD) to the driver, with the standard MMFS block size and the stream rate as specified by the user process. If the admission request is denied, the user process is notified immediately, and no stream is opened.  If successful, a new stream object is created for this process, and MMFS immediately requests as many blocks from the driver as necessary to support the stream’s startup requirements (see subsection 5.1.9). MMFS sends the actual reply to the stream open request of the user process once the work-ahead time has passed, or once the replies to all startup block requests have come in, whichever happens sooner.

### *6.3.4. Closing streams*

The STREAMCTL_CLOSE call closes a stream opened previously by that process. MMFS is not informed when a process closes a file descriptor, so each process that uses a multimedia stream has to make this call explicitly as well.

If the stream has run to the end of the associated file, then no block requests will be active for this stream any more. All MMFS has to do before the stream is closed, is to send a stream delete request (DEV_MM_DEL) to the driver. If MMFS is still actively retrieving blocks for the stream when the STREAMCTL_CLOSE request arrives, the situation is more complicated. See the discussion in subsection 5.1.11. All outstanding DEV_MM_READ requests for this stream will either have to be fulfillled or cancelled before the stream can be deleted.

A complicating factor is the fact that buffers can be shared between streams. Simply cancelling all outstanding requests is not an option. There may be one or more other streams that are also interested in the same block. After cancellation, these streams may not have sufficient time available to re-request the block with a slightly later deadline (as per the first point of the FS/driver contract). Such requests must therefore not be cancelled. On the other hand, requests with deadlines further ahead *must* be cancelled, or MMFS would have to keep the closing stream around until all its deep prefetching requests have been fulfilled.

Based on this, the following procedure is followed. For all buffers in the closing stream’s window that are *owned* by the closing stream, MMFS checks which stream should own the buffer next. If the buffer was filled already (i.e., in *prefetched* state), then the buffer is reassigned to the new owner with the next earliest deadline, or to the cache if no new owner could be found (buffer state transition 6).

If the buffer was not filled, it is in the *requested* state, and a DEV_MM_READ request for it is pending at the driver. The buffer is then set to the *cancelling* state (transition 7). Buffers in *cancelling* state still keep the closing stream as their owner. However, all other streams are checked, to see if any other stream is interested in this buffer.

If another stream is found that is interested in the buffer, and that stream’s deadline for the block is nearer to the current time than the work-ahead time, no DEV_MM_CANCEL request is sent out to the driver. Eventually, a DEV_MM_READ reply will eventually come in from the driver.

In the other case, that is, no interested stream was found or the stream’s deadline is further ahead than the work-ahead time, a DEV_MM_CANCEL request is sent. Eventually, either a successful-cancellation reply or a request acknowledgement comes in, depending on whether cancellation was still possible.

A buffer can then leave the *cancelling* state one of in the following ways:

- A successful cancellation reply from the driver comes in. The next best owner is sought for the buffer (similar to the procedure outlined in subsection 6.1.6). If a new owner is found at all, the buffer’s owner and deadline are updated, and since the buffer was not filled yet, it is re-requested with the next earliest deadline (transition 8). Cancellation requests are processed nearly instantly, so the new deadline is guaranteed to be further ahead than the work-ahead time (minus up to half the epsilon part of this value, but that is no problem). The new request is therefore still feasible. If no new owner could be found, the buffer is added to the list of free buffers (transition 9).
- A data retrieval acknowledgement comes in. This implies that either no cancellation request was sent, or the cancellation failed. If there is a next best owner, the buffer is assigned to this new owner as above. Since the buffer is filled, no driver request has to be issued in this case (transition 10). If no new owner was found, the buffer is added to the cache (transition 11).

Each closing stream keeps a counter of owned buffers that are still in *cancelling* state. When this counter reaches zero, the stream has completed getting rid of its buffers, and the DEV_MM_DEL stream-delete request is sent to the driver. When the reply to that request comes in, the stream is actually deleted. All requests that are not cancelled are guaranteed to arrive within the work-ahead time, so waiting for them to come in can also never take longer than the work-ahead time either. Closing a stream may therefore take up to the work-ahead time in the worst case. The process making the STREAMCTL_CLOSE call is always replied to immediately, so this is invisible to the application. This is not the case for STREAMCTL_SEEK.

### *6.3.5. The stream object*

Based on the previous sections and subsections, we can now define the actual stream object as used in MMFS. Every stream object is in one of the following six states:

- **Free**. The stream object is unused.
- **Adding**. The stream object has just been created for a user process. A DEV_MM_ADD admission request has been sent to the driver, and MMFS is waiting for the reply. The associated user process is blocked in the STREAMCTL_OPEN queryfs request.
- **Buffering**. The admitted stream is in its startup phase. It has requested its initial blocks, and is waiting for those blocks to come in or for the work-ahead time to pass, whichever happens first. The user process is blocked in the STREAMCTL_OPEN or STREAMCTL_SEEK queryfs request.
- **Active**. The stream is active. New blocks are requested from the driver periodically, and the user process can read from the stream. A REQ_READ request may be pending.
- **Cancelling**. The stream is waiting for all its blocks to be cancelled, either before buffering (seek) or deleting (close). The user process may be blocked in the STREAMCTL_SEEK queryfs request.
- **Deleting**. MMFS has sent a DEV_MM_DEL request to the driver, and is waiting for the reply before the stream is returned to the *free* state. No user process is associated with this stream during this time.

Figure 6.8 shows the state transitions as caused by user requests.

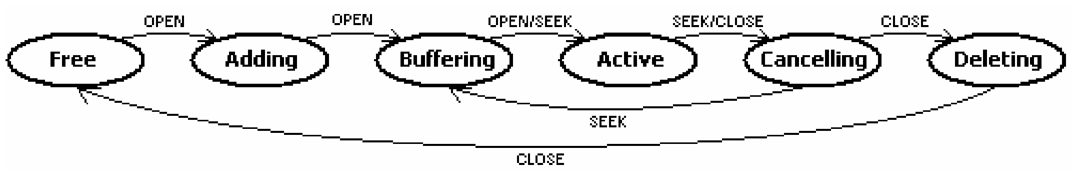


*Figure 6.8: stream state transitions*

Table 6.3 lists all fields of the 'struct stream' structure.

| **Field** | **Description** |
|---|---|
| int s_state; | One of the states *free*, *adding*, *buffering*, *active*, *cancelling*, *deleting*. |
| endpoint_t s_endpt; | Endpoint of the user process that owns this stream. |
| int s_node; | Node number of the file that this stream operates on. |
| bpos_t s_tail; | Block position number of the first block of the window. |
| bpos_t s_split; | Block position number of the first non-filled block in the window. |
| bpos_t s_head; | Block position number of the first block after the window. |
| int s_ncancel; | Number of *cancelled* buffers owned by this stream. |
| clock_t s_wat; | The ‘work-ahead time’ value for this stream, from the driver. |
| clock_t s_ticks; | The ‘ticks per block’ value for this stream, from the driver. |
| clock_t s_basetime; | The ‘base time’ value: the deadline of the tail block. |
| clock_t s_nexttime; | The ‘next request time’ value: when to advance the window head next. |
| int s_req; | Request pending from the user process: *none*, *open*, *read*, *seek*, *close*. |
| int s_id; | VFS request ID of the pending request, if any. Included in the reply. |
| struct readreq s_rr; | If pending request is *read*, state of suspended read request. |

*Table 6.3: the stream structure*

### *6.3.6. Recovery from driver crashes*

A last issue in MMFS is recovery from driver crashes, that is, handling of REQ_NEWDRIVER requests from VFS. As mentioned before, the modularity of MINIX allows file servers to fully recover from driver crashes, completely hiding such events from user processes. Once a new driver has been started to operate in the place of the old and crashed driver, VFS notifies the FS about this event. The FS can then resend any outstanding requests to the new driver, and continue as if nothing happened.

Recovery from driver crashes is not entirely compatible with MMFS’ design. The first thing to note is that if such a crash takes place, MMFS is likely to miss most if not all of its deadlines. Many of the requests that it will have sent off to the crashed driver, will not be feasible to be resent any more. In order to correct this, all streams’ deadlines have to be moved back in this case, and the deadlines of the block buffers they own have to be readjusted accordingly.

That is not all. The new driver does not actually know about the streams that the crashed driver admitted. When MMFS receives a REQ_NEWDRIVER message from VFS, it first has to resend all admission control requests to the driver – one per stream – and only then resend all block requests with their new deadlines.

There is a potential race condition here. If any MMFS instance was in the process of trying to accept a new stream while the old driver crashed, the new driver may receive various admission control requests in a different order. The driver may then accept a different set of streams. This could lead to rejection of one of the streams that were already active in MMFS. As multiple MMFS instances may be present, and as the driver is unaware of FS processes until it receives a message from them, this problem can not be solved in a way that is hidden from the stream reader user process. Therefore, no effort has been made to deal with this situation. MMFS simply assumes that resent admission requests are always granted.

## 6.4. Evaluation

Of all the code resulting from this project, MMFS has by far the most computationally expensive algorithms, warranting a note about this. Although excessive algorithmic complexity has been avoided wherever possible, MMFS has not been particularly optimized in this regard.

In particular, the worst-case complexity of buffer stealing by multiple streams in a single clock tick is $O(S^2 * B)$, where S is the number of active streams and B is the total number of buffers. This is only a theoretical upper bound, and is rarely ever even approached in practice. Our tests indicate that the practical computational overhead of the current implementation is negligible even if the number of streams is at its maximum. We therefore consider this to be acceptable.

# Chapter 7. CPU scheduling

The last point of interest on the path from the disk to the application is the aspect of CPU scheduling. Even with guarantees for retrieval at the disk level, a multimedia process could still miss its deadlines if there is no protection at the CPU level. Such misses could be caused by CPU-intensive nonmultimedia processes, but also by other multimedia processes that use up too much CPU time.

If MINIX 3 were able to provide realtime guarantees, that would solve this side of the problem altogether. However, turning MINIX 3 into a realtime operating system would be a complex and involved process, far beyond the boundaries of this project. In this chapter, we investigate what else can be done to limit multimedia deadline misses resulting from other CPU workloads.

Section 7.1 discusses a fundamental limitation with respect to scheduling system processes. Section 7.2 looks at the demands and algorithms for scheduling periodic user processes, and comes up with a matching design. Section 7.3 describes the implementation of the resulting scheduling extension.

## 7.1. Scheduling of system processes

In a microkernel operating system like MINIX, nearly all system work on behalf of a user process is performed by system processes, that is, other processes in user space. For accurate scheduling, the execution time of system processes would have to be included in the execution time of user processes. This requires that the kernel can establish which system process is working on the behalf of which user process at any time.

At least for disk I/O, this used to be possible, because of the strict chain of blocking calls as described in section 1.2. However, our VFS changes have removed the strict chaining. The result is a fundamental limitation: during the periodic execution of a multimedia user process, we can not schedule only the system processes that are working for that user process. Any solution for this would require significant changes that affect the kernel and all system processes.

With no means to link system process execution to application requirements on the CPU scheduling level, we have to fall back on the MINIX priority system for system processes.

One observation is that servers like PM and VFS (with our modifications) generally spend very little time on the calls they are processing. In the case of multimedia processes, we have already removed the obstacles that could cause a multimedia process to wait for any other process. This limits priority inversion to the point where we do not have to worry about that aspect at all.

We could therefore assign relatively high priorities to all the servers involved in supporting multimedia streams (VFS, MMFS, the disk driver, as well as PM), and assign relative lower priorities to all other servers. This is the best that can be achieved for system processes in this regard.

## 7.2. Design of a periodic scheduler for user processes

What remains is scheduling of the user processes themselves. To match our reservation-based approach to disk retrieval, we decided to add a reservation-based scheduler to the system. Subsection 7.2.1 argues that this is best done on top of the standard MINIX scheduler. Subsection 7.2.2 lists several demands for periodic multimedia processes, which result in a scheduling API described in subsection 7.2.3. Subsection 7.2.4 considers which scheduling algorithm to use. Subsection 7.2.5 discusses the implications of the limitation described in the last section, and describes the resulting scheduling system.

### *7.2.1. The MINIX priority-based scheduler*

The standard MINIX scheduler already performs well for interactive and CPU-intensive processes. It assigns a maximum priority and running time (*quantum*) to each process, and schedules the runnable process with the highest priority for the duration of its quantum. If the process uses up the full quantum, its priority is lowered. At regular intervals, the priority and quantum of all processes is increased if they are below their maximum.

The priority of CPU-intensive processes will therefore end up being lower than the priority of interactive processes with short running times, but neither will ever starve. The only difference between privileged system processes and normal user processes in this respect is that the system processes have a higher maximum priority.

There is therefore no point in reinventing a system that will achieve these properties. Instead, we can base our scheduling extension on the standard MINIX priorities, and let the MINIX scheduler do the rest. In the absence of multimedia processes, MINIX will then behave exactly as before.

### *7.2.2. User process requirements*

We assume that the a multimedia process does some computational work on the data it fetches from disk. For local video serving, this would typically involve decoding and displaying of each frame. For video-on-demand, it could involve (re)encoding of the stream and/or preparing the data to be sent over the network.

The ability to specify a period, and an execution time within this period, fulfills the basic requirements of such processes [33]. Processes then need at least two API calls: one to reserve a period and an execution time, and one to give up the remaining execution time for the current period. This allows the process to make a deterministic reservation, and yield to other processes whenever it does not need all of its reserved time in a period.

When considering disk retrieval, the completely periodic CPU scheduling behavior may be too strict in some cases. In particular, a process's read() call could be delayed when retrieving the next frame from MMFS. This may occur when the disk is somehow overloaded, but also when smoothing of VBR streams fails (see subsection 5.1.10). The stream may never be able to "catch up" at the disk level once it has missed a deadline, and all subsequent read() calls will be delayed as well. The process will then keep having its CPU periods desynchronized from disk access from that moment on.

To make up for this, we allow for a limited degree of control from the application into the scheduler. The periodic process can specify a 'drift' time value each time it defers execution to the next period. This allows it to move back its period by that time from that moment on.

### *7.2.3. The user process API*

We can now define the following two process API calls for the CPU scheduler:

```
psreserve(period, exectime)
psnext(drift)
```

The first call allows a process to reserve execution time of ‘exectime’ microseconds every ‘period’ microseconds. The process can specify different values at a later time in order to change its reservation, and it can “unreserve” itself by setting the period to zero. The values are rounded up to clock tick values. Admission control is used to prevent CPU overutilization and starvation of aperiodic processes.

### *7.2.4. Scheduling algorithms*

An actual scheduling algorithm takes care of translating those reservations into appropriate process execution, making sure that each periodic process actually gets the desired execution time within each period. We consider two scheduling algorithms: Rate Monotonic (RM) and (again) Earliest Deadline First (EDF).

The RM scheduler is known for the fact that it can be implemented by statically mapping process periods to UNIX priorities [15]. This simplifies the implementation. In our case, use of RM this way has a number of practical disadvantages:

- Very few priorities are actually available in MINIX. With few priorities, many different periods would have to be mapped to the same three or four priorities. Increasing the number of priorities has the effect of slowing down all scheduling decisions.
- Other modifications to the MINIX scheduler would be required to provide isolation from periodic processes that overrun their specified deadlines. One of the original assumptions for the RM algorithm was that such a situation would not occur at all [15].

- RM has a known upper limit on what it can schedule. With many processes, the maximum utilization supported by RM approaches ln(2) ≈ 0.69. In contrast, EDF supports a maximum system utilization of 1.0, meaning that if deadlines can be met at all, EDF will meet them.

Stepping away from the static mapping to MINIX priorities, EDF is then the more attractive option.

### *7.2.5. Scheduling in practice*

A process that makes a lot of calls to system processes must not be able to effectively run for much longer than it is entitled to. The execution time given to each periodic process must therefore include the time spent by system processes. The actual execution time that the user process gets is then equal to the specified execution time minus the time spent by system processes within that execution time.

To achieve this, our scheduler extension can assign a high MINIX priority to each periodic user process for the duration of its execution time. At any time, the “right” periodic process (if any) must be the only process with this high priority. In the case of EDF scheduling, the right process is the process with the earliest deadline (i.e., period end) that still has execution time left in its current period.

As long as the high priority is higher than other user processes, the process always gets to run whenever it is runnable. As long as the priority is lower than (the relevant) system processes, these system processes can still run during the user process’s execution time.

If it the process completes before using up all its execution time, it can give up the remaining time with the psnext() API call. In that case, it is completely stopped for the rest of this period. If, on the other hand, it exceeds its execution time, its priority is changed from high to very low. This allows it to continue in the background until it gets the next batch of high-priority execution time in its next period. That does mean it is now likely to overrun its deadline, because it may not get to run at all while having this low priority.

Figure 7.1 illustrates the execution of a periodic process this way. It gets a high priority for the duration of its execution time somewhere in its period, at the moment that it has the lowest deadline and still some execution time left. In the figure, this execution time is not interrupted by other processes that end up with the lowest deadline. When it calls psnext(), it is stopped until it gets the high priority in the next period. In the figure, the process overruns its deadline in the third period. It is then given a low priority. Only in the absence of other processes can it continue to run. The gray blocks of the process include all time spent by system processes.

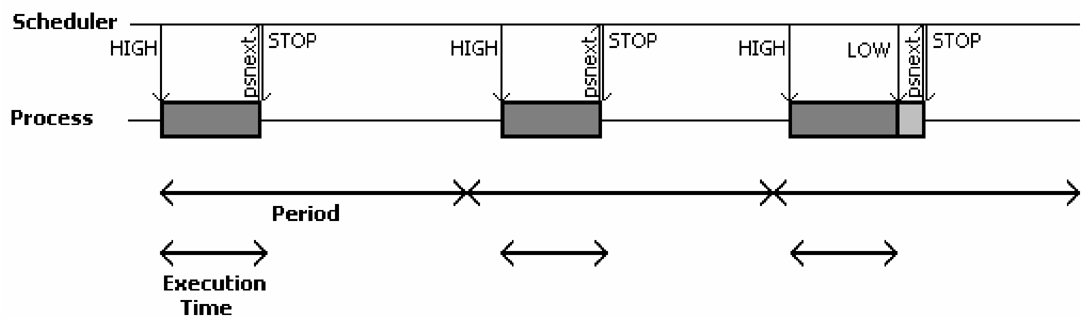


*Figure 7.1: example execution of a periodic process*

Although this is the best that we can achieve without a much more elaborate system in place, it is far from optimal. Background activity of system processes is accounted to whatever periodic process is running at the time. The effective execution time of that process is then reduced by that same amount. As a result, it may overrun its deadline if it has specified an execution time that matches its own execution time very closely. This entire approach is only justifiable because we are concerned with soft deadlines. The same approach would be unacceptable for realtime systems.

## 7.3. Implementation

Based on all the considerations of the previous section, this section presents the actual implementation of the resulting EDF-based periodic scheduler extension. Subsection 7.3.1 explains why the extension can be implemented in a user-space system process. Subsection 7.3.2 describes the procedures taken by the implementation.

### *7.3.1. A scheduling extension in user space*

The resulting algorithm only maintains a bit of per-process state, and is driven by only the two given API calls and a number of timers as input. As output, it only starts and stops processes and updates their MINIX priorities. It is highly isolated from the priority-based CPU scheduler that it builds on. As a result, it can be fully implemented in a user-space system process. We can then avoid having to extend the kernel, keeping it as small as it already was.

The only requirement in this respect is that the process implementing the scheduler always preempts the process being scheduled if one of the scheduling timers goes off. This is true if the priority used for periodic processes is lower than the priority of the system server implementing the scheduler. The overhead of message passing and context switches are simply negligible in comparison to the inaccuracies resulting from the system process accounting and clock tick alignment.

We have implemented this EDF-based periodic scheduling system in the Process Manager daemon, and called it “PSCHED”. The header file defining the two user process API calls is included as appendix A.4.

### *7.3.2. Actual implementation*

The scheduler extension schedules the process with the earliest deadline. It does this by giving the process a high priority for the duration of execution time within its period. The actual execution time may be shortened if the process calls psnext(), or split up if a new process appears with an earlier deadline during this time. Remember that a process’s deadline is equal to the end of its current period, which is also the start of its next period.

The implementation uses two heaps. One heap is used for processes that have not yet used up (or waived) their execution time within their current period. This “deadline heap” is sorted on deadlines. If the heap is not empty, then the process with the earliest deadline in this heap is the one with the high priority. The other heap is used for processes that have no execution time left in their current period. This “period heap” is sorted on period ends (i.e., in fact the same deadlines), which indicate when the process enters the new period. All periodic processes are in exactly one of the two heaps at any given time. The heaps are used in different ways; they cannot be merged.

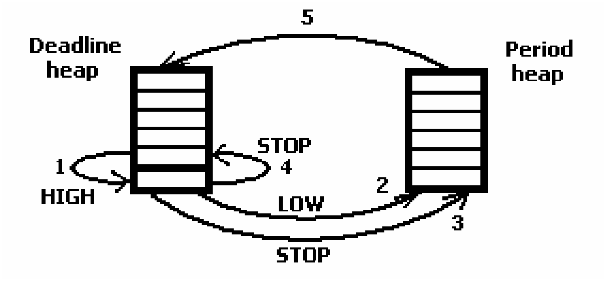


*Figure 7.2: transitions of periodic processes*

The implementation uses three timers. The first timer is set when the next periodic process is scheduled by the extension, that is, when the process is given a high priority (transition 1 in figure 7.2). It is set to go off after the duration of the process's (remaining) execution time in this period. As indicated, three events can cause the process to stop being scheduled with high priority:

- The timer triggers. The process has used up its execution time in this period. It is given a very low priority, removed from the deadline heap, and added to the period heap (transition 2).
- The process calls psnext(). The process is stopped altogether, removed from the deadline heap, and added to the period heap (transition 3). The timer is cancelled.
- Another process ends up having the earliest deadline. The current process is stopped, and its remaining execution time is saved. It remains on the deadline heap (transition 4). The same procedure as above is followed to start the new process with the earliest deadline, and that also involves restarting the timer.

The second timer is set to the lowest period end of the period heap. When it triggers, it refreshes the deadline of the processes for which the new period has started. Those processes are removed from the period heap and added to the deadline heap (transition 5). If the deadline heap now has a new head, the new process with the earliest deadline is scheduled as above.

The third timer is needed to deal with one practical aspect of the MINIX scheduler. As mentioned in subsection 7.2.1, MINIX' scheduler penalizes long-running processes by decreasing their priority over time. Periodic processes that have specified long execution times are likely to be running for a long time consecutively. They may therefore end up with a low priority, even though they started with a high priority. To prevent such processes from eventually reaching (and even crossing) the priority of normal-priority user processes, the third timer is used to “refresh” the high priority of the EDF-scheduled process regularly.

When a process calls psreserve() with a nonzero period and execution time, an admission check is performed. This check makes sure that the total resulting CPU utilization will not exceed a certain percentage of full CPU utilization. This percentage is a constant, and typically less than 100% to avoid starvation of aperiodic user processes. If the check is passed, the process is added to (or updated in) the EDF system.

When a periodic process calls psreserve() with a zero period, it is removed from the EDF system and returns to being an aperiodic process. Exiting processes are removed from the EDF system as well.

The implementation in PM makes use of the sys_nice() kernel call to assign high and low priorities to process, and to stop them (using the special PRIO_STOP nice value). For the timers, PM’s own timer multiplexing system is used, which internally uses the sys_alarm() kernel call. The currently EDF-scheduled process is set to run at a priority in between normal processes and the relevant system processes (nice value -5 in PM, priority 5 in the kernel). The low priority used is the lowest of the system (nice value 20 in PM, prioity 14 in the kernel).

Table 7.1 shows the fields added to the ‘struct mproc’ per-process structure in PM.

| **Field** | **Description** |
|---|---|
| clock_t mp_period; | The period as specified by the process, in clock ticks. |
| clock_t mp_exectime; | The execution time as specified by the process, in clock ticks. |
| clock_t mp_exec_left; | The number of clock ticks of execution time left in the current period. The process is on the deadline heap iff this value is nonzero. |
| clock_t mp_deadline; | The deadline, also known as the period end. |
| int mp_heap_pos; | The position within the current heap, for fast deletion. |

*Table 7.1: fields added to the ‘mproc’ structure*

# Chapter 8. Test results

In this chapter, we describe our efforts to test the results of our work. Section 8.1 looks briefly at a scheduling trace generation extension, which is used later to illustrate several tests. Section 8.2 determines the conditions for all the performance tests, using raw disk transfer tests to determine the gain obtained from using certain block sizes and from using SCAN sorting. In section 8.3, we compare MMFS to MFS in basic disk retrieval tests. Section 8.4 discusses several points MMFS properties. Section 8.5 looks into the organ pipe approach mentioned in chapter 4. Section 8.6 adds nonmultimedia I/O processes to the basic disk retrieval tests. In section 8.7, our new EDF scheduler is tested. Finally, section 8.8 briefly considers the network.

## 8.1. Scheduling traces, and a basic read() call

Quite early in our testing efforts, we were struck by the inability to visualize what was going on in practice. Although several performance analysis tools are present in MINIX 3, these mainly operate on a per-process basis. That makes them unsuitable for showing an important aspect the system behavior: scheduling of processes. This is relevant not only for the CPU scheduling part of this project, but also for general insight into the relative performance of, and interaction between, individual processes.

To overcome this, we implemented a simple scheduling trace extension to MINIX, which we called “schedtrace”. A user process can temporarily enable the scheduling tracer by passing a buffer to the kernel (via a PM call). During this time, the kernel generates an entry in a buffer in kernel space whenever a context switch takes place. Once the relatively small buffer in the kernel is full, it is copied out into the right position of the (typically much larger) buffer of the user process.

The tracing stops whenever the user process explicitly stops it with another call, or when the user buffer is completely filled up. The resulting buffer then contains the scheduling trace, and the user process can write it to a file for example. The actual entries in the buffer that the kernel generates upon each context switch, are eight bytes each. They contain the following information:

- The relative time spent between the current context switch and the last one.
- The first two letters of the process that was active during that time.
- The process slot number of that process.

The time value is measured by taking the difference between system clock values obtained using the i386 ‘rdtsc’ read-timestamp-counter instruction. This currently makes our implementation i386-specific. The timestamp values obtained are CPU speed dependent, and therefore mainly useful as relative values. Of course, some overhead is incurred in obtaining the timestamp, subtracting the previous value from it, and storing the result along with the other information in an entry. The overhead we measured is small though, and we are merely interested in rough visualization of the results, so this tradeoff is acceptable.

An external script takes care of converting the raw trace into a graph. It generates a horizontal bar for each two-letter process name. Drawing of the full names rather than the two letters required manual configuration. The complete trace is mapped to a predefined number of pixels, so the widths of the bars in each graph are not related to their heights. Per pixel, the height of each spike in a process bar indicates the relative time spent on that process in the time represented by the pixel. Red indicates that the process itself was running, yellow indicates that the kernel or a kernel task was running on the process’s behalf (i.e. during message passing and while the SYSTASK task is running). Hues of orange indicate various degrees of mixture of the two. The privileged ‘hlt’ instruction of the idle process has to be called from kernel context as well, so the idle process always shows up as yellow.

Of course, the generated graphs still require quite some knowledge of the active processes in order to be interpreted successfully, in particular because no information about their states is included. Nevertheless, with this new tool we could visualize a single read() call for 128 KB of data from a file on disk. Figure 8.1 shows the path taken from the user process (‘test’) to VFS to MFS to the driver (‘at_wini’), after which by far most of the total time is spent in the idle process, while the hard disk is fulfilling the retrieval request. After a kernel interrupt, the driver returns a result code to MFS. It was MFS’s buffer space that was used to store the actual data in, and MFS therefore has to spend some time copying the result from its own buffer to the user process, using a SYSTASK task call. After that, it sends a result code to

VFS, and starts a read-ahead operation. The ‘at_wini’ driver has a higher priority than VFS by default, so it gets to run first and initiates the next transfer for the read-ahead. Once that is done, VFS runs and replies to the user process.

Note that the ‘other’ category in the figure includes all other processes, and the spike occurring during the disk retrieval is the occurrence of a clock tick. In particular, the “random” random number generation system process runs during every clock tick.

During the time spent in the idle process, all of the other processes are waiting for a result. In particular VFS is performing almost no work at all in the whole action, but was originally still blocked from accepting any other requests from other processes during this time. It was this that made us consider unblocking VFS in the first place.

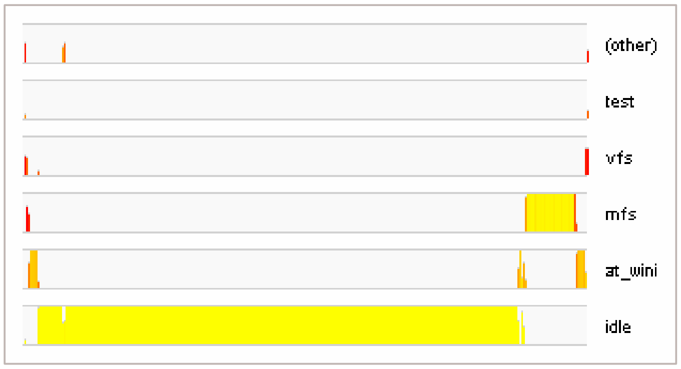


*Figure 8.1: scheduling trace of a single read() call*

## 8.2. Testing conditions

In this section, we briefly discuss our test system (subsection 8.2.1), perform disk retrieval tests for a number of scenarios to study the effect of different block sizes and SCAN sorting (subsection 8.2.2), and determine configuration parameters for the next tests based on these results (subsection 8.2.3).

### *8.2.1. The test system*

Most of the development for this project has taken place on a MINIX installation running in VMware. To eliminate all inaccuracies resulting from the use of such a virtual environment, we have conducted the actual tests on a “real” machine. The specifications of this machine represent the kind of machine that one would currently buy cheaply in order to store it in a closet and let it perform background work:

| | |
|---|---|
| Processor: | Pentium III, 866MHz |
| Working memory: | 256 MB SDRAM |
| Hard disk: | IBM Fireball Plus LM, 7200 RPM, |
| | 12,6 GB (13578485760 bytes) |
| Network card: | Realtek RTL8139 |

### *8.2.2. SCAN on modern disks*

One of the questions that arose during this project was whether latency minimizing disk scheduling algorithms like SCAN are still relevant in current times, given the tradeoff between disk scheduling optimizations and stream startup times. One of our first tests was therefore to test several disk usage scenarios and evaluate the difference between use of round-robin disk scheduling, and SCAN.

We decided to run tests for retrieval of 15 megabytes (15728640 bytes) worth of data at once, from all over the test hard disk, using round-robin (RR) and SCAN sorting (SC), with block sizes varying between 32 and 512 kilobytes. The resulting retrieval durations for each 15 megabyte batch are in 60HZ clock ticks, and we also show the time reduction from use of SCAN over use of RR as a percentage.

We used the following tests:

- Test 1 retrieves a set of blocks that is spread out completely evenly over the disk. The block order is randomized in the round-robin case. Any benefits obtained from using SCAN can be attributed mostly to reduction of disk arm movement.
- Test 2 retrieves a completely random set of blocks.
- Test 3 assumes that the disk contains 35 files of 330 megabytes each. Each request is for a random block within a file that is chosen using on a Zipf-based probability distribution. Note that this test does not span the entire disk.

- Test 4 retrieves a set of contiguous blocks. In the case of SCAN the retrieval is completely sequential. In the case of round-robin, the block order is randomized. Any difference in speed can be attributed mostly to reduction of rotational latencies.

All tests have been repeated twice for each of the two disk scheduling types with the same randomization. This was then repeated several times with different randomizations, to eliminate anomalies due to bad randomization as well as disk-level caching. The average values are shown in table 8.1.

| | **32 KB – 480x** | | | **64 KB – 240x** | | | **128 KB – 120x** | | | **256 KB – 60x** | | | **512 KB – 30x** | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | *RR* | *SC* | *Red.* | *RR* | *SC* | *Red.* | *RR* | *SC* | *Red.* | *RR* | *SC* | *Red.* | *RR* | *SC* | *Red.* |
| **Test 1** | 356 | 246 | 31% | 195 | 145 | 26% | 119 | 96 | 19% | 76 | 63 | 17% | 57 | 53 | 7% |
| **Test 2** | 358 | 222 | 38% | 196 | 134 | 32% | 117 | 89 | 24% | 76 | 64 | 16% | 57 | 51 | 10% |
| **Test 3** | 334 | 213 | 36% | 186 | 129 | 31% | 110 | 84 | 24% | 75 | 60 | 20% | 53 | 50 | 6% |
| **Test 4** | 199 | 39 | 80% | 120 | 38 | 68% | 77 | 36 | 53% | 55 | 36 | 35% | 47 | 36 | 23% |

*Table 8.1: duration of round-robin and SCAN retrieval*

Test 1 and 4 show that reductions in respectively mostly disk arm movement and mostly rotational latency from the use of SCAN are both significant. The SCAN results of test 4 yielded nearly the same times for all block sizes. This shows that for purely sequential access, the block size of individual requests is relatively unimportant.

The same does not hold for the other three tests: the SCAN retrieval for each block size takes longer than the round robin retrieval for blocks twice that size. This means that use of a large block sizes is an overall more important factor, and the choice of block sizes only depends on the choice for granularity of stream rates. As stated before, all stream rates have to be rounded up to a multiple of the block size. That means that with larger block sizes, fewer streams can be admitted.

Nevertheless, the benefits from use of SCAN are substantial for in particular small block sizes. In test 3, which mimics the real world scenario that we had in mind most closely, the time reduction obtained from SCAN is between 35 and 20 percent for block sizes ranging from 32 to 256 KB. With a block size 512 KB, the gain from using SCAN was reduced to less than ten percent, indicating that for block sizes of 512 KB and up, application of SCAN is not worth the increased maximum startup time. Even in this case, though, underutilization of the disk may cause prefetching of several sequential blocks at once. That can still result in even further reduction of round times, as shown in test 4.

### *8.2.3. Configuration parameters*

With these results in mind, we decide to use a 256 KB block size for our tests. We choose driver round times of one second (60 clock ticks). The worst-case scenario of SCAN retrieval then amounts to 56 blocks per round ($60^2$ / 64, the worst case in table 8.1). We reserve one block for nonmultimedia retrieval, so we can deterministically admit 55 streams with a rate of 256 KB per second. Without SCAN, this would be about eight fewer streams with the same rate.

We dedicate four block buffers to each stream at minimum. Four times 256 KB equals a megabyte, so the MMFS process uses a total of 55 megabytes. This is sufficient to support all streams, and allows flexibility in user process read granularity as discussed in subsection 5.1.8. Compared to the total size of the working memory in our test machine, this amount is quite acceptable. The deep prefetching approach makes sure that all memory is actually used even if fewer streams are present.

## 8.3. Basic performance comparison

In this setion, we compare MMFS against the “old” situation, that is, against the MFS MINIX File Server. Subsection 8.3.1 describes how we perform the tests. Subsection 8.3.2 performs tests where stream readers read large blocks at a low rate. Subsection 8.3.3 performs tests where stream readers read small frames at a high rate.

### *8.3.1. Simulation and setup*

To simulate a high number of stream readers, we have written a simulation utility that spawns a number of child processes. Each child executes a basic “read(); sleep();” loop that mimics the data retrieval behavior of any stream reader. It reads a predetermined amount of data in total, and measures the time that each read() call takes, as well as the total time taken by the process. It sleeps in between each two read() calls by a predetermined time. For sub-second delays, select() is used.

We continue with the same file setup as used in the third test case of the SCAN test, creating a dedicated partition for 35 files of 330 MB each. Every reader starts on one of the 35 files, but since we are interested in disk retrieval performance, the simulation utility evenly uses all files and spaces out the positions within each file between the readers. This avoids disk underutilization resulting from (otherwise highly desirable) buffer multi-use and cache hits in MMFS.

For fairness, the tests have been performed with two versions of MFS. One uses the default 5 MB cache (1200 blocks of 4KB), and one has been given a cache that is as large as MMFS' cache: 55 MB (14080 blocks). A similar partition was created for MFS as for MMFS, at the same disk position. During the tests, the MFS partition was always mounted read-only.

All the tests have been performed with the unblocked version of VFS. At least in the first tests, MFS gets no advantage or disadvantage from that. There is always at most one outstanding request from VFS to MFS, so if read requests from the simulation utility are queued, they are now queued within VFS rather than between VFS and MFS.

We assume that mere data retrievel is not the only thing that the user process will be doing. As such, we define a user process request for data retrieval as “lost” if the retrieval takes at least 50% of the period of the stream reader.

A difficult point is whether to let the stream reader compensate for the time it takes to fulfill read() requests. On one hand, if the read() call is delayed because the FS has somehow fallen behind, then the FS will not be able to “catch up” if the requested disk bandwidth is already at or over the maximum (causing the delays in the first place). One lost request is then likely to be followed by many more subsequent losses. On the other hand, compensation for read jitter/overhead may cause the stream to drop below the requested stream rate. There will always be *some* overhead.

As we cannot distinguish between the two cases (we could adapt MMFS to support a limited form of feedback, but not MFS), the stream reader simulator only delays for the read() time if the read request is considered to be lost. This keeps the actual stream rate (as perceived by the hypothetical end user) equal to the requested stream rate whenever possible, but gives better statistics about the actual number of lost requests.

For the purpose of this test, the driver's admission control has been turned off after 55 streams, to see the resulting behavior after the deterministic limit has been reached. The cache sizes of MMFS and the large-cache MFS are then increased to 80 MB to allow up to 80 streams. In the following graphs, a black vertical bar at 55 streams indicates where admission control originally stopped more streams from being admitted.

To allow this many stream readers to be active, we had to change  two MINIX constants:

- '_NR_PROCS' in 'minix/sys_config.h' (from 100 to 150), to allow for at least 80 processes alongside the MINIX system and shell processes.
- 'OPEN_MAX' in 'limits.h' (from 30 to 100), to allow the parent stream reader simulator to have pipes open to all its stream reader children to let them report back the test results.

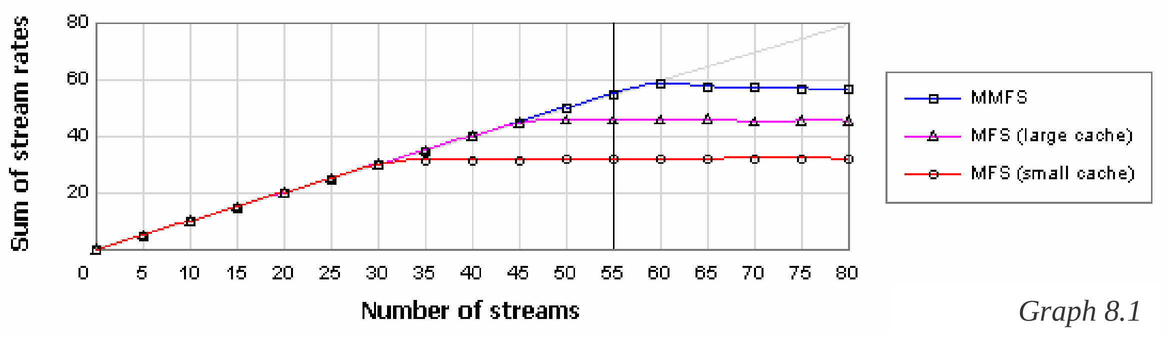
The number of threads in VFS was set to equal the maximum number of processes in total.

### *8.3.2. Large-block performance comparison*

In the first test, the stream reader simulator is set to read blocks with the same size as MMFS blocks: 256 KB each. A rate of 256 KB/sec is used by the stream readers, so that each stream reader reads one 256KB block per second, exactly equal to the speed at which MMFS requests blocks from the driver for each stream. This simulates a stream reader that does its own internal buffering on top of MMFS. The next subsection will present a more realistic scenario in this respect.

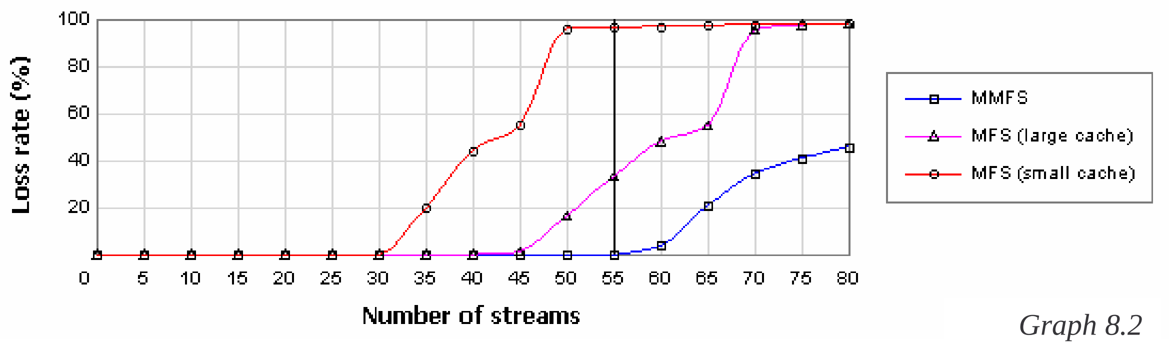
We deliberately match the exact rate of MMFS' block retrieval from disk in this test. Using stream rates not aligned to the effective request rate only results in fewer streams being admitted (see subsection 5.2.4). That is not very interesting from a testing point of view. Also, using a multiple of 256 KB/sec for one stream gives the same results as using multiple streams at the 256 KB/sec rate.

With a period of one second, the time after which read requests are considered "lost" is half a second (i.e., 30 clock ticks with the standard 60 HZ clock frequency). The number of forked stream readers is varied from 5 to 80 in steps of five. The streams are started up simultaneously, and all of them run for the time it takes to read 200 times 256 KB. With no delays, this takes 200 seconds. All data points in the graphs below are the result of at least three runs, so that each data point represents at least ten minutes of testing time. The file system is remounted between each two tests to prevent any accidental cache hits.



*Graph 8.1*

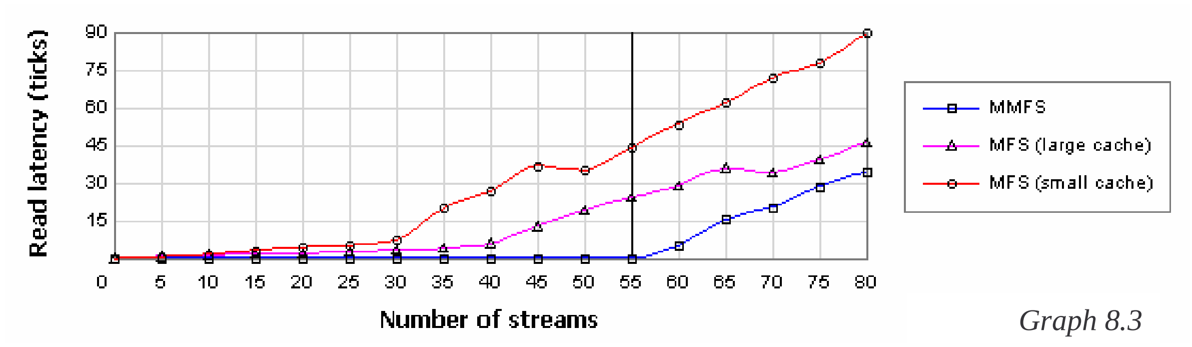
Graph 8.1 shows the total sum of the stream rates that all streams received combined. Here, the actual rate of a stream is represented as the average fraction of 256KB-blocks it was able to read per second, that is, the time it took to read all 200 blocks divided by 200. Ideally, this value is exactly equal to '1' for each stream, and equal to the number of streams in total. This ideal line is shown by the straight line from the lower left to the upper right corner of the graph.



*Graph 8.2*

Graph 8.2 shows the average loss percentage of each stream. Graph 8.3 shows the average response times of read() calls, in 60 HZ clock ticks.

The cumulative bandwith graph 8.1 clearly shows the upper limit of all three FS servers: MMFS follows the ideal line up to 55 streams, after which it settles around 58 streams. The MFS with the large cache can support 45 streams, and the MFS with the default cache hits its upper limit at 33 streams.



*Graph 8.3*

The difference between the MFS instances with different cache sizes can be explained by looking at MFS's prefetching strategy. MFS performs prefetching of consecutive data both during and after each read. It rounds up each read request to a minimum of 128 KB, and then performs an additional (possibly overlapping) 128KB read-ahead attempt after the read request, starting at the reader's final read position.

In this test, 256 KB blocks are requested at a time by the user processes, so no rounding up takes place. On the other hand, an additional 128 KB is requested from disk and cached immediately after. This means that half of the next 256 KB request for that file will be available in the next read() call. The remaining 128 KB has to be fetched at that moment, after which another consecutive 128 KB will be read ahead. Therefore, with an unlimited cache size, MFS will read 256 KB from disk as a result of every read() (except the very first one) – in two consecutive 128 KB requests rather than one 256 KB request like MMFS does, but test 4 of subsection 8.2.2 shows that that hardly matters.

However, once the cache blocks used for prefetching are reused before the next read() call can take advantage of them, then the next read call will end up retrieving the full 256 KB after all. In addition, it will read ahead 128 KB that will similarly end up being unused. Each read() call plus read-ahead then requires 384 KB of cache blocks, and the default MFS has 1200 cache blocks of 4 KB each, so the cache should be when at least 13 test streams are active (~ 1200 / (384 / 4)). In practice it happens earlier, because MFS also needs a number of blocks for the inode metadata that maps file positions to disk blocks. This explains why the small-cache MFS performs so badly: it quickly ends up reading streams at 384 KB per second.

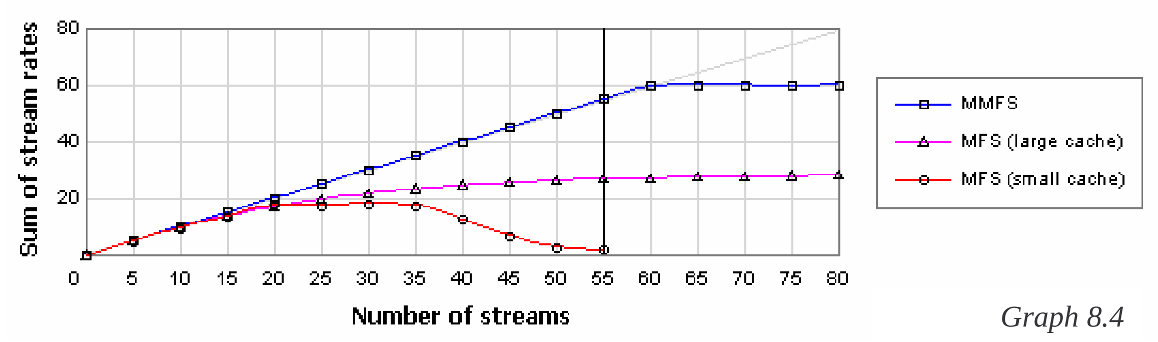
The large-cache MFS does not suffer from this. It constantly requests two consecutive pairs of 128 KB-blocks. Eventually it ends up losing out to MMFS due to the lack of SCAN sorting of its requests. The latency of the read() calls already starts going up before that, because each read() call is blocking all other read() calls for the duration of the complete 256 KB retrieval (in the form of two blocking calls to the driver). The average queuing time of the read() calls at VFS increases as more streams are active and their read() requests are not evenly spread out within each second. With a blocking VFS, MMFS would have suffered from the same problem.

In all cases, the MMFS performance is equal to what could be expected up to the limit of the admission controller. MMFS indeed supports 55 streams of 256 KB per second. Each stream has data delivered at the requested rate, with absolutely no loss and minimal read() duration, as all data requested by the stream readers is available in advance. MMFS can only support three or four streams more than the hard, worst-case limit of the driver’s deterministic admission controller. After that, the streams start suffering from increased read call duration as MMFS cannot prefetch all data in time any more. They experience reduced per-stream bandwidth, and eventually a significant retrieval loss as well.

### *8.3.3. Small-frame performance comparison*

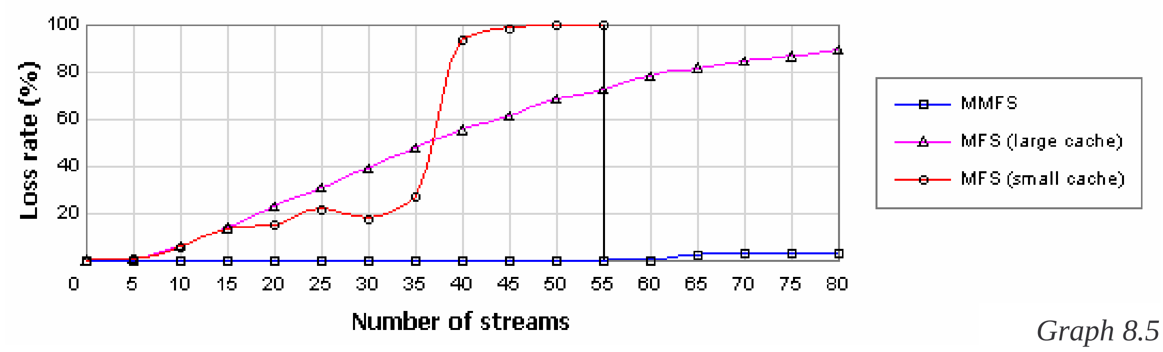
Typically, real-world stream readers would read individual frames rather than huge blocks at a time. To complement the large-block test, we developed a test that would read relatively small frames at a time, varying in size. We mimic the common “IBBPBBPBB” encoding pattern of I/P/B-frames for MPEG streams. We assume that a stream reader itself would be reading ahead to process B-frames as necessary. We use the rule-of-thumb average size ratio of 4:2:1 of I:P:B frames [5] to determine the read size used for the frames. With this pattern, the stream rate of 256 KB per second and a frame rate of 25 frames per second yields B-frames of 6740 bytes, and P and I frames of two and four times that size. The requested stream rate is 262111 bytes per second (6740 * 14/9 * 25), 33 bytes less than 256 KB/second. For MMFS, this means that frame reads are not aligned to its 256KB-blocks any more.

A stream rate of 25 frames per second is impossible with MINIX' standard clock frequency of 60 HZ. We therefore increased the clock frequency setting in MINIX to 1000 HZ, that is, to one clock tick per millisecond. Like the other configuration changes, this only required a change to a constant in a header file ('HZ' in 'minix/const.h') and recompilation of MINIX. No subsequent deterioration in performance was measured, and we have used this clock frequency for all the remaining tests. For this test, it means that each stream reader reads one I, P or B sized frame from the stream file every 40 milliseconds. The request is now considered lost if it takes more than 20 milliseconds.



*Graph 8.4*

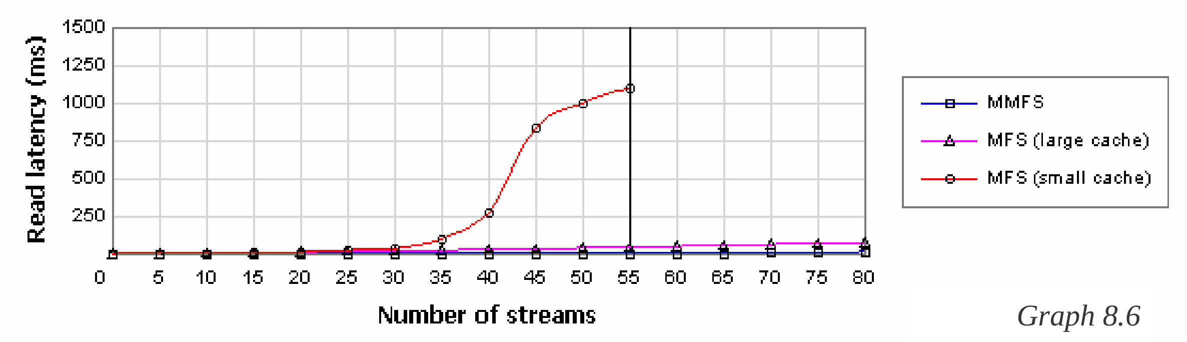
The resulting graphs 8.4, 8.5 and 8.6 show the stream rate, loss percentage, and average read() response time, respectively. The latter is in milliseconds this time. The first thing to note is that the small-cache MFS test was not continued beyond 55 streams. At that time, the cumulative stream rate had dropped to about 1.8 256 KB-blocks per second, so each test would take several hours.



*Graph 8.5*

MFS's general poor performance in this test is not the result of its prefetching strategy. Prefetching stops whenever a block is found to be already present in the cache. As long as the cache is not overloaded, only 128KB-blocks of data are fetched from the disk at a time.

Instead, the behavior of our stream reader is responsible. The scenario requires that 25 frames per second are retrieved and processed, and MFS simply cannot keep up with the low requirement of a maximum of 20 milliseconds per read. This causes the stream reader to consider many reads as lost, and delay the next reads. The fact that many reads take longer than 20 milliseconds is again because MFS is blocked during the retrieval of 128 KB into the cache. This starts being an issue with as little as fifteen streams, and causes the perceived stream rate to fall below expectations from that point on.



*Graph 8.6*

Despite the loss-induced delays, the large-cache MFS eventually approaches its maximum of roughly 28 streams. This is substantially less than the large block test, and fully due to the fact that it now retrieves 128 KB of consecutive data at a time from the disk rather than 256 KB. The 3:2 retrieval duration ratio between round-robin retrieval of the same data in 128 and 256 KB pieces shows up in the test in subsection 8.2.2 as well.

The small-cache MFS does not do as well as the large-cache MFS, even with a low number of streams. Inode metadata blocks are pushed out of the cache just as often as data blocks, forcing MFS to reread them often. This happened in the large-block test as well, but with the tighter delay requirements, the resulting latency increase is no longer negligible. A subtle play between these delays and (temporary) lower latency for other processes causes the small-cache MFS to have an loss rate that is – unintuitively – below that of the large-cache MFS for low numbers of streams.

The small-cache MFS experiences a true performance disaster when its cache is overflown. This does not happen as fast as in the last test: each stream now needs only 128 KB of cache in order not to have its prefetched blocks thrown out, and that yields a maximum of rougly 37.5 streams in theory (~ 1200 / (128 / 4)). Again, it is less in practice due to the inode metadata blocks. Worth noting is that during investigation of initial test results, it was discovered that MFS would actually write out inode blocks to disk even when mounted in read-only mode. This was solved by building in extra preventive checks into MFS.

Once again, MMFS is able to provide full guarantees for at least 55 streams. It levels out pretty soon after. Graphs 8.5 and 8.6 are somewhat misleading in this respect. MMFS only operates on whole 256 KB blocks, and both the loss rate and the read call durations are averages. That means that many of the small frames will still be available immediately once a 256KB-block has arrived, no matter how long it took to fetch that block. As a result, the loss and read latency *averages* stay low after 55 streams, but the *variance* (not shown) increases steadily.

## 8.4. MMFS performance

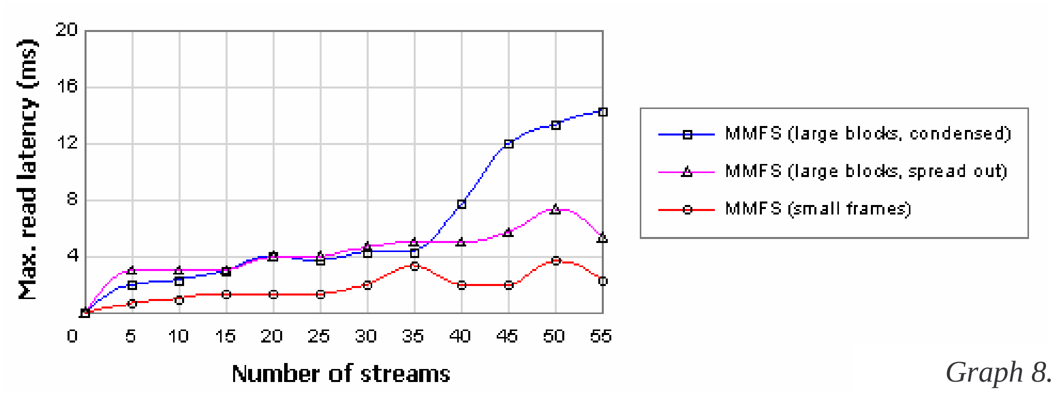
There are several aspects that are relevant only for MMFS and worth discussing in more depth. This section discusses the duration of read() calls (subsection 8.4.1), round durations (8.4.2), stream startup times (8.4.3), and closedown times (8.4.4).

### *8.4.1. Read call jitter*

If we zoom in on the bottom of the two read latency graphs (8.3 and 8.6), the average read response times of MMFS between the two tests reveals a difference, even with the large block test redone at 1000HZ. The jitter of “small” read calls is much less than with “big” read calls, despite the fact that more individual requests have a higher total overhead.

Closer analysis reveals that the difference is fully due to the time spent copying out data from the MMFS buffers to the user process. This is amplified by the fact that our stream reader was set to spawn all streams at the same time. Most streams are requesting a 256KB block around the same time in each second, resulting in a peak of activity per second. This is clearly visible in the scheduling trace included as appendix D.1, which shows about 1.6 seconds of running time of 55 stream readers (“reader”) and relevant system processes. The blocking nature of the copy-out process causes implicit queuing of read requests within MMFS, and with that the relative increase in read response times.

When we introduce a random delay in each stream reader after startup, the activity of the processes is more spread out. This can be seen in the scheduling trace in appendix D.2. Graph 8.7 shows the maximum (not average) read latencies of all runs, with both the original condensed and the now spread-out variant. With this modification, the read operations now take 2.1 milliseconds on average. The average was 2.4 milliseconds before, but the maximum is roughly halved. In reality, no assumptions can be made about when streams are started, but the chance that most of them perform all their read() calls at the same moment is low.

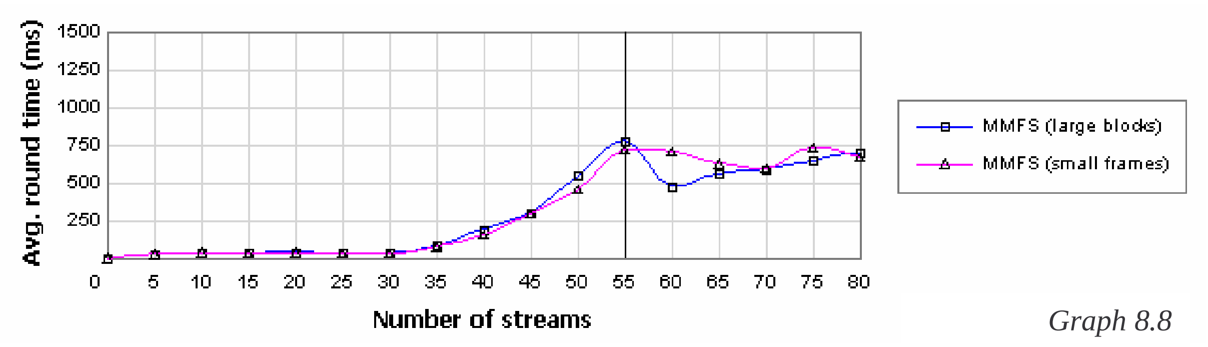


*Graph 8.7*

Smaller reads already take 0.001 milliseconds on average. In general, the use of smaller read requests is beneficial to MMFS: it limits the duration of individual read requests, because smaller amounts of data have to be copied out at a time, so the copying operations are more spread out by themselves.

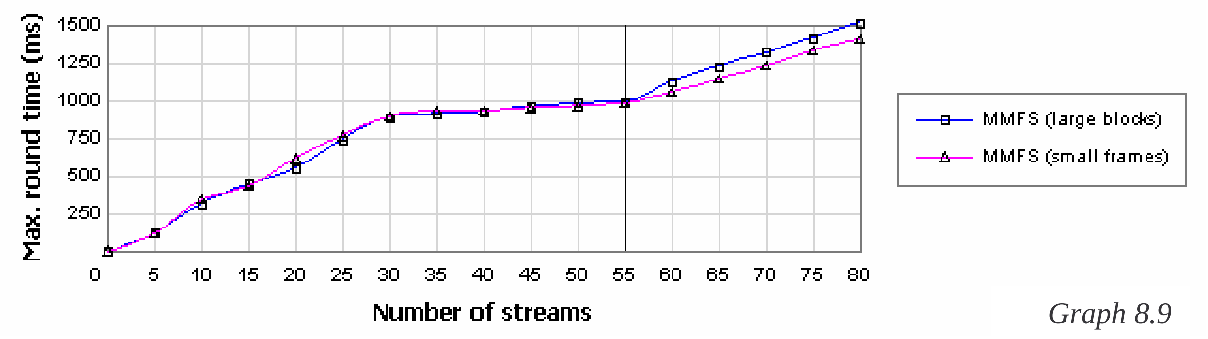
### *8.4.2. Multimedia round durations*

With dynamic driver round times (see subsection 5.3.3), the actual round durations are an important aspect that play a role for in stream startup times (see subsection 5.1.9), deep prefetching and VBR (see subsection 5.1.10), and integration of multimedia and nonmultimedia disk requests (see subsection 5.3.2). In all cases, shorter actual round durations yield a bigger advantage for both multimedia and nonmultimedia user processes. Graph 8.8 shows the average round durations during the large-blocks and small-frames tests.



*Graph 8.8*

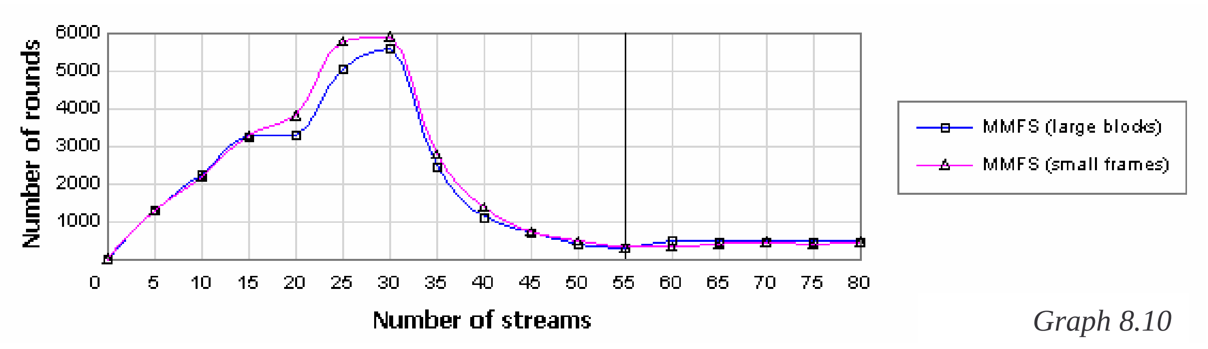
Up to 30 streams, the average round times stay very low. Even with 55 streams, the average is well below the upper duration of one second. Graph 8.9 shows the *maximum* round duration in each simulation run.



*Graph 8.9*

The maximum round duration is never even closely reached until about 30 streams. After that, the maximum is still below one second in all cases. After 55 streams, the maximum round sizes (in number of blocks), and consecutively their durations, start exceeding their limits, resulting in a linear increase beyond one second.

A closer analysis of individual round times reveals that the round with maximum duration takes place very quickly after all streams have been opened. Soon after, the sub-second round durations cause the streams to prefetch faster than the stream rate, after which the limit on the number of blocks per stream is hit within MMFS. Per-stream prefetching then continues at the same rate as the stream itself: in the test case, one block per second. With a relatively low number of streams, and therefore low disk utilization, small numbers of requests are queued at the driver at any time. That causes many more small rounds of the observed low average durations. Graph 8.10 shows the total number of rounds as a function of the number of streams.



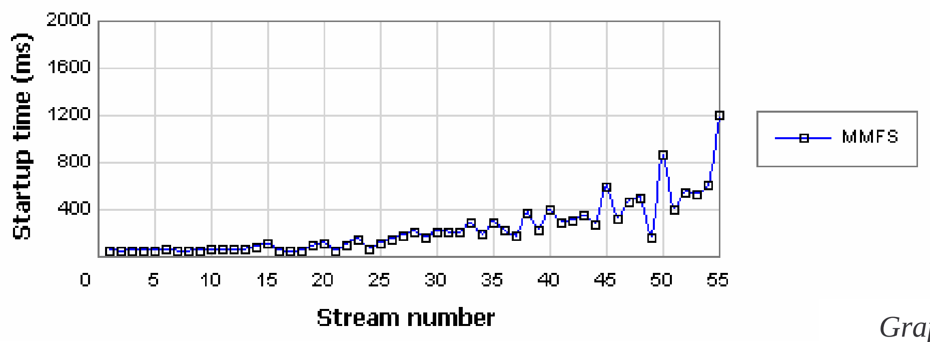
*Graph 8.10*

The number of rounds is rising almost linearly up to 30 streams, with nearly as many rounds as there are streams (for the large-block test, 30 streams each reading 200 blocks during the run equals 6000 requests). Here, each round consists of *one single* multimedia request, which is always fulfilled before at least two new requests come in. Only after 30 streams does the driver start aggregating multiple requests into single rounds on average. Eventually, the number of rounds drops to a value that is close to the minimum value of 200 rounds for 200 requests per stream, regardless of the number of streams.

In short, this means that the advantage from SCAN-sorting of individual rounds only starts becoming relevant from about 30 streams, that deep prefetching for streams is only bounded by the number of buffers in MMFS below this number, and that the short round times should give nonmultimedia requests low response times even with a number of streams close to the maximum. We will look at the last point in more detail in section 8.6.

### *8.4.3. Stream startup times*

In a sense, the startup time is the most important drawback of MMFS compared to file servers that do not rely on prefetching (e.g., MFS), because this startup time is always perceived

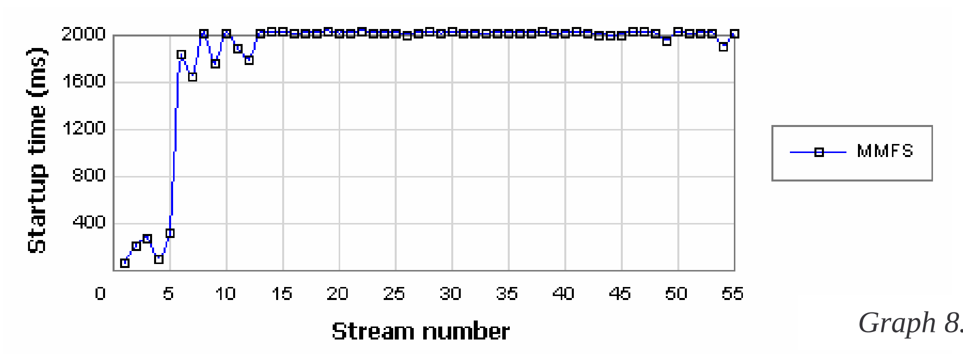


*Graph 8.11*

directly by an end user. It is therefore important how well our optimization of startup times (subsection 5.1.10) works in practice. Without this optimization, opening a stream would always take two full seconds. In the following test, the stream reader simulator was modified to open one stream at a time, and then wait five seconds before opening the next stream. Graph 8.11 shows the startup time for each individual stream.

The graph shows that MMFS takes full advantage of the disk underutilization when few streams are active. For the first 20 streams, the startup time is never more than 100 milliseconds. After that, the startup times start rising, but they never reach the full two seconds, not even at 55 streams, because of the sub-second round durations.

Of course, the stream open requests are completely evened out over time in this test. This is an ideal situation. If multiple streams are started at the same times, the disk utilization will experience a much larger temporary peak, resulting in longer startup times for each of the streams. Graph 8.12 shows the same test but without the five-second delay between the startups.



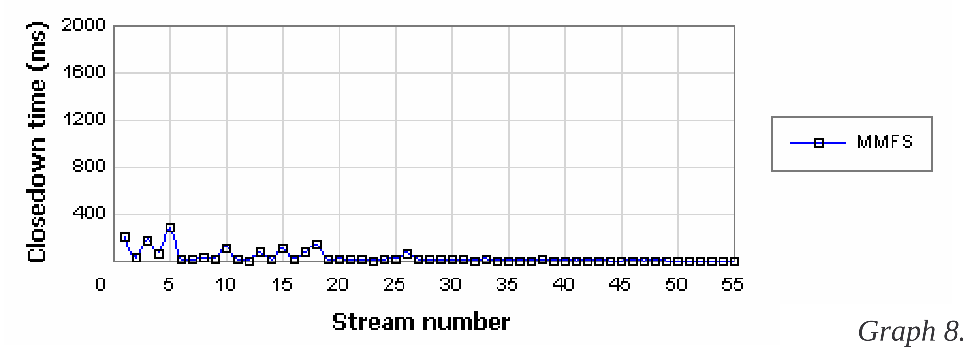
*Graph 8.12*

Scheduling latencies prevent all stream startup requests from taking place at the exact same moment. As a result, the first five streams get away with relatively low startup times. The startup times of the remaining streams quickly hit the ceiling. With 55 streams, the driver still fulfills all data retrieval requests with sub-second round durations, but the data is not available early enough to allow for the startup optimization.

An important fact to note is that our test setup has all streams positioned completely separate from eachother within all the files, so that there is no overlap between buffers in MMFS. Especially for startup times, this is the absolute worst case. If some initial data blocks of a starting stream are already available in buffers owned by other streams (or cached), then this reduces the startup time of that stream. After all, MMFS does not have to fetch those blocks from disk. If *all* blocks are available, the startup time is reduced to zero.

### *8.4.4. Stream closedown times*

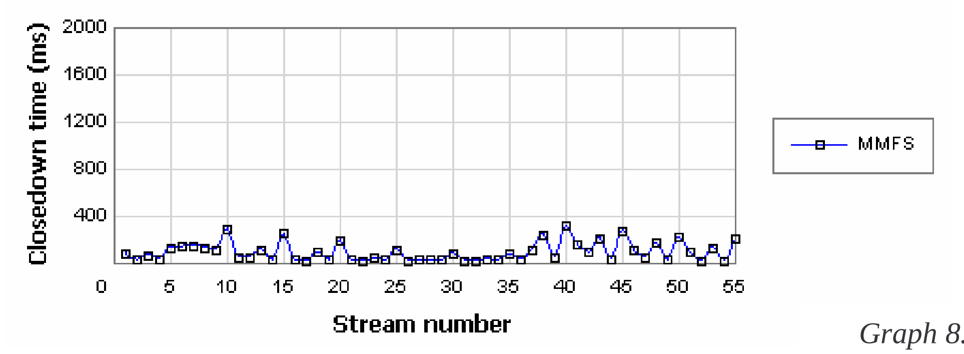
Another point of interest is the extra time it takes to close down streams, that is, to cancel any outstanding requests for data blocks with a deadline further ahead than the work-ahead time, and to wait for outstanding requests earlier than that time. We changed MMFS slightly to acknowledge the STREAMCTL_CLOSE request only after the stream has been fully closed and unregistered at the driver. This allows the stream reader to obtain statistics about the time it takes MMFS to close streams. The following test is constructed in the same way as the last one: starting off with 55 streams, the streams are closed one by one, five seconds apart, and the closing time of each stream is measured. Graph 8.13 shows the result, starting with the first stream closing first and ending with the 55th stream closing last.



*Graph 8.13*

Although closing streams could take up to two seconds due to the inability to cancel outstanding requests in time (see subsection 6.3.4), the values of this test are nowhere near that upper limit. This is because of the same lack of overlap between streams in our test setup that caused the worst-case startup times. If a nonfilled buffer of one closing stream is not part of the window of another active stream, MMFS tries to cancel its pending request immediately. The cancellation may or may not succeed, based on whether the request is already part of the current round (again, see section 6.3.4). This means that at worst, the cancel time of each stream is the duration of the current round at the time that the stream starts closing down. As shown in subsection 8.4.2, that is roughly 0.7 seconds. The highest measured individual close time in this test is indeed a little below 0.7 seconds, and the averages are much lower. The closedown times drop further when the round times get shorter as fewer streams are left.

When closing down all streams at once rather than five seconds apart, there is no significant increase in the closedown times of individual streams (graph 8.14). However, there is no advantage from shortened round times, so that the total average closedown time is higher.

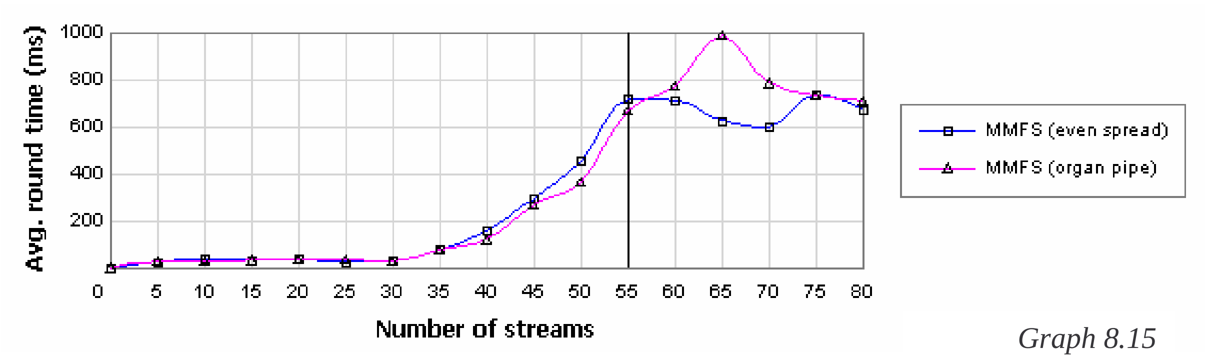


*Graph 8.14*

## 8.5. The organ pipe scenario

In the next set of tests, we temporarily moved away from the worst-case scenario (in many respects) of having all streams operate on different sections of all the files. We reintroduce the Zipf-based file access distribution and random initial in-file positioning that we also used in test 3 of subsection 8.2.2. Streams may now *overlap*, reducing the number of actual requests made to the disk driver.

It turns out, however, that the chance that two streams are operating within the same 1.75 MB section of a single file (in order to have overlap at all) is sufficiently small that the results are nearly identical to the earlier tests. As all streams have the same rate for each file, the occurrence of such overlap between two streams translates in disk usage reduction by one stream. On average, 0.5 stream is experiencing overlap in each test. Consecutively, the closedown times of streams remained practically just as low as in subsection 8.4.4, even though buffer sharing could have had a negative impact here. We have not included the new graphs for these tests here, as they are indistinguishably close to the original ones.



*Graph 8.15*

Graph 8.15 shows the average round durations for even spread and organ pipe scenarios for small frames retrieval. As expected, the driver's multimedia round times between 30 and 55 streams are lower than with the even spread of readers across the files. With the organ pipe approach, most blocks are closer to each other on disk, yielding a speed gain from faster SCAN disk retrieval in each round. The differences are small though. Overall, we conclude that the organ pipe approach is not very beneficial in our test setup.

## 8.6. Presence of other disk loads

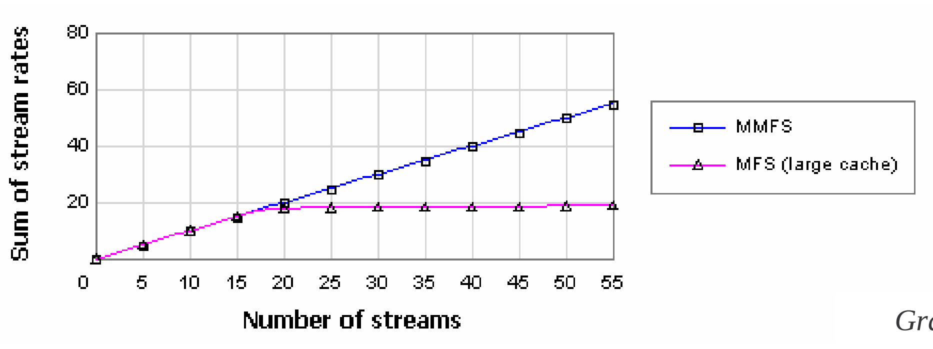
The previous tests show that MMFS is able to provide minimal read() call latency, and get more out of the hard disk than MFS. We therefore achieved our first and second goals. We now move on to the third goal: the isolation from other processes. This section covers isolation of disk access, the next section covers CPU scheduling.

In this section, we consider two scenarios: that of one I/O-intensive process on another partition (subsection 8.6.1), and, mainly for MFS, that of ten I/O-intensive processes on the same partition (subsection 8.6.2).

### *8.6.1. One I/O-intensive process on another partition*

In the following test, we test how wellprotected MMFS is against the I/O of nonmultimedia processes, and obtain some statistics on the I/O response times for these processes. This test combines the original tests with a single nonmultimedia process that is reading nonsequential 4KB-blocks from a 80 MB file on a different MFS partition at maximum speed.

We do not look at the small-cache MFS and the small-frames scenarios any more, as in both cases, the MFS performance is too low to be acceptable. Also, no more than 55 streams were tested this time. Graph 8.16 shows the combined effective stream rates for MMFS and the large-cache MFS version in a large-block based test.

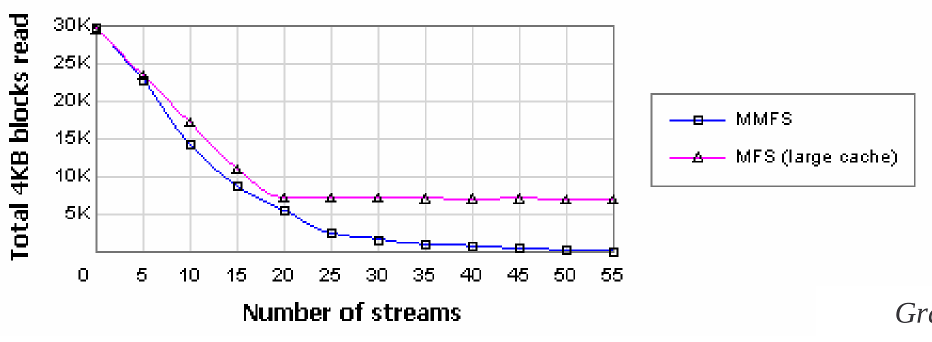


*Graph 8.16*

There is a much bigger gap in performance between MMFS and MFS than before. As far as VFS is concerned, the two types of processes are completely independent, as they operate on different partitions. Both types of requests (multimedia stream data requests and full-speed I/O requests) are therefore processed by the two file servers in parallel. In the case of MMFS, this leads to the desired combination of multimedia and nonmultimedia requests into rounds. As the graph shows, the stream readers experience no deteriorated performance at all.

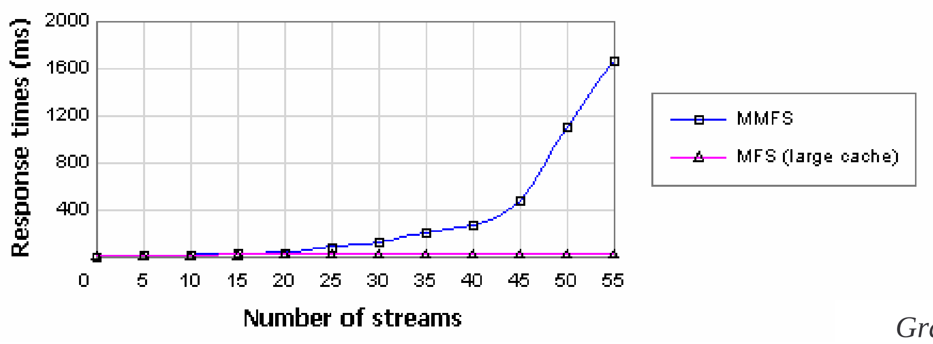
In the case of MFS, each of the two MFS instances are processing one request at a time, and their requests are combined in first-come-first-serve order by the driver. This roughly leads to a 1:1 ratio of requests served: for every multimedia stream request, one nonmultimedia request is processed. This effectively halves the bandwidth available for stream requests, and is only alleviated by the fact that the I/O-intensive process can not have a request queued at all times. For the large-blocks test however, the rough 1:1 ratio means that the two consecutive 128 KB blocks that make up each 256 KB-request (see subsection 8.3.2) are not guaranteed to be consecutive any more. This results in a little over half the total combined effective rate of the original large-blocks test.

As can be expected, the requirement of guaranteed stream rates of MMFS comes at the expense of the performance of the I/O-intensive process. Graph 8.17 shows the total number of 4KB blocks read by the I/O-intensive process, and, more importantly, graph 8.18 shows the average response times of individual read() calls of this process.



*Graph 8.17*

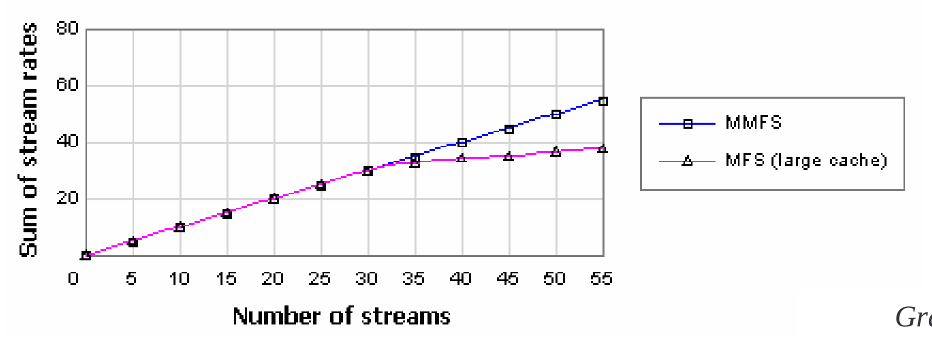
As long as the round sizes stay close to one multimedia request per round (even with the addition of the nonmultimedia requests), the ratio between multimedia requests and nonmultimedia requests stays close to 1:1, and the I/O-intensive process in the MMFS run is able to perform only slightly worse than the one in the MFS run. Above 20 streams, this starts changing. As round durations increase, the number of requests starts going down and individual read call latency starts going up, to the point where the rounds approach their maximum. The MFS instance that the nonmultimedia process operates on, also needs to retrieve a number of meta-blocks for the file, so the average read() call actually takes longer than a complete round.



*Graph 8.18*

### *8.6.2. Ten I/O-intensive processes on the same partition*

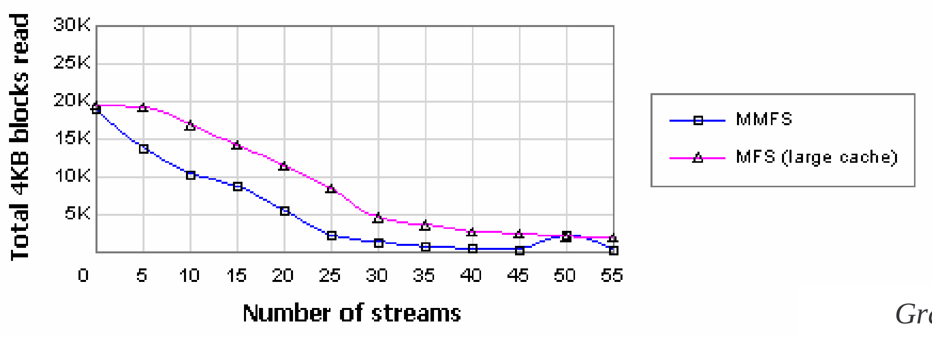
The last test is not entirely fair with respect to MFS when compared to the original unblocked VFS. The observed behavior is a result of two MFS processes being able to process requests in parallel, which is only possible due to the fine-grained locking implemented in VFS. With the original VFS, all the different requests were serialized before being processed by any MFS instance. The 1:1 ratio between the multimedia stream requests and the I/O-intensive process would then be much closer to N:1, where N is the number of streams. Or more general, closer to N:M, where M is the number of I/O-intensive processes. We test this by creating ten I/O-intensive processes that operate on the same partition as the streams. This more accurately reflects the situation for MFS as it was before unblocking VFS. As MMFS does not support reading from files without attaching a stream to the file, we have to let the stream reader read from another partition in the case of MMFS.



*Graph 8.19*

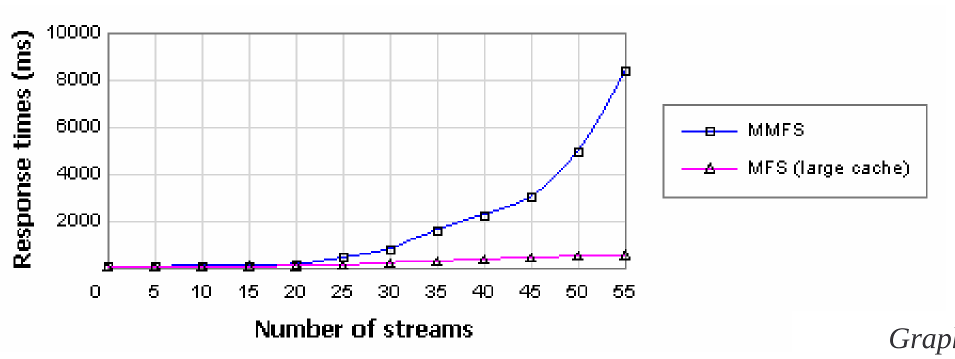
Although it is unusual to have ten processes running that all try to use the disk at the highest possible rate, this scenario does present a long-time version of peak activity. Ten other processes that happen to read from the disk at virtually the same time is possible if temporary heavy processing takes place. In that case, a similar drop in performance for multimedia processes could be expected, although generally for a shorter interval.

Graph 8.19 shows the resulting stream rates. Graph 8.20 shows the total number of blocks read by the ten I/O-intensive processes combined. Graph 8.21 shows the response times of individual read calls. Note the change in scale for graph 8.21 compared to graph 8.18. After all, ten processes are fighting for access to the disk instead of one.



*Graph 8.20*

As expected, the large-cache MFS performance for streams is roughly comparable to that of subtracting ten streams from the maximum observed in subsection 8.3.2. MFS is able to sustain as many as 30 streams without any loss, while it is able to serve more nonmultimedia requests at a lower average latency than MMFS. In this case, it actually does slightly better than MMFS. For 35 streams and up, MFS cannot keep up the stream rates any more.



*Graph 8.21*

With MMFS, as soon as multimedia rounds start forming with two or more multimedia requests, the nonmultimedia requests get a smaller relative fraction of each round. This happens even if there is in principle enough room to serve more nonmultimedia requests. That explains the relative worse performance of the I/O-intensive processes with MMFS up to 30 streams. A pragmatic improvement would be to let new nonmultimedia requests be inserted in a currently running round, as long as the worst-case time addition resulting from these insertions (which violate the SCAN sorting) would never make the current round exceed the maximum round duration time. We have not tried this in practice because of time constraints.

## 8.7. Presence of other CPU loads

The previous tests show that the newly developed VFS/MMFS/driver combination can support guaranteed disk retrieval for a high number of streams. In the next subsections we put our new “PSCHED” scheduler extension to the test, showing how well it isolates multimedia processes from nonmultimedia processes (subsection 8.7.1) and eachother (8.7.2).

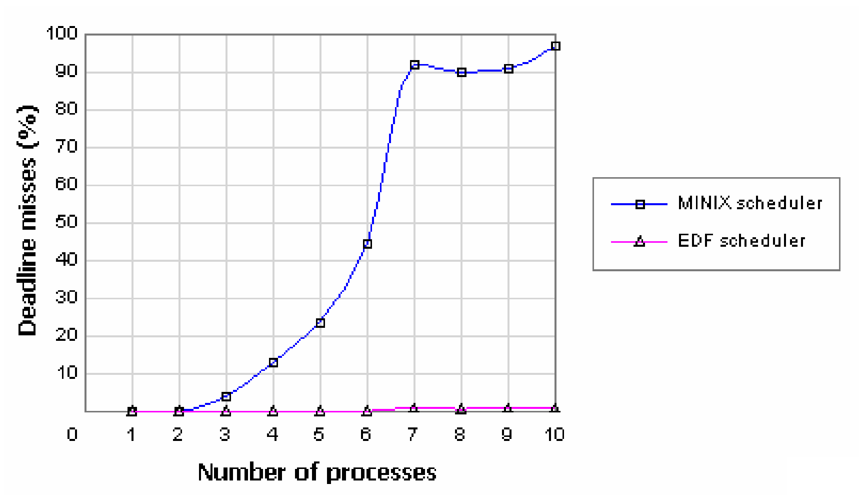
### *8.7.1. Isolation from nonmultimedia processes*

To test the EDF-based scheduler, a new scenario is needed. If we simply let each of the 55 streams request 15-20 milliseconds of CPU time per second, this would yield predictable results, because all periods are equal. Such a scenario would also not accurately reflect the need for stream readers to process the data retrieved.

Instead, we focus on a lower number of streams with different periods and execution times. Since the scheduling functionality has been developed with the new disk retrieval model in mind, we use only MMFS in the following tests. We consider the case of stream readers having to share the total CPU time with several CPU-intensive background processes. Our new PSCHED scheduler is compared to MINIX 3's standard priority-based scheduler. We avoid finetuning of the priorities of system processes in MINIX, as that could easily lead to a very specific solution that only works for the specific test setup [21].

We consider two stream reader processes that read a small frame and then require 10 milliseconds of processing time, once every 40 milliseconds (processes 1 and 2). On top of those, we add four stream reader processes that read a large block and then use on average 100 milliseconds per second with periods of 0.5, 1, 2 and 4 seconds (processes 3, 4, 5, 6). These six processes together use 90% CPU utilization. We set the scheduler admission control to a maximum utilization of 95%, so all these processes can be admitted. The processes all use a 256 KB/sec stream rate overall. On top of these six processes, we add four background processes that will use up as much CPU time as they can (processes 7, 8, 9, 10).

In the test, we vary the number of processes from one to ten, following the numbering scheme given above (i.e., the last four processes added are the nonmultimedia processes). We measure



*Graph 8.22*

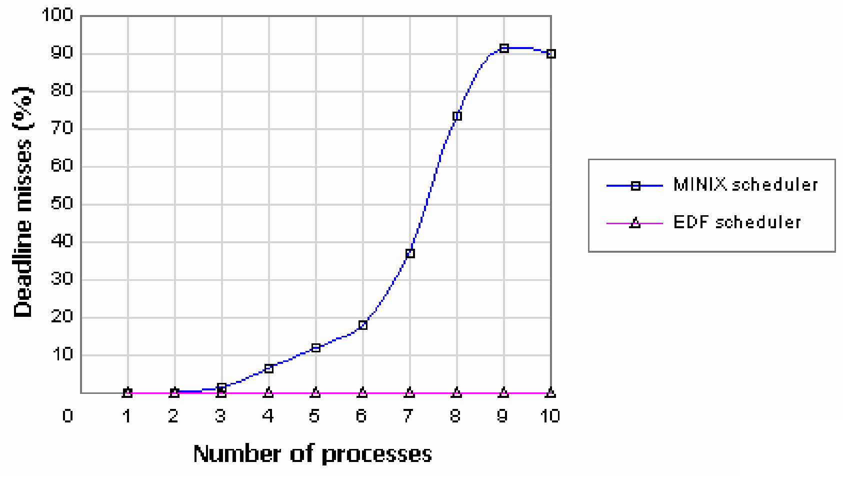
the number of deadline misses (period overruns) over a 200-second period. Graph 8.22 shows the percentage of deadline misses as a percentage of the total number of execution periods of the periodic processes (i.e., up to six processes), averaged over three runs. As illustration, appendices D.3 and D.4 show representative scheduling traces from the tests of the MINIX scheduler and our PSCHED extension, respectively.

It is clear that the standard priority-driven scheduler in MINIX 3 is unsuitable for periodic multimedia processing. With only three processes, deadline misses start occurring. The miss percentage goes over 90% in the presence of the nonmultimedia CPU-intensive processes. Not shown is that in the presence of two or more CPU-intensive multimedia processes, the multimedia stream readers are not able to keep up their stream rate any more. They simply do not get to make enough read() calls in the given time period.

This is not the case with our EDF-based scheduler. However, because of the system time accounting limitations of our approach, and the lack of system server priority tuning, we could not fully eliminate deadline misses. These deadline misses result from temporary high activity of any system process during the execution time of each user process. The actual execution time of each periodic process is very close to its specified execution time, so all these processes are extremely vulnerable to such interference. Deadline misses start occurring with the 90% utilization and CPU-intensive processes (i.e., 7 processes and more). On average, roughly 50 out of 10756 deadlines or a little under 0.5% were overrun, nearly all by the first two processes that have very short periods. The main source of the interference is clustered memory copying activity of MMFS.

If one of the multimedia processes overruns its deadline, its priority will be lowered to below that of the CPU-intensive processes. Without any other activity on the system, it might still be able to complete its work within the period at lower priority, but this becomes impossible with the presence of the CPU-intensive processes. That explains why deadline misses only start occurring after adding the CPU-intensive processes.

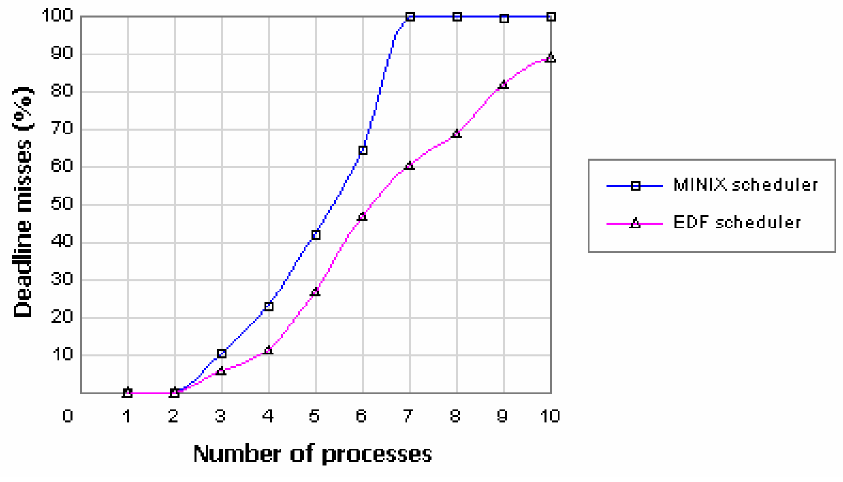
Graph 8.23 shows the deadline miss percentage after reducing the actual execution time of each of the periodic processes to 75% of their specified execution time. The number of deadline misses of PSCHED is now reduced to zero. Needless to say, the execution time of individual processes can typically not be specified and varied in reality. Instead, the application would choose a higher periodic execution time than it actually needs. The periodic scheduler can then admit fewer processes in total.



*Graph 8.23*

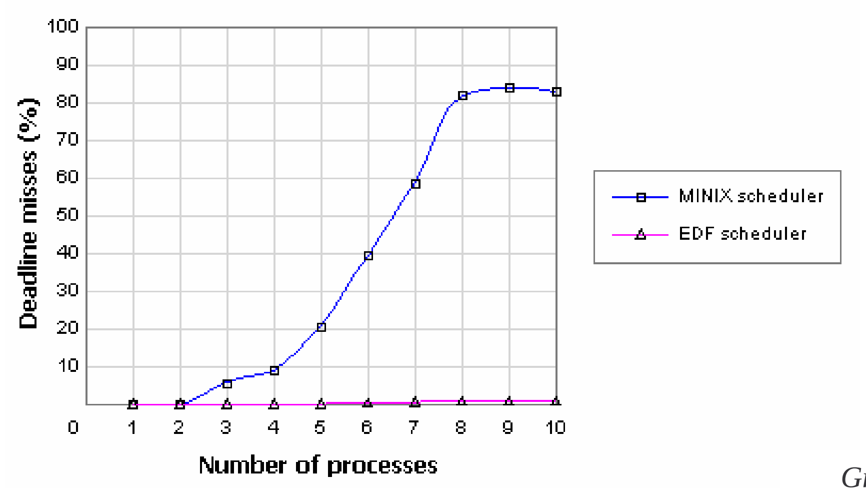
*8.7.2. Isolation from other multimedia processes*

Even though deadline overruns did occur in the last tests, the additional per-period time required by each process is small. In the next test, we show that our scheduler provides isolation of CPU-intensive multimedia processes from each other in general. The first stream reader is modified to overrun its execution time by 50% half of the time. The other processes are reset to having their actual execution times close to what they specified.



*Graph 8.24*

Graph 8.24 shows the resulting deadline miss percentage of the first process. 8.25 shows that of the remaining periodic processes (i.e. up to five). With few other processes present, the first process gets enough CPU time to finish its work within time at low priority. With many other processes present, it eventually ends up missing all of its deadlines. With PSCHED, the other periodic processes indeed end up with the same number of deadline misses as before. In contrast, MINIX' priority-based scheduler has more deadline misses for the correct processes in all cases. The percentage in graph 8.25 is somewhat lower, as a high number of deadline misses in graph 8.22 occur in the first process, which is not included in graph 8.25.



*Graph 8.25*

### 8.8. Network-based tests

The focus of this project has been on the path from the disk to the user application. However, we have also taken a brief look at the networking side of the complete problem. At least in theory, our test system should be able to achieve a network speed fairly close to the disk speed. While the disk achieves a little less than 15 MB per second with reasonably large block sizes, the 100MBit/sec Ethernet card should be able to achieve over 10 MB per second.

However, a series of tests on the test machine showed that even with perfect conditions, no more than roughly 5.5 MB/sec could be achieved at all. Scheduling traces revealed that during this time, the 'rtl8139' network driver, and to a lesser extent the 'inet' server, were using the CPU at nearly 100%. One of such scheduling traces has been included as appendix D.5. It shows five iterations of a "test" process performing a simple "read(); send();" loop to transfer data from a MFS instance to the network. The same behavior was observed with the "lance" network driver in VMware.

As could be expected, UDP performed a little better than TCP, even with many TCP connections active over a long time, but the 5.5 MB/sec rate was never exceeded. To make things even worse, the high CPU utilization of the network driver would require a tradeoff between even lower network speed and very little local data processing.

As a whole, this was somewhat of a disappointment. If the network is by far the biggest bottleneck, all efforts to optimize on disk speed would eventually be lost here for network-based multimedia applications. Under these conditions, no networked multimedia streaming server could ever be used to its maximum.

# Chapter 9. Conclusion

In this final chapter, we look back at our work (section 9.1) and consider a number of possible future projects based on this work (section 9.2).

## 9.1. Reflection

Our goals were to allow for fast, optimized and isolated multimedia data retrieval from the hard disk to user processes. To this end, we have made modifications to several parts of the MINIX 3 operating system. First, VFS was made nonblocking to allow multiple concurrent requests to file servers to take place. VFS was multithreaded, and a locking model was applied to it for both correct operation and fine-grained resource locking. On top of this, a read-only multimedia file system and file server were developed. The FS/driver protocol was extended to support multimedia requests. And, the generic disk driver library was extended to include a SCAN-EDF based rounds system and appropriate admission control. These additions allow for multimedia retrieval with soft guarantees even in the presence of other, nonmultimedia disk retrieval workloads. To limit interference from CPU workloads, a EDF-based periodic scheduler extension was implemented in the PM system server. In the process, a user-space cooperative threading library and a scheduling trace kernel extension were developed.

Our tests show that we indeed achieved our goals. The guaranted prefetching of MMFS minimizes the duration of individual read() calls. Even with minimal parameter tuning, we managed to get fairly close to the theoretical upper disk retrieval speed limit on our test machine. Performance of the multimedia retrieval was not affected by various other disk-intensive workloads. There was still room for nonmultimedia workloads to continue with fairly low response times during low load, and with reasonable response times during high load. Our scheduler was able to provide periodic execution time with low deadline miss rates and proper isolation from deadline overruns and CPU-intensive workloads.

However, there are some critical notes to make as well. Some of the points regarding VFS have already been made in section 3.6. In addition, the clean separation of the tasks between MMFS and the driver requires both a relatively complicated protocol and a strict rounding up of stream rates. Especially the latter puts a limit on the number of streams that is not necessary otherwise. A more pragmatic approach might lead to better overall performance, even though this might require stepping away from the SCAN-EDF based rounds system at all. Finally, if the benefits from the organ pipe approach are generally as low as section 8.5 suggests, then MMFS could be simplified by not sharing buffers between streams. These are all considerations that might have resulted in a different approach had we known them in advance.

We have provided only a generic support system for multimedia retrieval. We intentionally leave open the question how this system could be used in practice. In principle, many multimedia applications could benefit from the speed and soft guarantees provided by the system, although the limitations of this project make it less suitable to be used right away. Future extensions could tailor the system to specific application demands if necessary. In particular, addition of write support and better network performance would make the application area of the system much broader. The next section lists these points as well as other possible future work that could be based on our work.

## 9.2. Future work

### *9.2.1. Further VFS extensions*

As mentioned, the new unblocked VFS provides a base for several future extensions of MINIX. For example, it becomes possible to write a file server that is a client to a networked file system like NFS or Samba. This would require a new way for the file server to communicate with the ‘inet’ server: with the current approach, a process can not be both a privileged driver *and* a user process that opens a character-special file. However, dedicating as much as one bit in the request and reply types could provide the necessary separation of those requests. Asynchronous communication takes care of the rest. However, there may be other infrastructural problems that may need to be solved before implementation of an actual networked file system becomes possible.

Perhaps even more interesting would be support for file servers in the form of nonprivileged user processes. If such a user process is able to communicate with VFS like any other user process, then this would inherently allow the development of networked file systems as well. This would be similar to the Linux FUSE project [36] in some ways; if a compatible interface were to be developed, MINIX would instantly be able to use all the file systems that are implemented for FUSE. However, this would require some form of asynchronous communication with user processes at the very least, and that is currently not possible in MINIX.

Another possibility, in line with MINIX' general direction, would be the adaption of VFS to support recovery from crashing file system processes. Currently MINIX can recover from driver crashes, but if a file server crashes, VFS crashes as well and takes the whole system down with it. To a certain degree, the unblocked VFS now allows other processes and file systems to continue operating while one file server has crashed. VFS also retains enough state to resubmit outstanding requests after a replacement file server instance has been started. Whether full recovery from FS crashes can be achieved in some or all cases, and whether this is desirable in general, are still open questions.

### *9.2.2. MMFS and driver extensions*

This project has focused on reading from multimedia streams, although much of the required infrastructure for write support is already in place as well. MMFS could be extended or replaced in order to provide write support. This would require rethinking of both the file system format and the request issuing system, as both assume a consecutive file layout on disk – a known poor approach for read-write multimedia file systems.

Our implementation of the multimedia-supporting driver is also far from finished. Especially the deterministic block-based admission control system could be replaced by a much more sophisticated algorithm. That way, the hard disk could really be used to the fullest extent. This would possibly be based on many more metric values, perhaps obtained empirically from the disk. Our tests have indicated that there certainly is room for improvement. However, a new approach might require a new protocol and therefore also a rewrite of MMFS.

### *9.2.3. Zero copying*

One future improvement to multimedia performance that was unfortunately not feasible for this project, is the concept of zero-copying. As shown in figure 8.1, the FS spends a significant amount of CPU time on copying data from its own buffers to the user process. Many of our tests confirmed that this is indeed an issue. Copying could be avoided altogether if the driver would retrieve the data directly into a buffer accessible by the user process. This can even be combined with networking aspects, if the same buffer can also be used to send the data over the network without further copying.

This would have a number of implications, for example for the ability to share data blocks between processes. Sharing of buffers between user process is currently not supported at all, and solving all this may require significant infrastructural changes to file access. A possible solution could involve asynchronous file operations.

### *9.2.4. Multimedia networking*

As the tests revealed, the actual bottleneck of I/O from the disk via the user application to the network appears to be the network driver. This project has essentially provided the basic infrastructure necessary for multimedia retrieval from disk, so it would be interesting to complement it with an attempt to increase the effective speed of networking I/O, and provide multimedia-required guarantees for it. If successful, such a project would provide a direct improvement to the effectivity and usability of this project. At that point it would become interesting to write a new, or port an existing network streaming server that supports various media formats as well as the Real-time Transport Protocol (RTP). That could turn MINIX 3 into a full-fledged video-on-demand platform.

# References

[1] Al-Marri, J., Ghandeharizadeh, S.: An Evaluation of Alternative Disk Scheduling Techniques in Support of Variable Bit Rate Continuous Media. *Proceedings of the 6th International Conference on Extending Database Technology*, Mar. 1998.

[2] Anastasiadis, S.V., Sevcik, K.C., Stumm, M.: Server-Based Smoothing of Variable Bit-Rate Streams. *Proceedings of the ACM Multimedia Conference*, Oct. 2001.

[3] Anderson, R., Osawa, Y., Govindan, R.: A File System for Continuous Media. *ACM Transactions on Computer Systems*, Vol. 10, No. 4, pp. 311-337, Nov. 1992.

[4] Buddhikot, M., Parulkar, G.M., Cox Jr., J.R.:Distributed layout, scheduling, and playout control in a multimedia storage server. *Proceedings of the 6th International Workshop on Packet Video*, Sept. 1994.

[5] Chen, M.-S., Kandlur, D.D., Yu, P.S.: Storage and Retrieval Methods to Support Fully Interactive Playout in a Disk-Array-Based Video Server. *ACM Multimedia Systems Journal*, Vol. 3, No. 3, pp. 126-135, July 1995.

[6] Chiueh, T., Vernick, M.: An Empirical Study of Admission Control Strategies in Video Servers. *Proceedings of the 1998 International Conference on Parallel Processing*, Aug. 1998.

[7] Dan, A., Sitaram, D.: A Generalized Interval Caching Policy for Mixed Interactive and Long Video Environments. *Proceedings of SPIE Multimedia Computing and Networking Conference*, Jan. 1996.

[8] Dimitrijevic, Z., Rangaswami, R.: Quality of Service Support for Real-time Storage Systems. *Proceedings of the international IPSI-2003 Conference*, Oct. 2003.

[9] Engelschall, R.S.: Portable multithreading - the signal stack trick for user-space thread creation. *Proceedings of the USENIX Annual Technical Conference*, June 2000.

[10] Garcia-Martinez, A., Fernandez-Conde, J., Vina, A.: Efficient Memory Management in VoD Servers. *Computer Communications Journal*, Vol. 23, No. 3, pp. 253-266, Feb. 2000.

[11] Garofalakis, M.N., Özden, B., Silberschatz, A.: On periodic resource scheduling for continuous media databases. *The VLDB Journal*, Vol. 7, No. 4, pp. 206-225, Dec. 1998.

[12] Gemmell, D.J., Vin, H.M., Kandlur, D.D., Rangan, P.V., Rowe, L.A.: Multimedia Storage Servers: A Tutorial. *IEEE Computer Journal*, Vol. 28, No. 5, pp. 40-49, April 1994

[13] Haskin, R.: Tiger Shark - a scalable file system for multimedia. *IBM Journal of Research and Development*, Vol. 42, No. 2, pp. 185-197, March 1998.

[14] Leslie, I., McAuley, D., Black, R., Roscoe, T., Barham, P., Evers, D., Fairnbairns, R., Hyden, E.: The Design and Implementation of an Operating System to Support Distributed Multimedia Applications. *IEEE Journal on Selected Areas in Communications*, Vol. 14, No.

7, pp. 1280-1297, Sept. 1996.

[15] Liu, C.L., Layland, J.W.: Scheduling Algorithms for Multiprogramming in a Hard Real Time Environment. *Journal of the ACM*, Vol. 20, No. 1, pp. 32-42, Jan. 1973.

[16] Lougher, P., Shepherd, D.: The Design of a Storage Server for Continuous Media. *The Computer Journal*, Vol. 36, No. 1, pp. 32-42, 1993.

[17] Makaroff, D.J. Neufeld, G.W., Hutchinson, N.C.: An Evaluation of VBR Disk Admission Algorithms for Continuous Media File Servers. *Proceedings of the 5th ACM International Conference on Multimedia*, Nov. 1997.

[18] Martin, C., Narayanan, P.S., Özden, B., Rastogi, R., Silberschatz, A.: The Fellini Multimedia Storage Server. Journal of Digital Libraries, 1997. In: Chung, S. M. (Ed.), *Multimedia Information Storage and Management*, Kluwer Academic Publishers, Ch. 5, Aug. 1996.

[19] Mercer, C.W., Zelenka, J., Rajkumar, R.: On Predictable Operating System Protocol Processing. *Technical Report CMU-CS-94-165*, School of Computer Science, Carnegie Mellon University, May 1994.

[20] Nerjes, G., Muth, P., Paterakis, M., Romboyannakis, Y., Triantafillou, P., Weikum, G.: Scheduling Strategies for Mixed Workloads in Multimedia Information Servers*. Proceedings of the 8th International Workshop on Research Issues in Data Engineering*, Feb. 1998.

[21] Nieh, J., Hanko, J.G., Northcutt, J.D., Wall, G.A.: SVR4 UNIX Scheduler Unacceptable for Multimedia Applications. *Proceedings of Workshop on Network and Operating System Support for Digital Audio and Video*, Nov. 1993.

[22] Özden, B., Rastogi, R., Silberschatz, A.: Buffer Replacement Algorithms for Multimedia Storage Systems. *Proceedings of the IEEE International Conference on Multimedia Computing and Systems*, June 1996.

[23] Plagemann, T., Goebel, V., Halvorsen, P.: Operating System Support for Multimedia Systems. *Computer Communications Journal*, Vol. 23, No. 3, pp. 267-289, Feb. 2000.

[24] Reddy, A.L.N., Wyllie, J.: Disk Scheduling in a Multimedia I/O System. *Proceedings of the ACM Multimedia Conference*, Aug. 1992.

[25] Rogina, P., Wainer, G.: New Real-Time Extensions to the MINIX operating system, *Proceedings of International Conference on Information Systems Analysis and Synthesis*, Aug. 1999.

[26] Romboyannakis, Y., Nerjes, G., Muth, P., Paterakis, M., Triantafillou, P., Weikum, G.: Disk Scheduling for Mixed-Media Workloads in a Multimedia Server. *Proceedings of the ACM Multimedia Conference*, Sept. 1998.

[27] Shenoy, P.J., Goyal, P., Vin, H.M.: Architectural Considerations for Next Generation File Systems. *Proceedings of the ACM Multimedia Conference*, Nov. 1999.

[28] Shenoy, P.J., Pawan, G., Rao, S.S., Vin, H.M.: Symphony: An Integrated Multimedia File

System. *Proceedings of SPIE/ACM Conference on Multimedia Computing and Networking*, Jan. 1998.

[29] Shenoy, P.J., Vin, H.M.: Cello: A Disk Scheduling Framework for Next Generation Operating Systems. *Proceedings of the ACM Sigmetrics Conference*, June 1998.

[30] Stoica, I., Abdel-Wahab, W., Jeffray, K.: On the Duality between Resource Reservation and Proportional Share Resource Allocation. *Proceedings of SPIE Multimedia Computing and Networking Conference*, Feb. 1997.

[31] Tanenbaum, A.S.: *Modern Operating Systems*. Prentice Hall, 2nd edition, 2001.

[32] Tanenbaum, A.S., Woodhull, A.S.: *Operating Systems Design and Implementation*. Prentice Hall, 3rd edition, 2006.

[33] Wolf, L.C., Burke, W., Vogt, C.: Evaluation of a CPU Scheduling Mechanism for Multimedia Systems. *Software – Practice and Experience*, Vol. 26, No. 4, pp. 375-398, April 1996

[34] Yu, P.S., Chen, M.-S., Kandlur, D.D.: Grouped Sweeping Scheduling for DASD-based Multimedia Storage Management. *ACM Multimedia Systems Journal*, Vol. 1, No. 3, pp. 99-109, 1993.

[35] FUSE: Filesystem in Userspace. http://fuse.sourceforge.net/

[36] MINIX 3. http://www.minix3.org/

# Appendix A. Header files

## A.1. The ‘minix/systhread.h’ header file

```
#ifndef _SYSTHREAD_H
#define _SYSTHREAD_H

#include <ansi.h>

typedef int _systhread_thread_t;

typedef struct {
  _systhread_thread_t head;
  _systhread_thread_t tail;
} _systhread_queue_t;

typedef struct {
  _systhread_queue_t queue;
  _systhread_thread_t owner;
} systhread_mutex_t;

typedef _systhread_thread_t systhread_event_t;

typedef int (*systhread_main_t)(void);

typedef void (*systhread_proc_t)(void *);
typedef void *systhread_param_t;

typedef struct _systhread_store_t {
  struct _systhread_store_t *next;
  systhread_proc_t proc;
  systhread_param_t param;
} systhread_store_t;

_PROTOTYPE( int systhread_init, (unsigned int nthreads,
            unsigned int stacksize, systhread_main_t mainproc) );

_PROTOTYPE( void systhread_start, (systhread_proc_t proc,
            systhread_param_t param, systhread_store_t *store) );
```

```
_PROTOTYPE( void systhread_yield, (void) );
_PROTOTYPE( void systhread_yield_all, (void) );

_PROTOTYPE( void systhread_mutex_init, (systhread_mutex_t *mutex) );
_PROTOTYPE( void systhread_mutex_lock, (systhread_mutex_t *mutex) );
_PROTOTYPE( int systhread_mutex_trylock, (systhread_mutex_t *mutex) );
_PROTOTYPE( void systhread_mutex_unlock, (systhread_mutex_t *mutex) );

_PROTOTYPE( void systhread_event_init, (systhread_event_t *event) );
_PROTOTYPE( void systhread_event_wait, (systhread_event_t *event) );
_PROTOTYPE( void systhread_event_fire, (systhread_event_t *event) );

#endif /* _SYSTHREAD_H */
```

## A.2. The ‘sys/queryfs.h’ header file

```
#ifndef _QUERYFS_H
#define _QUERYFS_H

#ifndef _TYPES_H
#include <sys/types.h>
#endif

/* Query types */

#define QUERY_STATFS    1     /* the fstatfs() call */
#define QUERY_STREAMCTL 2     /* multimedia stream control */

/* Function prototype */

_PROTOTYPE( int queryfs, (int fd, int type, char *buf, int len)        );

#endif /* _QUERYFS_H */
```

## A.3. The ‘sys/streamctl.h’ header file

```
/* Data for queryfs(QUERY_STREAMCTL) call. */

#ifndef _STREAMCTL_H
#define _STREAMCTL_H

/* Operations */

#define STREAMCTL_OPEN  1     /* open a new multimedia stream */
#define STREAMCTL_SEEK  2     /* seek in a multimedia stream */
#define STREAMCTL_CLOSE 3     /* close a multimedia stream */

/* Buffer structure passed to queryfs() */

struct streamctl {
  int sc_op;                  /* the requested operation */
  unsigned int sc_rate;       /* stream rate in bytes/sec (for OPEN) */
  unsigned long sc_pos_lo;    /* low file position dword (OPEN/SEEK) */
  unsigned long sc_pos_hi;    /* high file position dword (OPEN/SEEK) */
};

#endif /* _STREAMCTL_H */
```

## A.4. The ‘sys/psched.h’ header file

```
#ifndef _PSCHED_H
#define _PSCHED_H

#ifndef _TYPES_H
#include <sys/types.h>
#endif

/* Periodic scheduling API prototypes */
/* All values are in microseconds! */

_PROTOTYPE( int psreserve, (unsigned long period, unsigned long exectime));
_PROTOTYPE( int psnext, (unsigned long drift) );

#endif /* _PSCHED_H */
```

# Appendix B. VFS locking scheme

## B.1. Locks acquired during VFS calls

The following table shows the *worst-case* locking behavior of the calls that can be initiated by processes other than VFS: the ‘do_’ calls that can be directly called by user processes, and the ‘pm_’ calls that are made on request by the PM server. Some calls have been split out to more narrowly define the worst-case locking for particular execution paths. Note that the table does not contain information about how many locks are acquired. For example, select() typically acquires readlocks on multiple filps in series.

| **VFS call** | **Vmnt lock** | **Vnode lock** | **Filp lock** | **Other locks** |
|---|---|---|---|---|
| do_access | R | - | - | - |
| do_chdir | R | R | - | - |
| do_chmod (chmod) | W+ (*) | W (*) | - | - |
| do_chmod (fchmod) | - | W | X | - |
| do_chown (chown) | W+ (*) | W (*) | - | - |
| do_chown (fchown) | - | W | X | - |
| do_chroot | R | R | - | - |
| do_close | - | O | X | CP |
| do_creat | W+ | W | X | CP |
| do_dup (dup) | - | R | X | - |
| do_dup (dup2) | - | O | X | CP |
| do_fchdir | - | R | X | - |
| do_fcntl (F_FREESP) | - | W | X | - |
| do_fcntl (not F_FREESP) | - | R | X | - |
| do_fslogin | - | - | - | - |
| do_fstat | - | R | X | - |
| do_fsync | R | - | - | - |
| do_ftruncate | - | W | X | - |
| do_getdents | - | R | X | - |
| do_getsysinfo | - | - | - | - |
| do_ioctl | - | R | X | - |
| do_link | W | - | - | - |

| | | | | |
|---|---|---|---|---|
| do_llseek | - | R | X | - |
| do_lseek | - | R | X | - |
| do_lstat | R | - | - | - |
| do_mkdir | W | - | - | - |
| do_mknod | W | - | - | - |
| do_mount | W+ | W | - | B |
| do_open (create) | W+ | W | X | CP |
| do_open (open, truncate) | R | W | X | CP |
| do_open (open only) | R | O | X | CP |
| do_pipe | R | W | X | - |
| do_queryfs | - | R | X | - |
| do_rdlink | R | - | - | - |
| do_read | - | R | X | CP |
| do_rename | W+ | - | - | - |
| do_rmdir | W+ | - | - | - |
| do_select | - | R | - | S |
| do_slink | W | - | - | - |
| do_stat | R | - | - | - |
| do_svrctl | - | - | - | - |
| do_sync | R | - | - | - |
| do_truncate | W+ (*) | W (*) | - | - |
| do_umask | - | - | - | - |
| do_umount | W+ | W | - | B |
| do_unlink | W+ | - | - | - |
| do_utime | R | - | - | - |
| do_write | - | W | X | CP |
| pm_dumpcore | - | O | X | CP |
| pm_exec | R | R | - | E |
| pm_exit | - | O | X | CP |
| pm_fork | - | - | - | - |
| pm_reboot | - | O | X | CP |
| pm_setgid | - | - | - | - |
| pm_setsid | - | - | - | - |
| pm_setuid | - | - | - | - |
| pm_unpause | - | - | - | - |

*Vmnt lock*: R = VMNT_READ, W = VMNT_WRITE, + = upgrade to VMNT_EXCL.

*Vnode lock*: R = VNODE_READ, O = VNODE_OPCL, W = VNODE_WRITE.

*Filp lock*: X = mutex.

*Other locks*: B = bfs lock, C = bfs lock for block-special vnodes, E = exec lock, P = VNODE_WRITE and select lock for pipe vnodes, S = select lock.

(*) All of do_chmod(), do_chown() and do_trunc() either writelock the vmnt and the vnode if one exists, or exclusively lock the vmnt otherwise, as discussed in subsection 3.5.6.

## B.2. Locks held during FS requests

The following table shows all locks that have been acquired by the current thread during calls made to a file system process.

| FS request | Vmnt lock | Vnode lock | Filp lock | Other locks |
|---|---|---|---|---|
| REQ_ACCESS | R | - | - | - |
| REQ_BREAD | - | R | X | B |
| REQ_BWRITE | - | W | X | B |
| REQ_CHMOD | W/X | W | - | - |
| REQ_CHOWN | W/X | W | - | - |
| REQ_CLONE_OPCL | R | (W) | - | - |
| REQ_CREATE | X | (O/W) | - | - |
| REQ_FLUSH | - | O/W | - | - |
| REQ_FTRUNC | (*) | W | X | - |
| REQ_GETDENTS | - | R | X | - |
| REQ_GETNODE | R | O/W | - | - |
| REQ_INHIBREAD | - | R | X | - |
| REQ_LINK | W | - | - | - |
| REQ_LOOKUP | R/W | - | - | - |
| REQ_MKDIR | W | - | - | - |
| REQ_MKNOD | W | - | - | - |
| REQ_MOUNTPOINT | X | (W) | - | B |
| REQ_NEWDRIVER | - | - | - | - |
| REQ_PIPE | R | (W) | - | - |
| REQ_PUTNODE | (*) | O | - | - |
| REQ_QUERYFS | - | R | X | - |
| REQ_RDLINK | R | - | - | - |
| REQ_READ | - | R | X | P |
| REQ_READSUPER | X | - | - | B |
| REQ_RENAME | X | - | - | - |
| REQ_RMDIR | X | - | - | - |
| REQ_SLINK | W | - | - | - |
| REQ_STAT | R (**) | R (**) | X (**) | - |

| | | | | |
|---|---|---|---|---|
| REQ_STIME | R | - | - | - |
| REQ_SYNC | R | - | - | - |
| REQ_TRUNC | W/X | - | - | - |
| REQ_UNLINK | X | - | - | - |
| REQ_UNMOUNT | X | - | - | B |
| REQ_UTIME | R | - | - | - |
| REQ_WRITE | - | W | X | P |

*Vmnt lock*: R = VMNT_READ, W = VMNT_WRITE, X = VMNT_EXCL.

*Vnode lock*: R = VNODE_READ, O = VNODE_OPCL, W = VNODE_WRITE.

*Filp lock*: X = mutex.

*Other locks*: B = bfs lock, P = VNODE_WRITE and select lock for pipe vnodes.

Parentheses indicate the vnode locking type used on the vnode that is created as a result.

(*) REQ_FTRUNC and REQ_PUTNODE can be called from a variety of contexts, some of which may hold a vmnt lock as well.

(**) REQ_STAT is used for both stat()/lstat() and fstat(); for the former two, the vmnt is locked; for the latter, the vnode and filp are locked instead.

# Appendix C. FS/driver protocol messages

### DEV_MM_READ_S and DEV_MM_WRITE_S

*Request*

| **Field** | **Field alias** | **Description** |
|---|---|---|
| m_type | | The request: DEV_MM_READ_S or DEV_MM_WRITE_S. |
| m2_i1 | DEVICE | The block device to read from or write to. |
| m2_i2 | IO_ENDPT | The endpoint owning the destination or source buffer. |
| m2_i3 | COUNT | The block size of this request, in bytes. |
| m2_l1 | POSITION | The low 32 bits of the byte position into the given device. |
| m2_l2 | HIGHPOS | The high 32 bits of the byte position into the given device. |
| m2_p1 | IO_GRANT | The requester's I/O grant for the destination or source buffer. |
| m2_s1 | MM_DEADLINE | The deadline of this request, clipped to 16 bits. |

*Reply*

| **Field** | **Field alias** | **Description** |
|---|---|---|
| m_type | | The reply: TASK_REPLY |
| m2_i1 | REP_ENDPT | The endpoint owning the destination or source buffer. |
| m2_i2 | REP_STATUS | The reply error code:<br>OK  Request fulfilled successfully.<br>EINTR  Request explicitly cancelled.<br>*(other)*  Same as for DEV_{READ\|WRITE}_S. |
| m2_i3 | REP_REQTYPE | The original request: DEV_MM_READ_S or DEV_MM_WRITE_S. |
| m2_p1 | REP_GRANT | The requester's I/O grant for the destination or source buffer. |
| m2_s1 | REP_MM_DEADLINE | The deadline of the request, clipped to 16 bits. |

### DEV_MM_CANCEL_S

*Request*

| **Field** | **Field alias** | **Description** |
|---|---|---|
| m_type | | The request: DEV_MM_CANCEL. |
| m2_i2 | IO_ENDPT | The endpoint that owns the buffer of the request to cancel. |
| m2_p1 | IO_GRANT | The grant of the request to cancel. |
| m2_s1 | MM_DEADLINE | The deadline of the request to cancel, clipped to 16 bits. |

*Reply*

| **Field** | **Field alias** | **Description** |
|---|---|---|
| m_type | | The reply: TASK_REPLY |
| m2_i1 | REP_ENDPT | The endpoint that owns the buffer of the request to cancel. |
| m2_i2 | REP_STATUS | The reply error code:<br>OK Request cancelled successfully.<br>EBUSY Request is in the current round; not cancelled.<br>ENOENT No matching request was found. |
| m2_i3 | REP_REQTYPE | The original request: DEV_MM_CANCEL_S. |
| m2_p1 | REP_GRANT | The grant of the request to cancel. |
| m2_s1 | REP_MM_DEADLINE | The deadline of the request to cancel, clipped to 16 bits. |

**DEV_MM_ADD**

*Request*

| **Field** | **Field alias** | **Description** |
|---|---|---|
| m_type | | The request: DEV_MM_ADD. |
| m2_l1 | MM_RATE | The rate of the stream to be admitted, in bytes per second. |
| m2_l2 | MM_BLOCK_SIZE | The block size used for block requests for this stream, in bytes. |
| m2_p1 | MM_STREAM | Opaque stream identifier; merely copied into the reply message. |

*Reply*

| Field | Field alias | Description |
|---|---|---|
| m_type | | The reply: TASK_REPLY |
| m2_i2 | REP_STATUS | The reply error code:<br>OK Stream admitted successfully.<br>EBUSY Unable to add stream at this time.<br>EINVAL Invalid stream rate or block size. |
| m2_i3 | REP_REQTYPE | The original request: DEV_MM_ADD. |
| m2_l1 | REP_MM_WAT | The work-ahead time to be used for this stream. |
| m2_l2 | REP_MM_TICKS | The minimum number of clock ticks between each two deadlines. |
| m2_p1 | REP_MM_STREAM | Opaque stream identifier from the request message. |

**DEV_MM_DEL**

*Request*

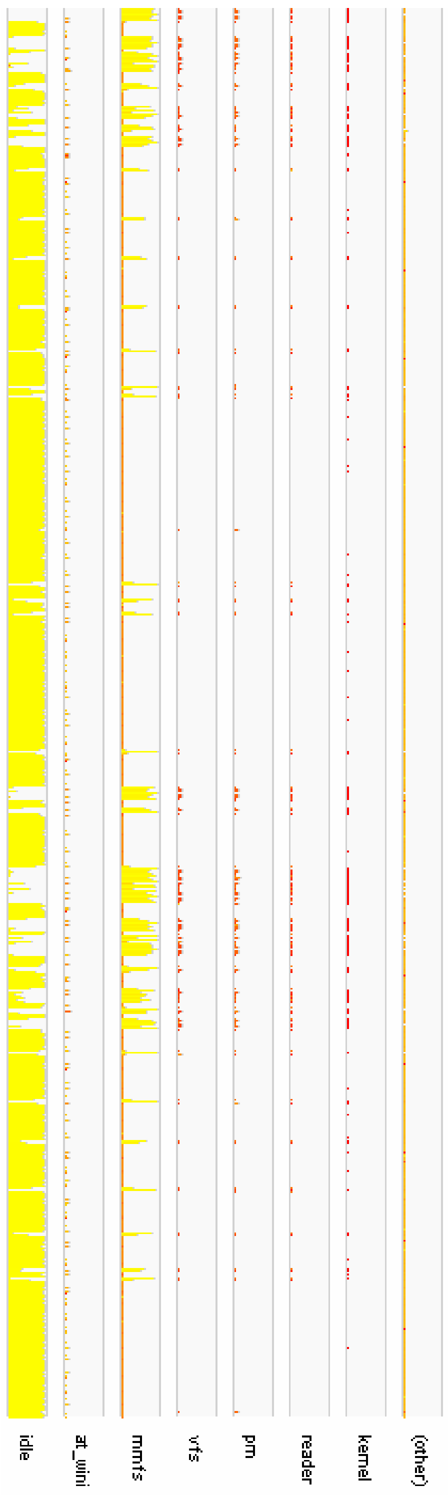
| Field | Field alias | Description |
|---|---|---|
| m_type | | The request: DEV_MM_DEL. |
| m2_l1 | MM_RATE | The rate of the stream to be deleted, in bytes per second. |
| m2_l2 | MM_BLOCK_SIZE | The block size used for block requests for this stream, in bytes. |
| m2_p1 | MM_STREAM | Opaque stream identifier; merely copied into the reply message. |

*Reply*

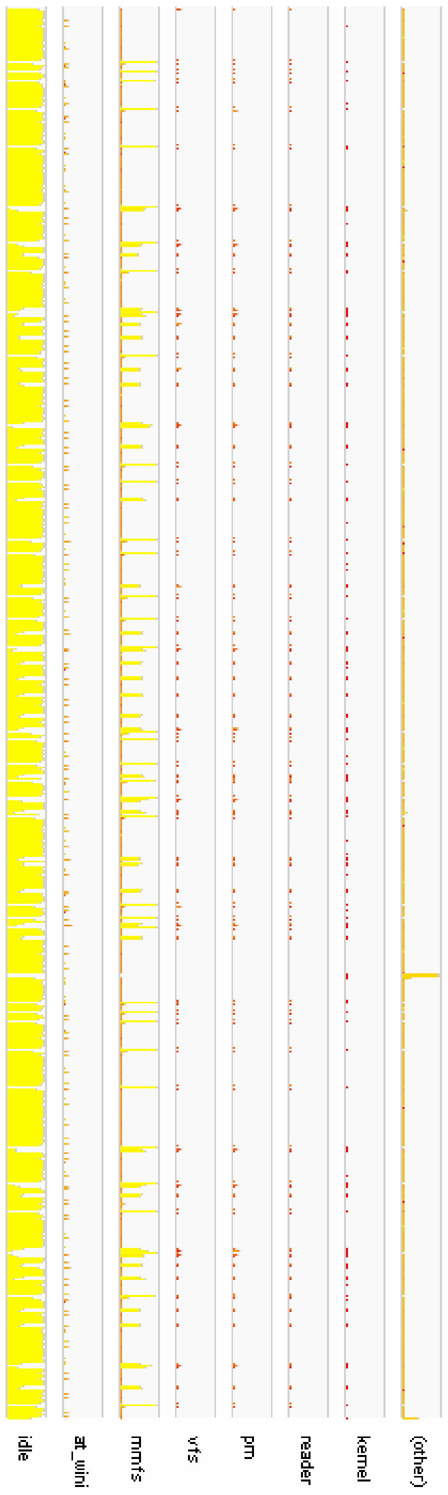
| Field | Field alias | Description |
|---|---|---|
| m_type | | The reply: TASK_REPLY |
| m2_i2 | REP_STATUS | The reply error code:<br>OK Stream deleted successfully.<br>EINVAL Invalid stream rate or block size. |
| m2_i3 | REP_REQTYPE | The original request: DEV_MM_DEL. |
| m2_p1 | REP_MM_STREAM | Opaque stream identifier from the request message. |

## Appendix D. Scheduling traces

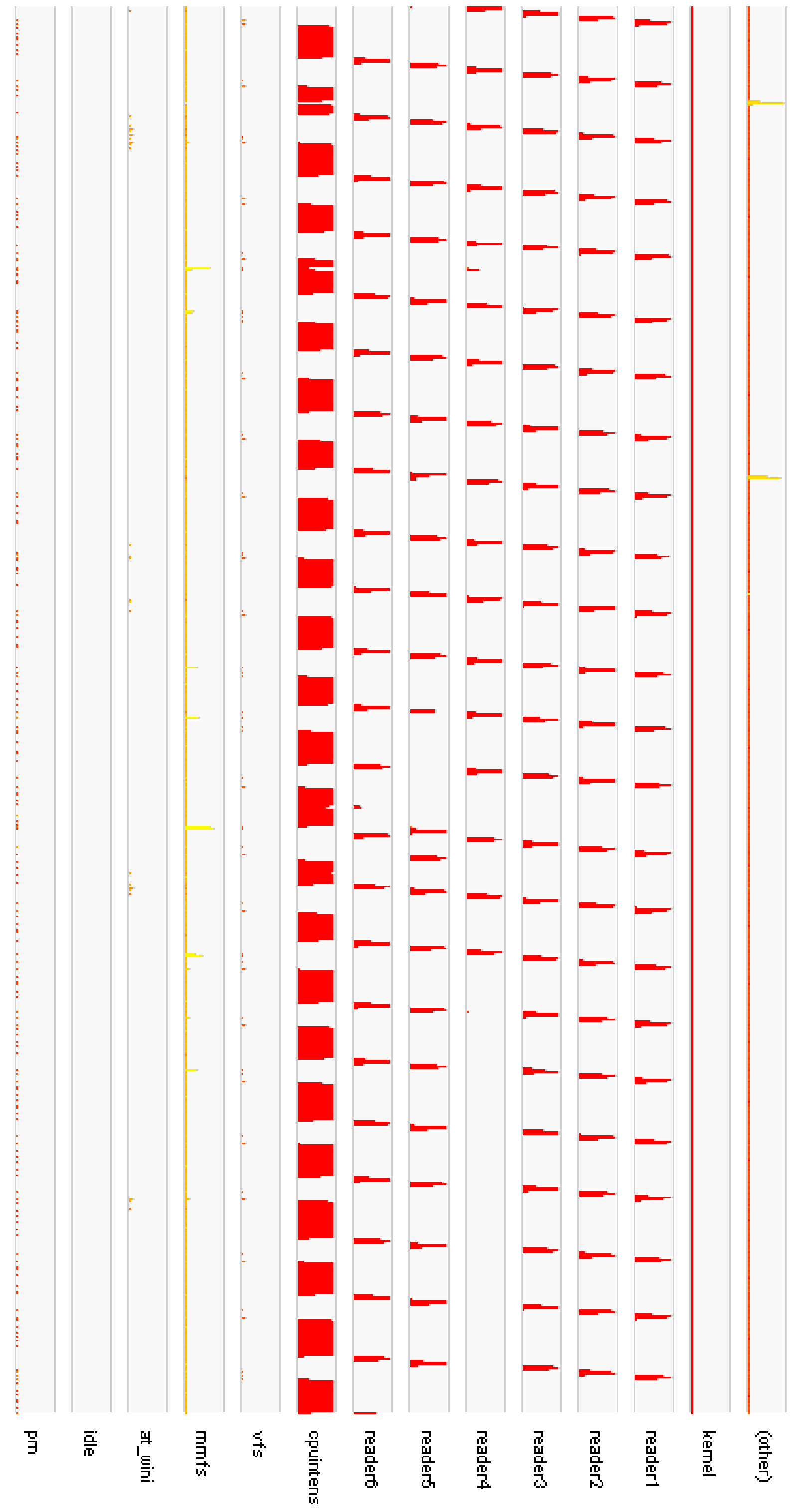
### D.1. Stream readers, condensed



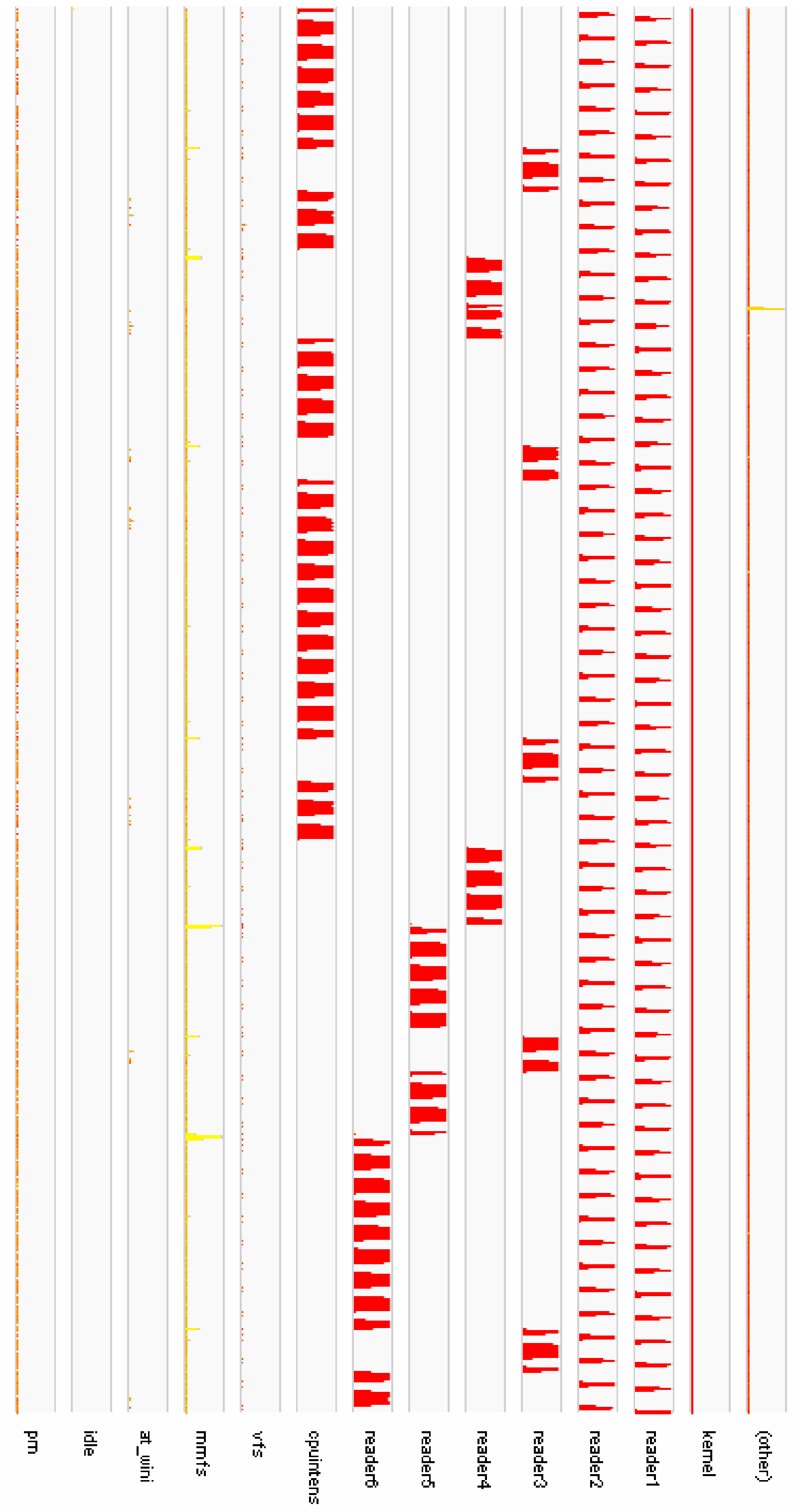
## D.2. Stream readers, spread out



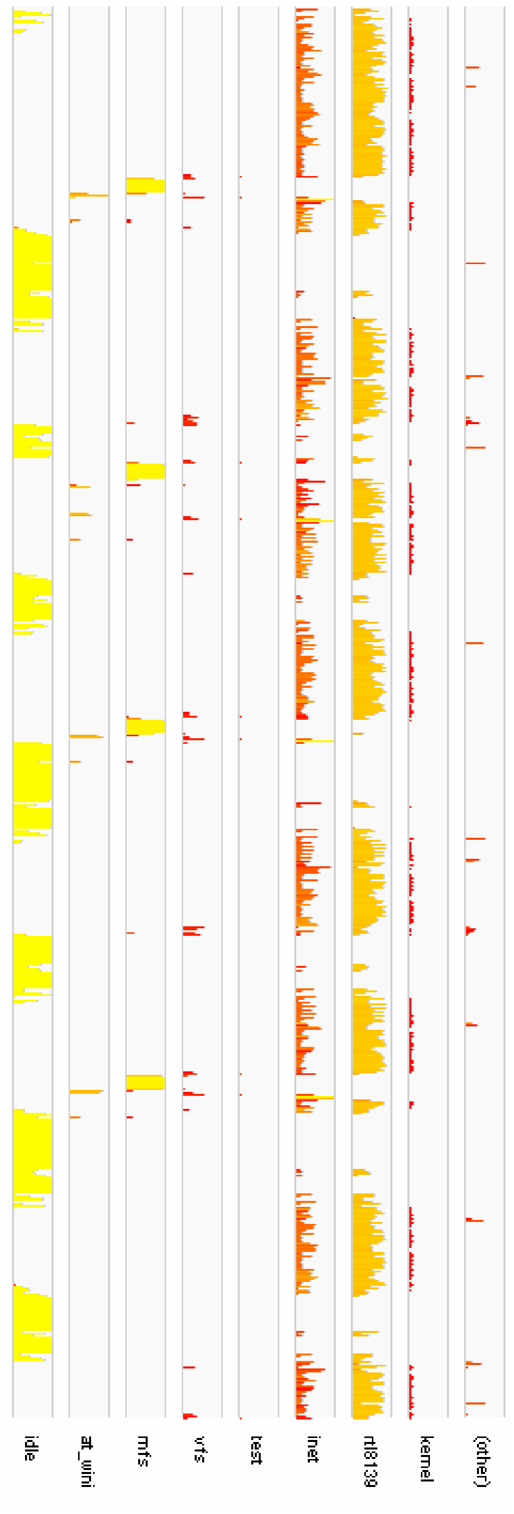
## D.3. Periodic processes, MINIX scheduling



## D.4. Periodic processes, PSCHED scheduling



## D.5. Network driver performance


idle
at_wini
mfs
vfs
test
inet
rtl8139
kernel
(other)