MinixVM: An Implementation of Virtual Memory in Minix 3

Carter Weatherly

710 Masters Project

The College of William & Mary

Submitted to the Department of Computer Science

of the College of William & Mary

in partial fulfillment of the requirements for the degree of

Masters of Science

April 24, 2009

Abstract

Minix 3 is the evolutionary development of Andrew Tanenbaum's original Minix microkernel. While the purpose of Minix 1 was to serve as an educational tool for teaching operating systems, the expressed purpose of Minix 3 is to provide a reliable, general-purpose operating system based on the microkernel architecture [3]. Because of its heritage, Minix 3 lacks many capabilities considered necessary for a modern, general-purpose operating system. One such major deficiency is the lack of a virtual memory implementation. Virtualizing memory is one of the most fundamental abstractions provided by a modern operating system. Without virtual memory, Minix 3 cannot transparently allocate more than the amount of physical memory, share discrete units of memory between address spaces, or map files into memory. This report presents MinixVM, a modification of the Minix 3.1.3a source to include virtual memory, and discusses the issues involved in integrating virtual memory into the microkernel architecture.

Contents

1 Introduction **7**

2 Minix 3 Design **9**

2.1 Microkernel Architecture 9

2.1.1 Advantages 9

2.1.2 Disadvantages 10

2.2 Minix 3 Components 13

2.2.1 Minix 3 Messaging 15

2.2.2 Minix 3 Image 17

2.3 Memory Management 18

2.3.1 Allocation 18

2.3.2 Executable Layout 18

2.3.3 Swapping 19

3 MinixVM Design **20**

3.1 Actors 20

3.2 Pager Server 20

3.2.1 Pager Server Data 21

3.2.2 Pager Server Build 22

3.2.3 Pager Server Limitations 23

3.3 MinixVM Operations 23

3.3.1 Pager Initialization 23

3.3.2 Exec Notification 25

3.3.3 Page Fault Handling 27

4 MinixVM Implementation **30**

4.1 Architectural Support 30

4.1.1 Memory Modes 30

4.1.2 Hardware Data Structures 30

4.1.3 Address Translation 31

4.1.4 Page Fault Handling . . . 32
4.2 Modification Summary . . . 33
4.2.1 Kernel Modifications . . . 33
4.2.2 PM Modifications . . . 34
4.2.3 VFS Modifications . . . 36
4.3 New Server Messages . . . 37
4.3.1 New Kernel Messages . . . 37
4.3.2 New PM Messages . . . 38
4.3.3 New VFS Messages . . . 40
4.3.4 Pager Messages . . . 41

5 Performance Analysis **43**
5.1 General Program Runtime Tests . . . 43
5.2 Sequential Scan Tests . . . 44
5.3 Summary . . . 46

6 Future Work **47**
6.1 Removing Segmentation . . . 47
6.2 System Server Reloading . . . 48
6.3 VM Extensions . . . 48
6.3.1 Page Swapping . . . 49
6.3.2 Copy-On-Write Page Sharing . . . 49
6.3.3 mmap() . . . 49
6.3.4 Dynamic Linking . . . 50

7 Conclusion **51**

8 References **52**

9 Appendix: MinixVM Patch Summary **53**

List of Figures

1 Minix 3 Architecture 14
2 Minix 3 Boot Image Layout 17
3 Minix 3 Boot Image Layout, with Pager 22
4 Pager Initialization Processing 24
5 Exec Notification Processing 26
6 Page Fault Processing 27
7 Segmented Address Translation 31
8 Paged Address Translation 32
9 General Program Runtime 44
10 Scan Performance 45

List of Tables

1 New Kernel Messages 38
2 New PM Messages 40
3 New VFS Messages 41
4 Pager Messages 42

1 Introduction

The world of operating systems is dominated by a single design. The *monolithic* kernel — where all of the operating system services are contained in a single binary running at the highest hardware-enforced privilege level — underpins most modern operating systems. This approach was a natural extension of the pre-operating system days where application programs interacted directly with the hardware (treating the operating system effectively as a "library" through which hardware resources are accessed). However, the monolithic design is not the only one available to operating system developers.

The microkernel architecture was originally conceived as a notion to simplify operating systems, which had become more and more complex as the capabilities of the hardware they controlled increased. The fundamental idea was to move as much code out of the kernel as possible, placing it in clearly delineated "servers" that run at a less privileged level than the kernel.

Minix 3 employs the microkernel architecture, and is the focus of this report. While Minix 3 purports to be a general-purpose operating system, it lacks many of the operating system primitives that developers have come to expect from modern operating systems. In particular, Minix 3 lacks a virtual memory (VM) implementation. This report presents MinixVM, a design and prototype implementation of VM in Minix 3.

The majority of this report is dedicated to the design and implementation of MinixVM; however, the ultimate goal of this report is to explore the difficulties of programming a microkernel within the context of MinixVM. As such, much of the presentation of MinixVM is interspersed with discussions on the microkernel architecture and how it sometimes forces awkward design and implementation decisions.

The remainder of this report follows this organization:

- **Section 2: Minix 3 Design**

  Presents a generic overview of the microkernel architecture; the design of Minix 3 as it relates to the design and implementation of MinixVM; and discusses some of

the advantages and disadvantages of the microkernel architecture.

- **Section 3: MinixVM Design**

  Describes the design of MinixVM, and how this design is shaped by the underlying design of Minix 3; describes the new system server, pager; and outlines the major operations that MinixVM must support.

- **Section 4: MinixVM Implementation**

  Describes the hardware architectural support for virtual memory; provides a summary of the material modifications made to existing Minix 3 components to implement MinixVM; and lists the new messages added to support MinixVM.

- **Section 5: Performance Analysis**

  Contains a brief performance analysis of the MinixVM implementation, focused on the overhead required to handle page faults.

- **Section 6: Future Work**

  Lists the items from the MinixVM design that are not complete in the MinixVM implementation and suggests extensions to the MinixVM design.

- **Section 7: Conclusion**

  Concludes the report, including a commentary on the viability of MinixVM and microkernels in general.

Also included is an Appendix, which summarizes the changes made to the Minix 3.1.3a source.

2 Minix 3 Design

This section describes the design of microkernels, the relevant portions of Minix 3 design, and how the design of the operating system imposes limitations on the implementation.

2.1 Microkernel Architecture

A monolithic kernel is frequently referred to as being "vertical stacked" in that it places layer upon layer of abstraction on the hardware in its creation of operating system services. A microkernel, on the other hand, is called "horizontally stacked" in that it provides the same services as a monolithic kernel, except that the majority of these services exist in userspace as multiple interdependent binaries.

Like a monolithic kernel, a microkernel includes a binary that runs at the highest privilege level, but this binary is designed to be as small as possible. Typically the "kernel" of a microkernel only contains the architecture-specific code necessary to boot the system and interact with the hardware, essential hardware drivers (e.g. the programmable interrupt controller or clock driver), minimal scheduling primitives, and an efficient mechanism that the remaining microkernel components use to communicate with each other (typically inter-process communication or IPC). The remainder of the services typically provided by an operating system are supplied by various "servers" which run in userspace. The level of access that these servers are given to hardware is determined by their function (this necessarily creates a set of trusted "system servers" without which the system would be effectively non-functional).

2.1.1 Advantages

The following is a list of the main advantages of the microkernel architecture over other operating system architectures. Where relevant, examples from Minix 3 are provided.

- Functional Separation

  From a pure design perspective, microkernels provide a perfect component-based architecture, where each component has independent and clearly-defined func-

tional responsibilities. This *should* simplify systems development and maintenance, much like the usage of shared libraries does for application development.

- Service Isolation

  In a monolithic kernel, all operating systems services operate in the same address space. If there is an error in any of this code, it could potentially corrupt the address space of the kernel and halt the system unexpectedly. In a microkernel, such a fault would be isolated to the offending userspace process.

- Fault Tolerance

  Since operating system services are both isolated and component-based in the microkernel architecture, they lend themselves naturally to the mechanisms of fault tolerance from distributed systems research. In fact, Minix 3 includes in its set of system services a "Reincarnation Server" which will restart a service if ceases to receive status notifications, making the protected components fault tolerant [2]. If this could be extended to the entire operating system (it cannot, see Section 2.1.2) then Minix 3 would be fault tolerant.

- Trusted Computing

  The microkernel architecture provides an enticing target for creating a trusted operating system, since current methods still rely heavily upon source code auditing and the core components of microkernels are small. Also, the isolation of system services to userspace limits the amount of damage an exploited service could cause.

2.1.2 Disadvantages

The following is a list of the main disadvantages of the microkernel architecture over other operating system architectures. Where relevant, examples from Minix 3 are provided.

- Operating System Development

  The primary disadvantage of the microkernel architecture is that, in its attempt to make it more difficult to propagate faults and define clear functional boundaries, it also complicates programming operating system functionality. Even the most straightforward concepts require drastically different (and significantly more complex) implementation than in a monolithic kernel. This is especially true with VM and will become more apparent in Section 3. This complexity in design also directly leads to difficulty in testing and debugging, effectively compounding the typical problems in developing an operating system with the difficulties involved in developing a distributed system. These issues, as well as the ready availability of operating systems based upon monolithic kernels have been largest barrier to microkernel adoption.

- Functional Coupling

  The microkernel architecture is ideally composed of loosely-coupled communicating components, but in practice this ideal is one of the first sacrificed when attempting to provide service implementations. This is clearly evident in Minix 3, where there is a hierarchy of "trusted" servers, or system servers (e.g. the process manager and virtual file system manager). Without the set of system servers, the operating system would not function. It is true that the components could be replaced (as is expected when the functionality has been abstracted), but only if the replacement components make the same assumptions about the processing of data — in a real system like Minix 3, components share data outside of their clearly defined APIs (e.g. the data copying necessary for true message handling described in Section 2.2.1). This extra-API processing prevents true functional separation in Minix 3 (likely, in any full-featured microkernel).

- Data Fracturing and Coupling

  Each service may need to manipulate the same abstraction in different ways. Data

copying between servers is relatively expensive, so in order to avoid that, microkernels tend to structure such "shared" data in a fashion similar to a relational database. All servers agree upon an identifier for the abstraction and link that with their own set of relevant data. This makes it difficult to manage these data structures, introduces data coupling between the servers, and adds further communications overhead to synchronize these data structures. An example of this in Minix 3 is the process: the kernel, PM, and VFS all maintain their own set of data relevant to their processing for each process in the system.

- IPC Overhead

  There have been claims, and indeed early evidence suggested, that microkernels suffer from inherent performance deficiencies compared to monolithic kernels, given the IPC overhead and the frequent kernel/user mode switches. These claims have largely been refuted by [6]; however, it is true that microkernels will always incur more overhead in offering services than monolithic kernels (due to the IPC). In the case of Minix 3 in particular, the server IPC mechanism simplifications (described in Section 2.2.1) ease the creation of system servers (all internal operations are guaranteed atomic); however, it comes with a decrease in performance. System servers are the brokers of shared resources, which means that multiplexing the access to these resources is impossible. There are, however, ways around these limitations, as will be discussed in Section 3.

- Fault Tolerance Myth

  It seems as if the microkernel architecture would provide fault tolerance in the way that distributed systems do, but in practice, this is very difficult to achieve in the general case. In fact, in Minix 3, this goal is only partially achievable. If a system server were to crash, it could theoretically be restarted; however, the new instance would lack all of the state information of the previous instance. Without this data, the system would cease to function (e.g. imagine having the process

table deleted from a running monolithic kernel). The alternative would be to treat each operation on a critical system server as a transaction, mirroring the data in a separate server (either as a data store or warm failover component). This has the disadvantage of (likely significantly) decreasing performance, and not actually solving the issue — it would turn a single system server failure to whole system failure into a double order failure (system server and transactional backing store failure). Because of these issues, Minix 3 only attempts to "reincarnate" device driver servers [2]. While device drivers account for the majority of system crashes in some modern operating systems [1], this partial implementation does not make Minix 3 a true fault tolerant operating system.

- Reverse Dependencies

  A reverse dependency, in the context of the microkernel architecture, occurs when the kernel depends upon a userspace component to complete the processing of a kernel-level operation. One such generic example would arise if the process scheduler were implemented in a userspace server. While the mechanisms to perform the scheduling exist in the kernel, if the kernel needed to run the scheduler, it would have to reach out to the userspace scheduler. This is a particularly insidious form of functional coupling, which may not be entirely unavoidable in the microkernel architecture (see Section 6.1 for a concrete example applicable to MinixVM).

2.2 Minix 3 Components

Minix 3 roughly follows the typical microkernel design described in Section 2.1. Figure 1 illustrates the layout of Minix 3 (image taken from [2]).

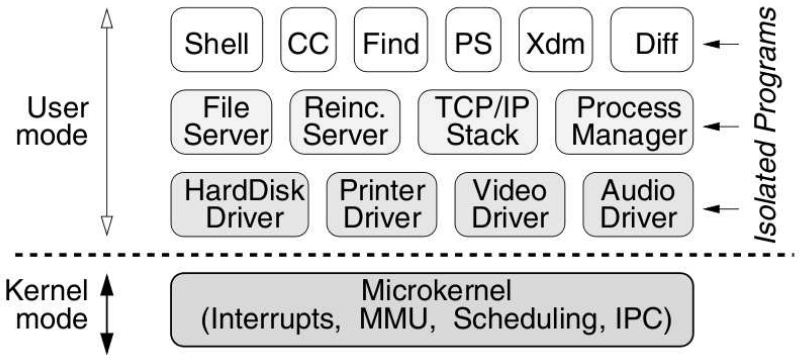


Figure 1: Minix 3 Architecture

Of the components listed, the following are directly relevant to the design and implementation of MinixVM (much more detail will be given in Section 3 and Section 4).

- **Microkernel**

  The microkernel is the only component of Minix 3 that runs in kernel mode (clear from Figure 1). It manages the interrupt vectors and exception handlers and so will be the first component in MinixVM to begin handling page faults. This portion of Minix 3 will be referred to as the "kernel" since it is the only code run in kernel mode. It is important to note that the kernel *cannot* dynamically allocate memory (memory allocation is controlled by the process manager).

- **Process Manager**

  The process manager, or PM, controls all aspects of a process in Minix 3, except for the actual scheduling or low-level management (all of which is in the kernel). All memory management in Minix 3 is controlled by PM (see Section 2.3 for why). Since almost all process-related messaging routes through PM, it is central to MinixVM.

- **File Server**

  The file server is typically a collection of at least two services: the virtual file system service (VFS) and the file system driver service. Almost all I/O operations route through VFS, and it is this service that MinixVM must interact with in order to perform its I/O requests (e.g. reading a page from disk into memory).

2.2.1 Minix 3 Messaging

The Minix 3 processes communicate through the Minix 3 IPC, or messaging system. The messaging system resides in the kernel, since it needs to be able to access arbitrary memory (copy the message contents from one address space to another). This system supports sending fixed length messages to any process equipped to receive messages (processes must block on a `receive()` in almost all cases for this condition to be true). Using fixed-length messages simplifies the messaging mechanism, since the kernel need not allocate arbitrary memory to deliver messages. As a further simplification, a receiver may process only one message at a time (queue size one). If a process attempts to send a message to another process which is handling a message already, the sender will block. With the fixed-length messages and queues, the messaging mechanism implements the simplest form of reliable message delivery; however, it does so at the cost of decreased messaging throughput and server utilization. Minix 3 works around the data-size limitation by sending pointers to larger data structures in the message and having the receiver use the kernel to copy the memory to its address space.

Processes that may receive messages (almost all) are identified by message endpoint numbers that the kernel assigns to each process. These are guaranteed to be unique — in Minix 3, the kernel simply adds a constant to the process ID to generate this number, except with the system servers where these endpoints are fixed. The endpoint number is required in all message sending API functions.

There are three types of message sending semantics in Minix 3, listed here by the API function call used to send a message using the specified semantics. There is only one way for process to receive a message: by calling the `receive()` API function and blocking until

message receipt (the so-called stop-and-wait semantics). Receiving processes may specify which processes are allowed to send messages to them by a parameter in the `receive()` call.

- `send()`

  Most messages in Minix 3 are sent using the `send()` API call, as it minimizes messaging overhead while still allowing for the delivery of data in the message. The `send()` API call is synchronous. This means that the sender will block until the kernel copies the message data from the sender's address space to the receiver's address space. If the receiver is not currently blocked on a `receive()` call, the sender will block until the receiver completes its processing and calls `receive()` again.

- `sendrec()`

  When processing requires a remote resource respond to a request to complete, `sendrec()` is used. The semantics of `sendrec()` are the same as traditional RPC — the process blocks while a remote resource processes a request, which looks like a regular function call in the original process, until the results of the operation are known to the original process. Using `sendrec()` incurs the most messaging overhead of all the sending methods; however, there are some circumstances where these semantics are required. Again, most Minix 3 communications are designed to use `send()` to avoid this overhead.

- `notify()`

  The only true asynchronous messaging mechanism in Minix 3, `notify()` is used rarely since it carries no payload. In terms if implementation, a `notify()` to a process merely sets a bit (depending upon the type of notification) in a bitmap contained in the process metadata. Including `notify()` in the messaging API is necessary for passing asynchronous events from the kernel to userspace (hardware interrupts and signals).

2.2.2 Minix 3 Image

As mentioned in Section 2.1, the microkernel design necessarily creates a hierarchy of servers offering system services. There are certain servers without which Minix 3 would not function. These services form the "system servers," a set of trusted (in that they have access to certain hardware resources) servers. Since the system servers are necessary for proper functioning, they are compiled into the image that is loaded by the boot monitor, which contains the kernel. (Note that while this deviates from the ideal of having a "bootstrap server" load required services, the microkernel architecture is maintained since the system servers may be replaced at compile-time.)

Figure 2 shows the typical output of a Minix 3 image, describing the layout of the microkernel and system servers in the boot image ('kernel' is the microkernel and the remaining components are the system servers).

```
  text     data      bss      size
 29968     5516    54036     89520  ../kernel/kernel
 27376     7612    83832    118820  ../servers/pm/pm
 48672    11792   105148    165612  ../servers/vfs/vfs
 21488     7448    47504     76440  ../servers/rs/rs
 32560     8012    16180     56752  ../servers/ds/ds
 30368     7748   111516    149632  ../drivers/tty/tty
 11120   535368    16908    563396  ../drivers/memory/memory
  9792     2268    68936     80996  ../drivers/log/log
 31792     5832  4990632   5028256  ../servers/mfs/mfs
  7008     2440     1356     10804  ../servers/init/init
------   ------   ------   -------
250144   594036  5496048   6340228  total
```

Figure 2: Minix 3 Boot Image Layout

The Minix 3 quirk of adding the system servers to the boot image becomes important when deciding what will be paged and when paging will be activated. This will be described more fully in Section 3.2.

2.3 Memory Management

2.3.1 Allocation

Minix 3 abstracts both the processor and memory using the same construct: the process. Most modern operating systems use the process to abstract the CPU, for scheduling purposes, but use some other finer-grained construct for abstracting memory. The reason for this that while whole-process memory allocation (where a process is given all of the memory needed to run during execution) simplifies the allocation process (one need only to find a large enough chunk of contiguous physical memory) it complicates the efficient use of memory. Whole-process memory allocation necessarily leads to internal fragmentation of memory, since processes rarely use their entire memory allocation, and also will eventually lead to external fragmentation of free memory "holes" that are difficult to allocate (an example of the NP-complete problem of online bin packing).

Minix 3 does attempt to prevent wasted memory through duplication of read-only code sections by only loading the text section of a binary into memory once per program, sharing that memory region amongst the multiple program copies. However, it does suffer from the aforementioned problems of internal and external fragmentation.

2.3.2 Executable Layout

To simplify loading, and due to the lack of VM support, Minix 3 executables must statically declare required memory resources at compile time (in the Minix-native a.out header). This constraint imposes a significant limitation upon dynamic memory allocation and runtime stack size, as they must compete for statically allocated memory. Miscalculating resources in this design must therefore result in either a processes being killed due to memory exhaustion or significant internal fragmentation (memory allocated that is rarely used). This design also makes porting existing programs to Minix 3 difficult, since most modern applications assume a nearly limitless supply of memory (presuppose the presence of VM).

2.3.3 Swapping

Minix 3 has limited support for swapping out processes to disk in order to alleviate memory pressure. This support is limited by the coarse allocation of memory in process-sized chunks. In order to swap any part of a process, the entire process must be swapped. This approach presents a very narrow window in which a system under significant memory pressure may remain operational before thrashing, since the disk I/O required to read/write an entire process would consume the majority of context-switch time.

3 MinixVM Design

The fundamental difference between Minix 3 and MinixVM is in the definition of the memory object. In Minix 3, memory objects are allocated to processes, in the size required by the process. In MinixVM, memory is allocated in context-independent, but process-related discrete pages. This further abstraction of memory demands further complexity in memory management, but also provides a disproportionate amount of flexibility in this management.

This section will discuss the design of MinixVM, presenting the pager server and the operations that must be implemented to support MinixVM.

3.1 Actors

In a VM implementation, there are typically five actors: the hardware, page fault handler, page allocator, process manager, and file system interface. The page fault handler is the routine called by the hardware when it detects a page fault; the page allocator is the logic that determines which page to allocate to the faulting process; the process manager coalesces the memory management information with the process; and the file system interface is used to read data in from disk, as in demand paging executables.

Most of these roles can be supported by modifying pre-existing Minix 3 components. The page fault handler can replace the default action in the kernel; the process manager (PM) can be modified to support routing page fault-related messages, and VFS can be modified to read executables on page fault. However, to fully support this design, a new system server is required: pager.

3.2 Pager Server

The pager server has simple and clearly-defined responsibilities: it manages the address spaces for all paged processes and determines whether or not a page fault represents a valid memory access for a given process. Note that pager must assume the role of memory manager since it must manipulate the physical address space directly.

3.2.1 Pager Server Data

In order to manage memory and handle page faults, pager requires a number of data structures.

- Process Metadata

  There is certain information, such as the process ID and endpoint number, that pager must know in order to be able to associate messages about processes with its own data structures. This means that pager joins the set of kernel, PM, and VFS, by containing its own data on running processes. This also means that pager must be notified whenever a process is created (see Section 3.3.2).

- Hardware Page Tables

  The hardware page tables are the data structures that are read directly by the MMU hardware on each memory access when paged mode is activated (see Section 4.1 for the architectural details). The inclusion of the hardware page tables in a userspace program is necessary since the kernel cannot do dynamic memory allocation, and is a good example of the awkwardness imposed upon the design by the microkernel architecture. Fortunately, the kernel can read from all physical memory – it need only be told where the base address of the hardware page table is located (see Section 3.3.2 for how this is done).

- Supplemental Page Tables

  The hardware page tables only constitute the pages allocated to the respective process that are currently loaded into memory. There are cases where pages may be allocated but not in memory, such as when a page is swapped out to disk. Pager must maintain this supplemental information in order to accurately manage memory and determine whether or not a memory access is valid.

3.2.2 Pager Server Build

As mentioned in Section 2.2.2, the Minix 3 microkernel is combined with the critical system servers into a single binary that is loaded by the boot monitor on boot. Figure 3 shows the layout of the boot image with the pager server included.

```
  text     data      bss      size
 30656     5732    54456     90844  ../kernel/kernel
 27744     8076    81904    117724  ../servers/pm/pm
 49520    12212   107548    169280  ../servers/vfs/vfs
 21488     7448    47552     76488  ../servers/rs/rs
 32560     8012    16180     56752  ../servers/ds/ds
 30368     7748   111516    149632  ../drivers/tty/tty
 11120   535368    16908    563396  ../drivers/memory/memory
  9792     2268    68936     80996  ../drivers/log/log
 31792     5832  4990632   5028256  ../servers/mfs/mfs
  6176     2176   118624    126976  ../servers/pager/pager
  7008     2440     1356     10804  ../servers/init/init
------   ------   ------   -------
258224   597312  5615612   6471148  total
```

Figure 3: Minix 3 Boot Image Layout, with Pager

It does not matter where pager is included in this binary, since system servers are not paged (they may run before the pager runs – see Section 3.2.3). In an ideal implementation, as in most modern operating systems, a small amount of bootstrap code would run before any of the system servers are run, enabling VM and loading them into memory in a way that pager expects. Minix 3 does not currently use this approach, as it does not need to (since memory slots are merely "occupied" by running programs).

It was the lack of support for laying out processes in the system image in a VM-friendly way that forced the creation of a separate pager server, rather than an extension to PM. As a separate server, it may be loaded into memory by PM (more on this in Section 6.3) in such a way that it could be paged. This would be necessary to support the potentially large data structures required for managing the separate virtual address spaces for each process.

3.2.3 Pager Server Limitations

The pager server has several limitations imposed upon it, mostly due to the underlying design of Minix 3.

- Certain processes cannot be paged

  Pager supports paged operation of all processes *not including* the system servers. This is a consequence of how system servers are loaded (as a bulk image, by the boot monitor) and the fact that pager does not run before the other system servers do (see Section 6.2 for a general solution to this problem).

- Limited heap space

  As mentioned in Section 3.2.2, since pager is a system server, it has the same limitation of sharing its heap space and stack space from a statically, compile-time allocated space.

3.3 MinixVM Operations

This section describes the control flow of the major operations that MinixVM must support: pager initialization, exec notifications, and page fault handling. Since a microkernel is a collection of loosely coupled functional components, these interactions will be presented as a sequence of messages. In some cases these are function calls, but in most cases, they are Minix 3 messages; the latter cases will be noted.

3.3.1 Pager Initialization

Pager initialization is both the most infrequent operation required by MinixVM (once at boot) and the operation requiring the most change to Minix 3 (see Section 4.2.2 for implementation details).

More detail on the messages can be found in Section 4.3.

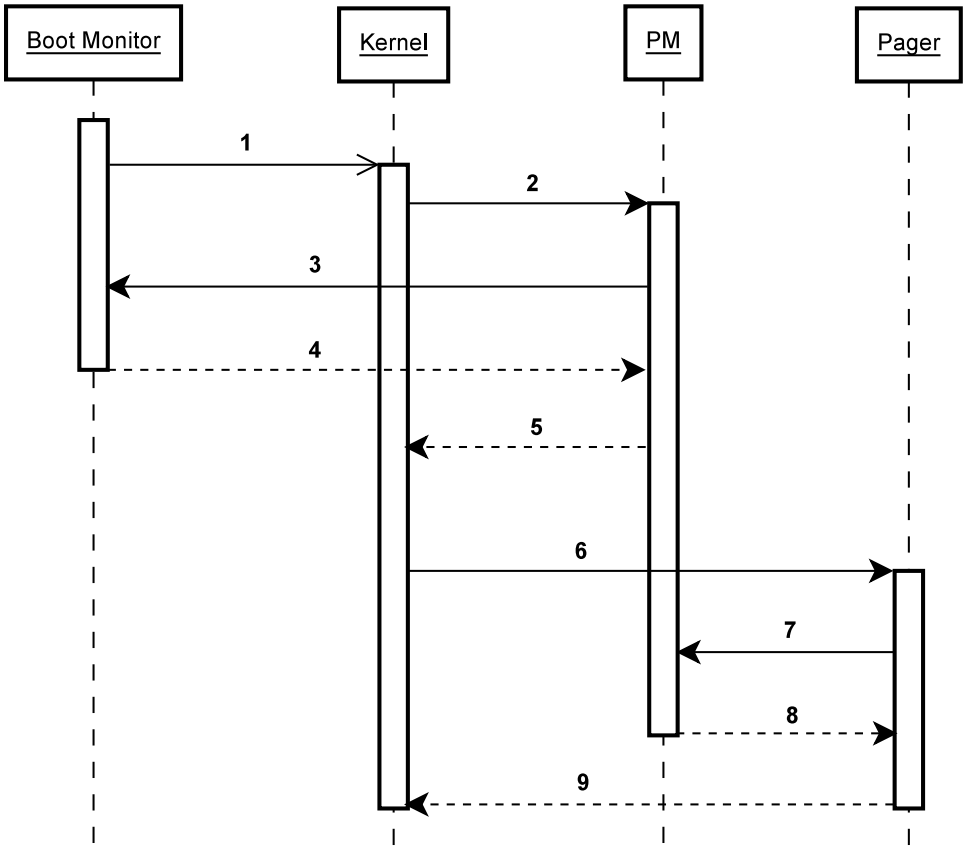


Figure 4: Pager Initialization Processing

1. At power on, the boot monitor takes control of the hardware, loads the boot image, and vectors control into the image (kernel).

2. The kernel performs some hardware initialization, and then transfers control to PM, which resides in the boot image.

3. PM begins initializing itself. As a part of this initialization, PM gets the map of physical memory from be boot monitor (since the boot monitor loaded the boot image, it knows which parts of memory are currently in use).

   Note that this step cannot be moved to Step 7 — PM's initialization must complete before pager's initialization begins. This is because as a part of PM's initialization, PM will redefine the segment descriptors for the system servers, effectively

changing the memory layout in such a way that it will not agree with the map contained by the boot monitor. Pager cannot simply query the boot monitor when it initializes.

4. The boot monitor responds with the physical memory map.
5. PM returns control to the kernel, since its initialization is complete.
6. Some time later (after allowing other system servers to initialize themselves), the kernel transfers control to pager's initialization code.
7. As a part of pager's initialization, it will read the updated memory map from PM (so that pager can initialize its memory management). This is done by sending PM a `PM_PAGER_INIT` message.
8. PM responds to pager with the memory map.
9. Pager returns control back to the kernel, now that it is done initializing. If no more system servers remain to be initialized, the kernel starts init.

3.3.2 Exec Notification

The following figure shows the message sequence required to implement the exec notification scheme in MinixVM. The exec notification is required so that pager knows to set up its process image once a new process is created. Note that the two other actors who must update their process images, the kernel and VFS, are notified with the `SYS_EXEC` and `PM_EXEC` messages, respectively.

Also note that a corresponding exit notification is not necessary. PM will send the messages to free the allocated memory once it receives the exit notification, and the old page data will simply be overwritten once pager receives another exec notification for the same process ID (this assumes PM is properly managing processes).

More detail on the messages can be found in Section 4.3.

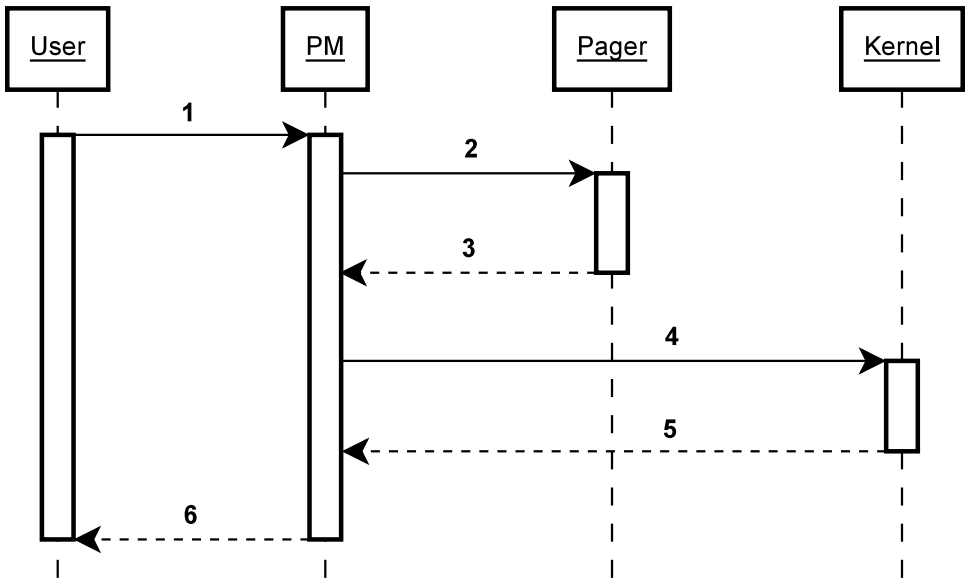


Figure 5: Exec Notification Processing

1. The user initiates an `exec()` by sending a message to PM (though a library-wrapped `exec()` call). This is a blocking call since the requesting process cannot continue until the exec is complete.

2. PM sends a `PAGER_EXEC` message to pager, to notify pager of the new process creation. This is a blocking call because PM needs the page directory base address from pager before continuing.

3. Pager responds to PM with the page directory base register of the new process.

   The response from the exec notification is also a convenient place to pass the page directory base address (see Section 4.1), to the kernel, since PM must already communicate with the kernel to finish the exec.

4. PM sends a `SYS_EXEC` message to the kernel (through the `sys_exec()` system call) to complete the exec. This call must be blocking since Step 1 was a blocking call (PM must synchronously respond to the requesting process).

5. The kernel responds with the exec status.

6. PM passes the exec status to the requesting process.

Note that only Steps 2 and 3 are new in MinixVM (the remainder were included either for completeness or because they were modified).

3.3.3 Page Fault Handling

The following figure shows the message sequence required to handle a page fault in MinixVM (only the go-path is shown). Page fault handing is at the core of the MinixVM implementation and is the most complicated new operation that it supports.

More detail on the messages can be found in Section 4.3.

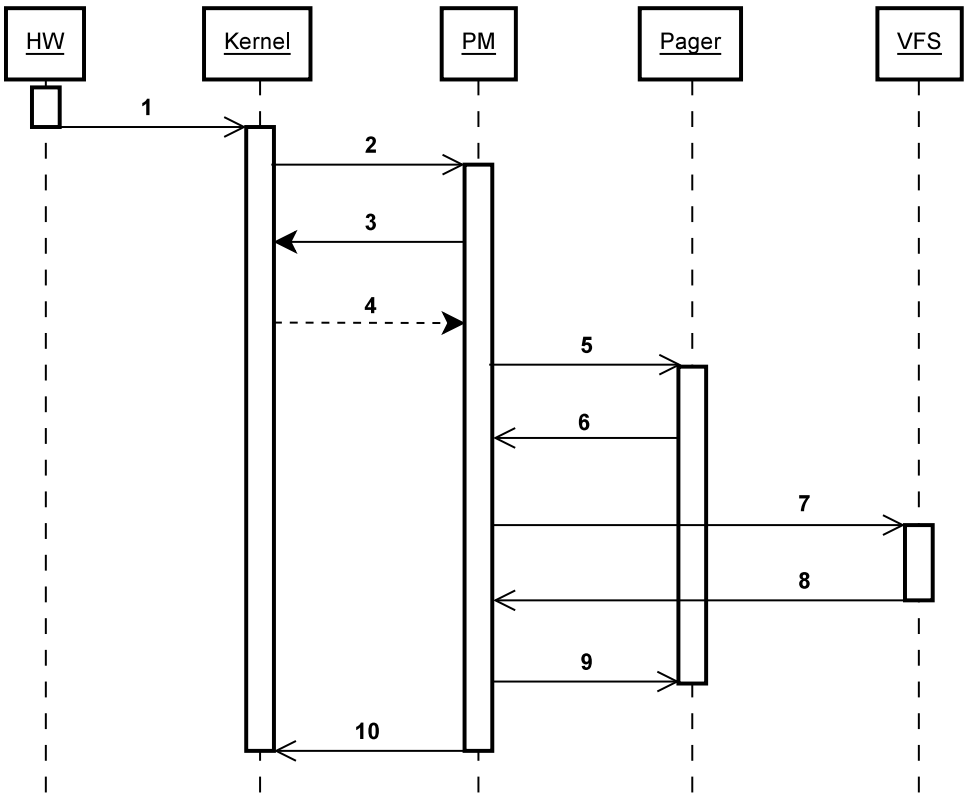


Figure 6: Page Fault Processing

1. The hardware senses a page fault exception and vectors control into the location specified by the kernel at boot (the page fault handler).

2. Kernel sends non-blocking notification (`HARD_INT` message) to PM.

   This is implemented by sending PM a `HARD_INT` (hardware interrupt) message, since it is the only available asynchronous messaging mechanism the kernel may use to contact PM.

3. PM processes the page fault notification from Step 2 and sends a `SYS_PAGEFAULT` message to the kernel to get the page fault details. PM blocks until it receives a response.

4. Kernel replies to PM with the page fault details.

5. PM sends a `PAGER_PGFCHECK` message to pager, to check the validity of the page fault. PM will block until pager receives the message (it will not wait for pager to fully process the message – see Section 2.2.1).

6. Pager sends PM a `PM_PGFCHECK_REPLY` message with the results of the validity check. If the result is negative, PM will report a bad page fault to the kernel.

7. PM sends a `PM_PGFAULT` message to VFS, requesting that it read the page from disk into memory. PM will block until VFS receives the message.

8. VFS sends a `PM_PGFAULT_REPLY` message to VFS, including the response status.

9. PM sends a `PAGER_PGFDONE` message to pager, to let it know the status of the page fault processing.

   This is required since pager would have know way of knowing whether or not Step 7 failed. If there is a failure after its last communication with PM, page tables and the like could become corrupted (e.g. list an entry that does not exist).

10. PM sends a `SYS_PAGEFAULT` message to the kernel, with the status of the page fault processing.

Note that in this design, the `send()` semantics were used as often as possible (Steps 5, 6, 7, 8, 9), since this maximizes the throughput without overly complicating the message passing sequence (see Section 2.2.1 for the details on `send()` semantics).

There is one case where the asynchronous notification mechanism must be used. In Step 2, where the kernel first communicates the page fault to PM. Since this notification is sent from the page fault handler, it is executed in interrupt context. This means that *no* progress will be made on the CPU until the handler terminates. Using `notify()` is the fastest mechanism Minix 3 offers to the kernel to communicate with PM.

4 MinixVM Implementation

This section describes the implementation of MinixVM, following the design listed in Section 3.

4.1 Architectural Support

Virtual memory is something that necessarily requires support from hardware to be feasible from a performance perspective, given that the address translation must occur on *every* memory access. Minix 3 only supports the Intel IA-32 architecture — this section provides a *brief* overview of what the operating system must do in order to use the hardware memory management support in the IA-32 architecture.

The information from this section was taken largely from Intel's comprehensive system programming guides ([4, 5]).

4.1.1 Memory Modes

When an IA-32 machine first boots, it is in so-called "real-mode," where memory addresses correspond directly to the physical addresses, in a limited 16-bit wide address space. To take advantage of the full 32-bit address space, and to provide protection of one chunk of code from another, IA-32 provides a "protected-mode." Protected-mode is entered soon after boot in Minix 3. Once in protected-mode all code segments must be given segment descriptors, which are later used for address translation and privilege checks, in hardware.

4.1.2 Hardware Data Structures

The following special purpose registers are important to the MinixVM implementation:

- `CR0` – Paging is enabled when bit 31 of this register is set.
- `CR2` – Upon receiving a page fault exception, the hardware will fill this register with the faulting linear address.

- `CR3` – This register contains the address for the base of the hardware page tables and must be changed on each process context switch.

4.1.3 Address Translation

The first step in address translation in IA-32 protected mode is to take a segment-relative address given to the hardware for memory access and convert into a linear address, which may be used to access memory. To perform this translation, the segmentation hardware uses the segment descriptors to translate addresses as summarized in Figure 7.

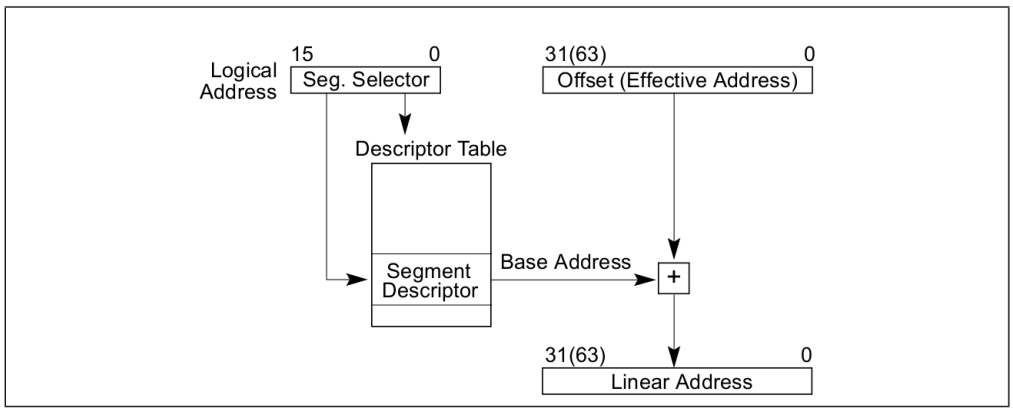


Figure 7: Segmented Address Translation

One important thing to note from this architectural addressing requirement is that the segment descriptor gives the base for the linear address. Because of this, binaries targeted for segmented architectures typically address with absolute addresses, starting at 0 (since the base for the segment will be added during any memory access).

IA-32 also provides paging hardware, which may be used to implement paging. Where segmentation is an absolute requirement in protected-mode, paging is not. However, paging hardware is necessary to implement VM efficiently (since the translation occurs on every memory access). The IA-32 architecture translates linear addresses to physical addresses as shown in Figure 8.

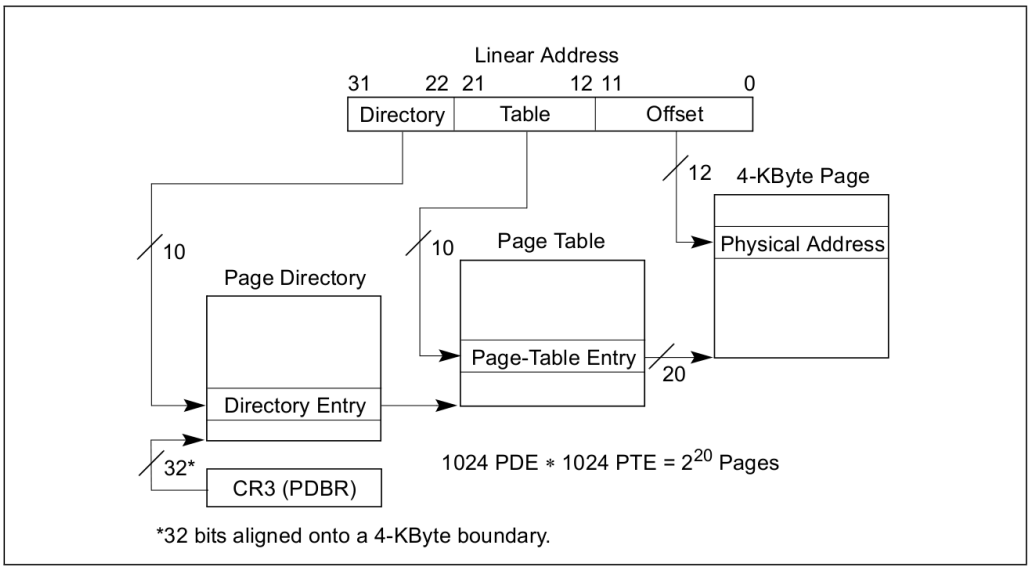


Figure 8: Paged Address Translation

The important thing to note from the paging support is that it is *always* used in conjunction with segmentation (since segmentation is activated when protected-mode is entered). This can be seen from Figure 8, where the input to the page translation hardware is the linear address produced by the segmentation hardware.

4.1.4 Page Fault Handling

A page fault exception is raised by the paging hardware when it is given a linear address for which there is no translation for the currently loaded page tables. In processing the exception, the hardware will load the `CR2` register with the faulting address and then vector into the page fault handler provided by the operating system. The page fault exception could be caused by either a legitimate access that does not have a translated address or an illegitimate access (no valid translation, write to a read-only page, or an unprivileged read). The latter case can only be positively determined by the software that manages processes. The paging hardware aids this determination by pushing a bitfield onto the interrupt frame containing flags for: present/access rights violation, read/write, kernel/user access.

4.2 Modification Summary

This section summarizes the modification necessary to existing Minix 3 components in order to implement MinixVM. The new server messages required are detailed in Section 4.3.

4.2.1 Kernel Modifications

- Register Access

  As mentioned in Section 4.1, on a page fault, the Intel architecture loads the `CR2` register with the linear address that raised the fault. An assembly routine `read_cr2` was added so that this value could be read in the page fault handler.

- Page Fault Exception Handler

  The default action upon receiving a page fault exception in Minix 3 is to send a `SIGSEGV` to the offending processes. This is consistent with the fact that a page fault exception should never be thrown in Minix 3 (since memory is segmented a wild memory access would result in a general protection fault). In MinixVM, the page fault handler records the page fault information (i.e. reads the faulting address out of the `CR2` register) and notifies PM of the page fault.

- Virtual Memory Mapping

  A few new functions were added to easily support the mapping of virtual addresses to physical addresses.

- Exec System Call

  The exec system call `SYS_EXEC` was modified to record the page directory base address into the kernel's process data structure (a new entry added for this purpose). The page directory base address must be in the kernel, so that the `CR3` register can be set to its value during a context switch.

- Process Switch

  As mentioned in the last item, the kernel must load the CR3 register with the base address of the hardware page tables. The process switch code was modified to make this change.

4.2.2 PM Modifications

Of all the Minix 3 components, PM was modified most heavily. This is because PM is designed to be the center of most MinixVM processing, in order to minimize the impact of the MinixVM implementation on Minix 3 processing (most of the memory management operations are routed through PM in Minix 3).

- Memory Management

  The largest change to PM was in removing memory management from the PM core. Memory management is only represented by allocation in Minix 3, so only allocation needed to be moved to pager. The reasoning behind this is simple: pager requires complete control of the physical address space. The three main memory allocation functions, allocate, free, and copy free memory holes were converted from function calls in PM to messages to pager: PAGER_MEM_ALLOC, PAGER_MEM_FREE, and PAGER_MEM_HOLES_COPY, respectively. This way, no other applications need be modified — their memory requests are simply routed through pager. In addition to message pass-through, the initialization of PM was changed significantly. Rather than reading the memory layout at boot and setting up the memory allocation data structures, it merely records this information for pager to process.

- Pager Initialization

  As mentioned in the previous point, memory management was moved to pager and memory initialization was deferred to pager as well. However, pager cannot

initialize the memory map until it has the bootstrap information, namely, where the kernel and system servers are loaded into memory (to avoid allocating that memory to other processes). PM is the first process to run at boot, and as a part of its initialization, it records the memory map (by querying the boot monitor) for pager. When pager first runs, it sends a `PM_PAGER_INIT` message to PM. PM responds with a pointer to this recorded information, which pager copies into its own address space, initializing memory much in the same way that PM does in Minix 3.

- Page Fault Processing

  Despite the new server, pager, in MinixVM the center of control remains in PM (this is clear from Figure 6). PM was modified to support this processing:

  * *Page Fault Exception* – As noted in Section 4.1.4, a page fault exception is routed through the kernel to PM. The handler for the `HARD_INT` notification was added to PM to begin the page fault processing. `HARD_INT`, however awkward in this scenario, was chosen since PM in Minix 3 does not receive any hardware interrupts, is unlikely to need to, and because that was the one remaining notification mechanism available to the kernel.
  * *Page Fault Validation* – In the page fault handling, PM is required to validate a memory access with pager. Since this is implemented using `send()` semantics, support for receiving pager's response was added to PM.
  * *Page Reading* – Once the access is determined valid, PM must request that VFS read the specified page from disk. Since this call is also implemented using `send()` semantics, support was added to process VFS's response.

- Exec Processing

  The processing within PM for executing a new process was slightly modified so that PM would not load the executable from disk (leaving the executable to be

demand-paged later), to add the exec notification message to pager (Section 3.3.2), and to support the new SYS_EXEC signature.

4.2.3 VFS Modifications

This section describes the VFS modifications, all of which were made to support VFS's role in page fault handling (Section 3.3.3).

- Executable Data Addition

  In order to support the MinixVM processing (namely reading pages from disk – see the next point), the VFS image of a process was modified to include a few more fields relating to executables.

  * `fp_exec_ino` – The inode of executable. Without this, the VFS-portion of the page fault handling would not know where the executable resides on the file system (or would need to perform an expensive lookup on the file name every page fault).
  * `fp_exec_fs_e` – The endpoint of the file system driver which controls the file system where the executable resides. To uniquely identify a file in Minix 3, the inode and the file system driver endpoint are needed, since all operations require knowing which driver to communicate with and since the inode is only unique on a single file system.
  * `fp_exec_hdrlen` – The length of the executable header. This is needed to properly calculate the offset into the executable file when reading portions of it into memory (see the following points).
  * `fp_exec_textlen` – The length of the text section for the executable.
  * `fp_exec_datalen` – The length of the initialized data section for the executable.
  * `fp_exec_bsslen` – The length of the uninitialized data section for the executable.

Adding these fields required modification of the exec-handling code, simply to make sure that the values were recorded the first time VFS reads the executable header.

- Page Reading

  Since VFS must read a page into memory from disk to support page fault handling (Section 3.3.3), code was added to VFS in order to do this. This code will request that the relevant file system driver read either a page-sized chunk or to the extent of the binary, whichever is smaller, into the page-aligned address it is given. The only twist to this scheme is in how the data section is handled. In the case where the page encompasses all initialized data, the entire page is read from disk. If the page encompasses all BSS data, the page is zeroed. If, however, the page contains a combination of initialized and uninitialized data, the initialized data is read in and the remainder of the page is zeroed. In order to make the determination of initialized/uninitialized data, this code requires the modifications to the VFS process metadata. VFS assumes that the memory write is legal, which it should be after the verification from pager.

4.3 New Server Messages

This section describes the new server messages and message parameters required by MinixVM implementation.

4.3.1 New Kernel Messages

Message	Parameters	Description
SYS_PAGEFAULT	PGF_TYPE	The type of the message (one of PGF_GET, PGF_DONE)
	PGF_PROCNR	The number of the faulting process
	PGF_ENDPT	The endpoint number of the faulting process

Continued on next page

Table 1 – continued from previous page

Message	Parameters	Description
	PGF_PADDR	The physical address associated with the virtual address
	PGF_VADDR	The virtual address (the faulting address)
	The SYS_PAGEFAULT system call message is used by the pager to both get the page fault information when it has been notified of a page fault by PM, and to report back to the kernel about the validity of the page fault.	

Table 1: New Kernel Messages

Note: new messages to the kernel are represented by system calls.

4.3.2 New PM Messages

Message	Parameters	Description
PM_PAGER_INIT	NONE	N/A
	The PM_PAGER_INIT message is sent by pager during its initialization, so that it may get PM's image of memory usage (since pager is the memory allocator). PM replies to pager with PAGER_INIT_REPLY	
PM_PGFAULT_REPLY	PM_PGFAULT_ENDPT	The endpoint number for the faulting process
	PM_PGFAULT_VADDR	The faulting (virtual) address
	PM_PGFAULT_BASE	The base address of the segment where the page fault occurred
	PM_PGFAULT_SEG	The segment within which the page fault occurred (text, data, or stack)

Continued on next page

Table 2 – continued from previous page

Message	Parameters	Description
	The PM_PGFAULT_REPLY message is sent by VFS once it is done processing the page fault (started when PM sends PM_PGFAULT to VFS).	
PM_PGFCHECK_REPLY	PM_PGFCHECK_PROCNR	The number of the process that raised the page fault
	PM_PGFCHECK_ENDPT	The endpoint number of the process that raised the page fault
	PM_PGFCHECK_VADDR	The (virtual) faulting address
	PM_PGFCHECK_ANS	The result of validity check
	The PM_PGFCHECK_REPLY message is sent by pager once it has completed determining the validity of of the page fault address access. This message is sent in response to PM sending PAGER_PGFCHECK to pager.	
PM_PAGER_EXEC_REPLY	PM_PAGER_EXEC_PDBR	The address for the base of the hardware page tables
	The PM_PAGER_EXEC_REPLY message is sent by pager once it has completed updating its process tables, in response to PM sending pager PAGER_EXEC.	
PM_PAGER_ALLOC_REPLY	PM_PAGER_ALLOC_BASE	The base address for the page or set of pages allocated
	The PM_PAGER_ALLOC_REPLY message is sent by pager once it has completed allocating memory in response to a PAGER_MEM_ALLOC message from PM. This functionality is in PM in Minix 3 and in pager in MinixVM, since pager must do the memory management.	

Continued on next page

Table 2 – continued from previous page

Message	Parameters	Description
PM_PAGER_ALLOC_REPLY	PM_PAGER_ALLOC_BASE	The base address for the page or set of pages allocated
	The PM_PAGER_ALLOC_REPLY message is sent by pager once it has completed allocating memory in response to a PAGER_MEM_ALLOC message from PM.	
PM_PAGER_HOLES_COPY_REPLY	PM_PAGER_HOLES_COPY_STATUS	The status code returned by the copy operation
	PM_PAGER_HOLES_COPY_BYTES	The number of bytes the hole list occupies
	PM_PAGER_HOLES_COPY_HI	The high watermark for allocation
	The PM_PAGER_HOLES_COPY_REPLY message is sent by pager once it has completed copying the hole list to the specified location (this is a throwback to the way Minix 3 manages memory and would not be present if VM were completely implemented). This functionality is in PM in Minix 3 and in pager in MinixVM, since pager must do the memory management.	

Table 2: New PM Messages

4.3.3 New VFS Messages

Message	Parameters	Description
PM_PGFAULT	PM_PGFAULT_ENDPT	The endpoint number for the faulting process
	PM_PGFAULT_VADDR	The faulting (virtual) address

Continued on next page

Table 3 – continued from previous page

Message	Parameters	Description
	PM_PGFAULT_BASE	The base address of the segment where the page fault occurred
	PM_PGFAULT_SEG	The segment within which the page fault occurred (text, data, or stack)
	The PM_PGFAULT message is sent by PM when it needs VFS to process a page fault (read in a page from disk at the faulting address.	

Table 3: New VFS Messages

4.3.4 Pager Messages

Message	Parameters	Description
PAGER_INIT_REPLY	PAGER_INIT_CHUNKS	A pointer to the memory chunk data structure in PM's address space
	The PAGER_INIT_REPLY message is sent by PM in response to pager sending a PM_PAGER_INIT message. Pager copies the data structure from PM's address space.	
PAGER_EXEC	PAGER_EXEC_PID	The process ID of the newly exec'ed process
	PAGER_EXEC_ENDPT	The endpoint number of the newly exec'ed process
	PAGER_EXEC_MEM	A pointer to the process memory map (in PM's address space)
	The PAGER_EXEC message is sent by PM to pager on an exec(), so that pager may populate its process image.	
PAGER_PGFCHECK	PAGER_PGFCHECK_PROCNR	The number of process to check

Continued on next page

Table 4 – continued from previous page

Message	Parameters	Description
	PAGER_PGFCHECK_ENDPT	The endpoint of the process
	PAGER_PGFCHECK_VADDR	The (virtual) faulting address to check
	PAGER_PGFCHECK_ANS	The result of the validity check
	The PAGER_PGFCHECK message is sent by PM to ask pager if a faulting address is valid or not.	
PAGER_PGFDONE	PAGER_PGFDONE_STAT	The page fault processing status
	The PAGER_PGFDONE message is sent by PM to notify pager of the status for the entire page fault process (so that pager may keep its data structures consistent.	
PAGER_MEM_ALLOC	PAGER_MEM_ALLOC_CLICKS	The number of clicks (pages) to allocate
	The PAGER_MEM_ALLOC message is sent by PM to ask pager to allocate the specified amount of memory.	
PAGER_MEM_FREE	PAGER_MEM_BASE PAGER_MEM_CLICKS	The base address of the pages to free The number of clicks (pages) to free
	The PAGER_MEM_FREE message is sent by PM to ask pager to free the specified amount of memory at the specified location.	
PAGER_MEM_HOLES_COPY	PAGER_MEM_HOLE	A pointer to a block of memory (in PM's address space) into which the hole list can be copied
	The PAGER_MEM_HOLES_COPY message is sent by PM to ask pager to copy the memory hole list to the specified location. (This is a throw-back to the way Minix 3 manages memory and would not be present if VM were completely implemented.)	

Table 4: Pager Messages

5 Performance Analysis

This section presents a brief, two-part performance analysis of the MinixVM prototype. The first part analyzes the performance impact of paging in several common scenarios. The second part examines the performance impact of paging as a function of the amount of memory paged (i.e. number of page faults handled) for a fabricated sequential-scan test.

For all of the tests, the POSIX-specified `time` utility (provided by Minix 3) was used to measure the amount of wall-clock time that elapsed from the beginning to the end of each test. Each result is the average of three different wall-clock measurements, and the measurement variance is small enough to be elided from the analysis. Also, for each test, *only the program tested is paged*. The MinixVM prototype is capable of co-existing with the segmented memory management scheme is such a way that one can specify a list of programs to be paged — all others use the segmented memory management scheme. For example, during the `bzip2` performance test, *only* the `bzip2` binary will be paged after being `exec`'ed.

5.1 General Program Runtime Tests

The general program runtime tests were designed to reflect a small set of common workloads, including compression, decompression, archiving, and compiling.

For all of the general program runtime tests excluding the `kbuild` test, the subject file was a 180 megabyte tarball containing 4242 files (the complete MinixVM source and supporting project files). For these tests, the timed operation is the default action of the program listed on the subject file (e.g. the `gunzip` test decompresses the subject tarball that was previously compressed with `gzip`). The `kbuild` test measures the amount of time required to build the system servers and bootable microkernel image from a clean MinixVM source tree. The program actually paged in this test was `cc`, the C compiler from the Amsterdam Compiler Kit (the compiler suite used to compile Minix 3).

Figure 9 shows the results from these tests:

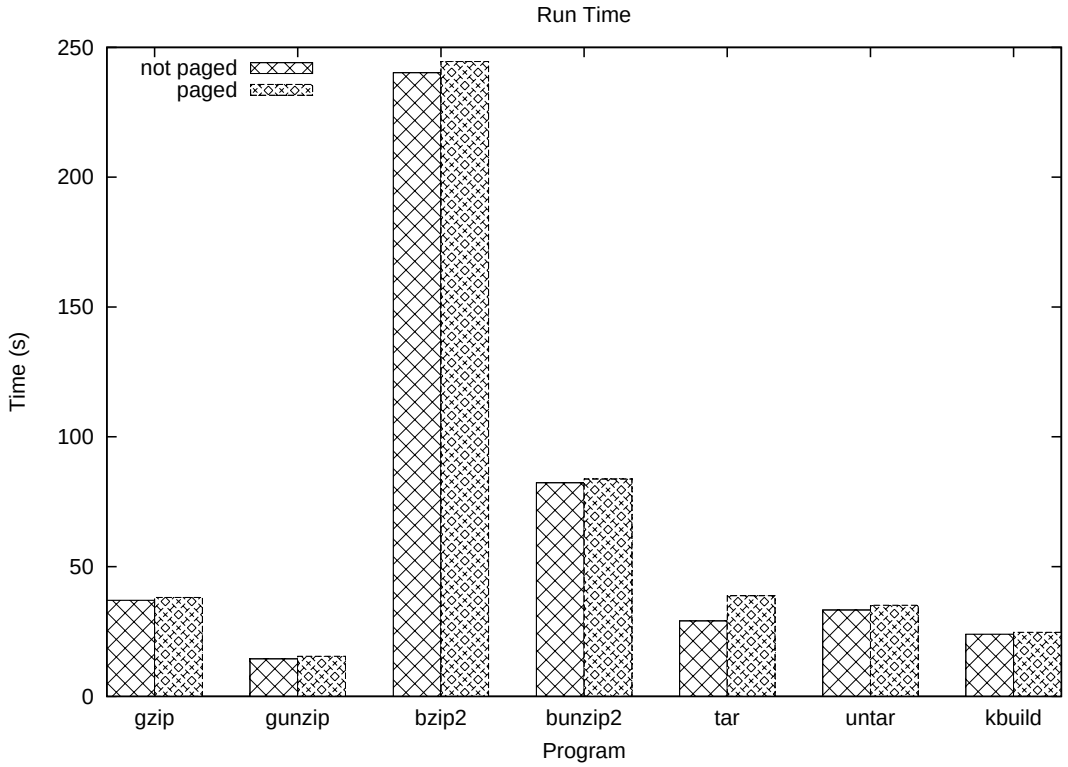


Figure 9: General Program Runtime

As would be expected, there is a measurable overhead with the paged versions of the tests. On average, the paged versions are 1.08 times slower than their non-paged counterparts.

Interesting to note is the significant difference in the performance for the `tar` test (1.33 times slower in the paged version). This is likely due to the memory access pattern of `tar` — one-use sequential. This is an effect that will be come more apparent in the binary scan test described in the next section.

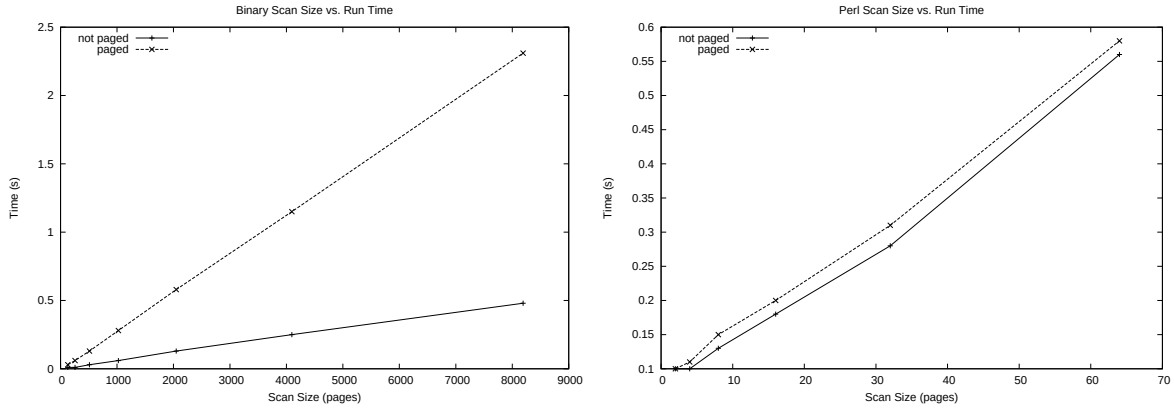
5.2 Sequential Scan Tests

While the tests of common workloads given in the previous section are informative, they fail to describe the performance overhead as a function of program memory requirements. To test this relationship, a set of sequential scan tests were created.

There are two types of the sequential scan test. Both tests follow the same algorithm:

statically declare an array of sizes ranging from 2 to 64 in power-of-two increments, initialize the array, reduce it to a scalar, and then print the result. The difference between the two tests is that the first (the binary test) is written in C, compiled to an executable, and then the entire executable is paged. The second test (the perl test), however, is written in perl and interpreted (the perl binary is paged in this test).

Figure 10 summarizes the results from these two tests:



(a) Binary Scan

(b) Perl Scan

Figure 10: Scan Performance

From Figure 10, it is clear that there is a far more significant performance impact for the binary test than the perl test. More specifically, on average, the paged version of the binary test is 4.7 times slower than the non-paged version (ranging from 3 to 6 times slower), where the paged version for the perl test is only 1.07 times slower. This is most likely an artifact of the fact that the second test is interpreted — the binary must load all of the statically allocated memory in unique pages, eventually, where the perl interpreter can reuse memory pool data to manipulate the array, resulting in fewer page faults. The slight difference in the performance of the paged and non-paged versions in the perl test is likely due to the overhead associated with the page faults that the paged version could not avoid.

Also of note is the observation that the runtime for the paged version of the binary test increases much faster than its non-paged counterpart. This is to be expected, since the paged

version must pay the cost of several expensive mode switches (in the form of the page fault handling) with each new page, whereas the non-paged version need only pay this cost once (during initial load).

5.3 Summary

It is not surprising that *all* paged versions of the tests in this section suffered a performance penalty due to the paging overhead. This overhead is not due to the extra processing required so much as the frequent mode switches (from user to kernel mode and vice versa) required by the page fault handling algorithm (described in Section 3.3.3). This is most apparent in binary test from Figure 10, where the runtime of the paged version grows in proportion to the number of page faults handled.

These data demonstrate that, because the mode switches are required by the microkernel architecture, a VM implementation in a microkernel must necessarily be slower than a comparable VM implementation in a monolithic kernel.

6 Future Work

Due to time constraints, a full MinixVM implementation was not completed. This section enumerates the largest outstanding issues as well as extensions that could be added on to MinixVM.

6.1 Removing Segmentation

The MinixVM implementation as presented in this report is not a complete VM implementation. In order to fully support VM, the address space of a process must be presented as the maximum possible (e.g. 4GB on a machine with 32-bit addressing), not limited by segment boundaries.

There are a number of issues that prevented the easy removal of segmentation:

- Binary Layout

  Binaries in Minix 3 are generated starting with address offset zero in each section. The Minix 3 loader depends upon the architectural segment descriptor to apply segment offsets when creating the linear address. Ordinarily, an operating system has an established loading convention, so that binaries have their sections tagged with addresses that the loader expects *in VM*. In the MinixVM implementation presented is this report, this is impossible, since segments encompass the entire program and these physical addresses are not known until runtime. (In a VM environment, it does not matter that two programs "load themselves at the same location" since they are loading at the same *virtual* addresses.)

  In order to support VM, either the binaries would need to be regenerated to have the agreed upon virtual addresses, or the binaries would need to be dynamically translated (addresses rewritten) at runtime. The former would probably be best approach, since it only occurs the one-time compilation penalty.

- Memory Copy Operations

  Minix 3 makes heavy use of memory copy operations, since its message system is only capable of sending fixed-length messages. When a message in Minix 3 has a payload that exceeds the fixed-length, it instead sends a pointer to the data in the sender's address space. The receiver uses a system call to have the kernel copy the data into the receiver's address space. This is straightforward to implement in segmented memory management, since a byte-by-byte copy is available. This process is *not* straightforward in a VM environment. First, the address would need to be translated to the physical pages (page by page). In MinixVM this must either occur in pager *or* the kernel must read pager memory (since that is where the page tables are located). If the memory copy refers to pages that are swapped out, the translation *must* occur in pager (since the hardware page tables are just the subset of pages allocated to a process that are currently in memory). Therefore, if swapping is enabled, MinixVM will have a reverse dependency between the kernel and pager — the kernel would depend upon a *userspace* program in order to perform a kernel operation. Without being able to do dynamic memory allocation in the kernel, this problem is not solvable in the general case. This is yet another example of where the the "ideal" microkernel architecture must be violated to provide even the most fundamental OS abstraction.

6.2 System Server Reloading

Ideally, the system servers would be reloaded into virtual memory address spaces at boot, by some bootstrap code, so that they could be paged like any other process.

6.3 VM Extensions

There are a number of extensions that naturally fall out of a VM implementation with minimal effort. All of these extensions are only valid in a MinixVM environment — the memory model of Minix 3 makes them impossible.

At the very least, pager should be reloaded in order to avoid issues with allocating space for the page table data structures. In lieu of reloading it in a paged manner, it would just have its stack section moved, to the same effect.

6.3.1 Page Swapping

Swapping refers to writing out pages from memory to disk, in order to increase the available amount of memory. This is an artificial increase, but critical to maintaining the abstraction that processes may use the entire space of available memory.

6.3.2 Copy-On-Write Page Sharing

Copy-On-Write (COW) page sharing is a technique of memory sharing by which a `fork()` does not necessarily result in the copying of all memory from the parent process to the child (as it does by default in Minix 3). Instead, pages are mapped into the page tables of both of the processes, used until one writes to a shared page. Once written to, both processes get their own copies. This technique results in efficient memory utilization in the common case where the child replaces its memory map with an `exec()` soon after the `fork()`.

6.3.3 mmap()

Memory mapping through `mmap()` is one of the core functionalities of the POSIX specification (also known as the Single Unix Specification) [8], and is missing from Minix 3 — because memory mapping is extremely difficult to implement in a purely segmented memory management scheme. Memory mapping is core to many modern applications, making it impossible to port applications from other environments (e.g. Linux) without having to rewrite portions of non-trivial application code. In some cases, it is impossible to replicate the functionality of memory mapping, making the porting impossible.

6.3.4 Dynamic Linking

Dynamic linking — reading common library code into memory as needed during execution — is a common technique used in modern operating systems to amortize the memory footprints of running programs and to reduce on-disk library storage. In order to reduce memory usage, dynamic linking only loads that code which is necessary, and as a further optimization, since most library code only writes to the stack (or in the calling program's heap) library code pages can be mapped into many running processes simultaneously. To reduce the on-disk profile of programs, dynamic linking does not require that every binary effectively be self-contained – duplicating shared functionality. Dynamic linking also cuts down on load time, since oftentimes, the required pages are already loaded into memory.

7 Conclusion

The design and implementation of MinixVM presented in this report demonstrates that it is possible to implement VM in a microkernel that cannot do dynamic memory allocation in the kernel. It also clearly demonstrates that the design is overly complicated in relation to the conceptual representation of VM. This should not be surprising, since most of the operating system primitives that developers have come to expect were originally designed for monolithic operating systems.

As for the fate of the microkernel architecture, Tanenbaum famously stated in 1992 that Linux, and by extension the monolithic approach to implementing operating systems, was obsolete [9]. The current proliferation of Linux and other monolithic kernels compounded with the relative obscurity of Minix 3 perhaps provides a more realistic appraisal of the monolithic kernel's obsolescence.

While in terms of high-level design microkernels seem to provide significant advantages over their monolithic counterparts, it is an inescapable fact that programming microkernels is hard. Even noted kernel hacker and open source evangelist Richard Stallman admitted this in his failure to create the GNU Hurd microkernel operating system [7]. All of the complicated and in some cases intractable problems associated with building distributed systems are wrapped up in a microkernel. This is in addition to the fact that developing an operating system is already a sufficiently hard engineering problem.

These difficulties not withstanding, it would seem as if the failure of the microkernel architecture is built into its design. Developing a real-world complex system is an exercise in constant compromise. The problem with the microkernel design is that most of its advantages are based upon the rigid adherence to the boundaries imposed by the design. As soon as these boundaries are violated, many of the advantages evaporate, and all that remains is a difficult to program hybrid operating system.

It is easy to argue that progress has been made in using the microkernel design to increase fault tolerance and provide more trusted computing environments, but as demonstrated by MinixVM, this comes at the cost of providing even the most basic operating system services.

8 References

[1] A. Chou, J. Yang, B. Chelf, S. Hallem, and D. Engler, An Emperical Study of Operating System Errors, In Proc. 18th ACM Symp, on Oper. Syst. Prin., pages 73–88, 2001.

[2] Jorrit N. Herder, Herbert Bos, Ben Gras, Philip Homburg, and Andrew S. Tanenbaum "Construction of a Highly Dependable Operating System", Proc. 6th European Dependable Comp. Conf., Oct 2006.

[3] Jorrit N. Herder, Herbert Bos, Ben Gras, Philip Homburg, and Andrew S. Tanenbaum MINIX 3: A Highly Reliable, Self-Repairing Operating System, ACM SIGOPS Operating Systems Review, vol. 40, nr. 3, pp. 80–89, July 2006.

[4] Intel 64 and IA-32 Architectures Software Developer's Manual, Volume 3A: System Programming Guide, Part 1, November 2008.

[5] Intel 64 and IA-32 Architectures Software Developer's Manual, Volume 3B: System Programming Guide, Part 2, November 2008.

[6] J. Liedtke. On μ-Kernel Construction. In Proc. 15th ACM Symp. on Oper. Syst. Prin., pages 237250, Dec. 1995.

[7] J.T.S. Moore (writer, director, producer), "Revolution OS", Documentary film on the free software movement, available at: <http://www.revolution-os.com/>, 2001.

[8] The Single UNIX Specification, Version 3, ISO/IEC 9945:2003, IEEE Std 1003.1-2001.

[9] Andrew Tanenbaum, "LINUX is obsolete", Posted on comp.os.minix, archived at: <http://groups.google.com/group/comp.os.minix/browse_thread/thread/c25870d7a41696d2/f447530d082cd95d?tvc=2>, 1992.

9 Appendix: MinixVM Patch Summary

The following is a summary of the MinixVM patch (for Minix version 3.1.3a), created using the `diffstat` utility.

```
 etc/usr/rc                    |    5
 include/minix/com.h           |   72 +++++++
 include/minix/const.h         |    6
 include/minix/syslib.h        |    2
 kernel/Makefile               |    2
 kernel/arch/i386/.depend      |    2
 kernel/arch/i386/Makefile     |    1
 kernel/arch/i386/exception.c  |   47 ++++
 kernel/arch/i386/klib386.s    |   12 +
 kernel/arch/i386/memory.c     |   90 +++++++++
 kernel/arch/i386/mpx386.s     |    2
 kernel/arch/i386/proto.h      |    1
 kernel/arch/i386/sconst.h     |    1
 kernel/config.h               |    1
 kernel/pagefault.h            |    9
 kernel/proc.h                 |    1
 kernel/proto.h                |    3
 kernel/system.c               |   12 +
 kernel/system.h               |    5
 kernel/system/.depend         |   37 +++
 kernel/system/Makefile        |    7
 kernel/system/do_exec.c       |    8
 kernel/system/do_pagefault.c  |   75 +++++++
 kernel/system/do_vm.c         |   11 +
 kernel/table.c                |    2
 lib/syslib/sys_exec.c         |    4
 servers/Makefile              |    3
 servers/pager/.depend         |  103 ++++++++++
 servers/pager/Makefile        |   34 +++
 servers/pager/alloc.c         |  256 ++++++++++++++++++++++++++
 servers/pager/alloc.h         |   12 +
 servers/pager/entries.h       |   56 +++++
 servers/pager/exec.c          |   76 +++++++
 servers/pager/exec.h          |   11 +
 servers/pager/page.h          |   12 +
 servers/pager/pager.c         |  126 ++++++++++++
 servers/pager/pager.h         |   25 ++
 servers/pager/pagetable.c     |  120 ++++++++++++
 servers/pager/pproc.c         |    3
```

```
servers/pager/pproc.h        |   26 ++
servers/pager/version.c      |    1
servers/pm/.depend           |   81 +++++++-
servers/pm/Makefile          |    7
servers/pm/alloc.c           |  410 ++++++++++++++++++++++++++++++++++++++++++
servers/pm/exec.c            |   49 ++++-
servers/pm/globalpage.c      |  131 ++++++++++++
servers/pm/main.c            |   59 +++++-
servers/pm/pagefault.c       |  151 ++++++++++++++
servers/pm/proto.h           |   16 -
servers/pm/queue.h           |  215 ++++++++++++++++++++
servers/pm/type.h            |    1
servers/vfs/.depend          |   80 ++++++++
servers/vfs/Makefile         |    3
servers/vfs/exec.c           |   18 +
servers/vfs/fproc.h          |    8
servers/vfs/main.c           |   13 +
servers/vfs/misc.c           |    2
servers/vfs/page.c           |  145 ++++++++++++++
servers/vfs/pagefault.c      |   33 +++
servers/vfs/proto.h          |    8
tools/Makefile               |    1
tools/revision               |    1
62 files changed, 2274 insertions(+), 440 deletions(-)
```

Download the full MinixVM patch (for Minix 3.1.3a).