TOWARDS A TRUE MICROKERNEL OPERATING SYSTEM

A revision of MINIX that brings quality enhancements and strongly reduces the kernel in size by moving device drivers to user-space

Jorrit N. Herder

A thesis

in

Computer Science

Presented to the Vrije Universiteit Amsterdam in Partial Fulfillment of the Requirements for the Degree of Master of Science

February 23, 2005

vrije Universiteit *amsterdam*

TOWARDS A TRUE MICROKERNEL OPERATING SYSTEM

A revision of MINIX that brings quality enhancements and strongly reduces the kernel in size by moving device drivers to user-space

Jorrit N. Herder

APPROVED BY

prof.dr.	**Andrew S. Tanenbaum**: (supervisor)	____________________
dr.	**Bruno Crispo**: (second reader)	____________________

Abstract

An operating system forms the foundation for all of the user's computer activities. Therefore, it should be trustworthy and function flawlessly. Unfortunately, today's operating systems, such as Windows and Linux, fail to deliver to this ideal, because they suffer from fundamental design flaws and bugs. Their monolithic kernel tend be overloaded with functionality that runs at the highest privilege level. This easily introduces bugs and breaches the Principle of Least Authorization (POLA) with all the related risks. A malfunctioning third-party device driver, for example, can easily reek havoc on the system and leave it in a state of total mayhem.

Microkernel operating systems have a different design that makes them less vulnerable to these problems. A microkernel provides only a minimal set of abstractions that runs at the highest privilege level. Extended operating system functionality is typically available by means of user-space servers. By splitting an operating system into small, independent parts, the system becomes less complex and more robust, because the smaller parts are more manageable and help to isolate faults, respectively.

This thesis describes an effort to create a more reliable operating system by exploiting modularity. MINIX was chosen as the base operating system for this project because it already is relatively small and simple, but provides POSIX compliance at the same time. MINIX' kernel can be characterized as a hybrid microkernel because it includes device drivers. MINIX' memory manager (MM) and file system (FS), however, are already implemented as independent user-space servers.

The main contribution of this work is that MINIX was fully revised to become a true microkernel operating system. In kernel-space, several system calls were added to support the user-space device drivers, MINIX' interprocess communication (IPC) facilities were improved, and a new shutdown sequence was realized. In user-space, a new information server (IS) was set up to handle debugging dumps and a library was created to maintain a list watchdog timers. These modifications made it possible to strongly reduce the size of MINIX' kernel by transforming the PRINTER, MEMORY, AT_WINI, FLOPPY and TTY tasks into independent, user-space device drivers.

Preface

With the completion of this thesis a five and a half year period of Computer Science studies comes to end. When I first came to the VU in September 1999 my first semester's mentor happened to be Andy Tanenbaum. From the mentor meetings I basically only remember that he welcomed his group of students with homemade chocolate chip cookies.

After completing my Bachelor's in January 2003 I decided to do my Master's at the Computer Systems section. By chance, Andy again became my mentor. During a compulsory progress meeting he overwhelmed me with the proposal to work on a project he had in mind. Although the project conflicted with my regular courses I gladly accepted the challenge. This paid off later when it was decided to let this project coincide with my master's project. This thesis describes the project's outcome—thus far.

There is a number of people who I want to thank for helping me to accomplish this result. First of all, I thank Andy Tanenbaum for asking me to work on this great project and for his excellent support along the road. The combination of Andy's enthusiasm, sharp insights, and high standards provide a rewarding environment to work in.

Furthermore, I thank Ruediger Weis for his highly motivating words about this project. According to him I've been working on "the Crown Jewels of Europe's software history." It's also good to know that "MINIX has potential to take over the world." Let's see where this goes . . . [1]

While most of the work was done at home, Andy arranged a proper workspace at the VU late 2004. I decided to work at the VU for a few days per week to get familiar with the Computer Systems section. I thank the entire section for their warm welcome. I feel lucky with my roommates and the nice group of people that daily joins for lunch.

Finally, I thank my parents and my girlfriend, Eva Marinus, for their invaluable support in the broadest sense of the word.

[1] Mid 2004 a project proposal was submitted to NWO, which ranked it 1st out of 58 computer science projects. Backed by an NWO grant I will continue my work on this project as a Ph.D. student as of March 1.

Contents

List of Figures

Listings

Note: all listings in this thesis concern new or revised MINIX sources.

Chapter 1

Introduction

Operating systems are the lowest level software that control the computer's bare hardware and provide the base upon which application programs can be written. An operating system thus forms the foundation for all of the user's computer activities. Therefore, it should be trustworthy and function flawlessly. Unfortunately, today's operating systems, such as Windows and Linux, fail to deliver to this ideal. Most computer users are accustomed to frequent operating system crashes and are plagued by digital pests.

The reason for these problems is that today's operating systems suffer from fundamental design flaws and bugs. Their design can be characterized as a large monolithic program which is in full control of the machine. Common design guidelines such as modularity and the principle of least authorization (POLA) are typically violated to win some performance. Third party device drivers, for example, normally are an integral part of the kernel and have all privileges with which they can bring down the entire system.

Another major problem is the inevitable existence of bugs. Software statistics typically provide a number of 1 to 20 bugs per 1000 lines of code (LoC). This number may be even worse for operating systems that are more complex than the average piece of software. Today's most widely used operating systems have millions of lines of code (MLoC) and thus may easily have thousands of bugs. Windows XP, for example, has about 50 MLoC which translates up to 1 million bugs in the worst case.

In this master's project an effort is made to create a more reliable operating system by exploiting modularity. By splitting an operating system into small, independent parts the system becomes less complex and more robust because the smaller parts are more manageable and help to isolate faults, respectively. MINIX was chosen as the base operating system for this project because it already is relatively small and simple, but provides POSIX compliance at the same time. The primary goal is to transform MINIX' device drivers into independent programs so that MINIX becomes a true microkernel operating system.

The remainder of this chapter is structured as follows. Section 1.1 provides an introduction to general operating system principles. Section 1.2 discusses microkernel operating systems in more detail and Section 1.3 introduces MINIX. Section 1.4 provides a precise problem statement and Section 1.5 presents the approach that was taken for this project. Finally, Section 1.6 outlines the structure of the rest of this thesis.

1.1 Operating systems

This section introduces general operating systems principles. The next subsection starts with important operating system concepts. Subsection 1.1.2 classifies different operating system structures. Subsection 1.1.3 compares the key properties of monolithic and microkernel operating systems

1.1.1 Basic concepts

In general, operating systems can be viewed from two perspectives. They can be regarded as both a *resource manager* and an *extended* or *virtual machine* [1]. The resource manager is responsible for providing access to the machine's hardware and securely multiplexing all requests. Resources, for example, may include the hard disk, CD-ROM, printer and video memory, and may be shared among multiple applications.

The extended machine is meant to enhance the hardware's capabilities with higher-level functionality. Traditional operating systems may, for example, provide file systems and windowing functionality. The extended machine defines the interface for the application programs. In contrast to the resource management, this interface may be portable between different hardware architectures.

The kernel is that part of the operating system that runs at the highest privilege level, called *kernel mode* or *supervisor mode*. Kernel mode refers to a CPU flag that indicates whether or not the running process is allowed to execute all possible instructions. Programs that run in user mode do not have this flag set and thus are not allowed to perform certain instructions.

The operating system interface is defined by the system calls that are available to the programmer. System calls use a special machine instruction, called a trap, which causes the processor to change from user mode to kernel mode and to dispatch the kernel's system call handler. This way user processes can request system services to perform restricted actions such as accessing the hardware.

Processes can communicate with help of *interprocess commmunication* (IPC) facilities that often are implemented by means of a system call. IPC, for example, can be implemented by passing a request message followed by a *context switch*. A context switch means to stop the running process and save its volatile state, so that another process can be restored and restarted.

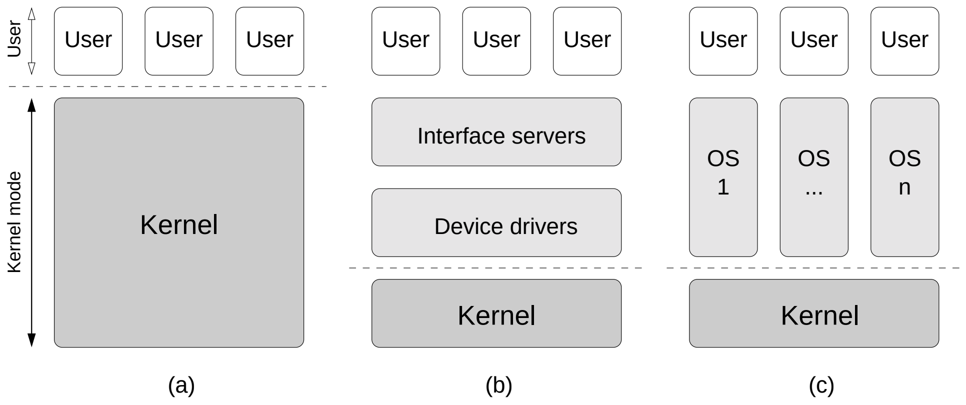

Figure 1.1: A classification of operating systems. Monolithic kernels (a) are usually unstructured. Microkernels (b) often have a layered structures. Exokernels (c) are known as vertically structured operating systems.

The latter is done by issuing a special machine instruction after restoring a process' state.

A key issue for operating systems is to make sure that user applications cannot interfere with the kernel and with each other. This is done by encapsulating processes in address spaces that are physically protected from each other. The memory management unit (MMU) hardware enforces this protection by verifying each attempt to access memory. Illegal accesses result in an exception that is caught by an exception handler in the kernel, which can take appropriate actions.

1.1.2 Operating system structures

Operating systems can be categorized based on their kernel design. The distinction between the resource manager and extended machine views on an operating system turns out to be relevant for categorizing different structures. Three broad categories can be distinguished:

Monolithic kernels Monolithic kernels provide rich and powerful abstractions of the underlying hardware. All operating system services are compiled as single, monolithic program that runs in kernel mode, whereas applications run in user mode and can request system services from the kernel. The kernel thus both is resource managemer and an extended machine. From a high-level perspective, a monolithic kernel is unstructured. This is illustrated in Figure 1.1(a). Examples of monolithic operating systems are Windows, BSD UNIX and Linux.

Microkernels and hybrid kernels Microkernels are characterized by a minimal set of kernel abstractions, but need not necessarily be small in size. Microkernels provide a small set of simple hardware abstractions and use applications called servers to provide more functionality. Only a minimal part of the operating system runs in kernel mode, whereas all applications run in user mode. In contrast to monolithic kernels, a microkernel's main function is resource management. A microkernel operating system has a loosely layered structure with client-server communication between the layers. This is illustrated in Figure 1.1(b). An example of a true microkernel is L4 [2].

Hybrid kernels are very much like microkernels, but are augmented with additional services running in kernel-space for performance reasons. MINIX [1] and Mach [3], for example, are part of this category because their device drivers are part of the kernel. QNX [4] also is a hybrid microkernel because its process manager is part of the kernel.

Virtual machines and exokernels Virtual machines and exokernels do not provide a hardware abstraction layer like other operating systems; instead, they respectively duplicate or partition the hardware resources so that multiple operating systems can run next to each other with the illusion of having a private machine. A virtual machine monitor or exokernel runs in kernel mode and is responsible for the protection of resources and multiplexing hardware requests, whereas each operating system runs in user mode, fully isolated from the other. Virtual machines and exokernels are also known as vertically structured operating systems. Each of the operating systems running next to each other can either have a monolithic kernel or a microkernel structure. This is illustrated in Figure 1.1(c). Examples of virtual machines and exokernels respectively are VMware [5] and the MIT exokernel [6].

As virtual machines and exokernels are merely a means to provide an interface to an operating system, they will not be discussed any further. The distinction between monolithic kernels and microkernels is more interesting for this master's project.

1.1.3 Kernel properties

An overview of operating system properties is given in Figure 1.2. While operating systems with a monolithic kernel often have a good performance, microkernel operating systems tend to do better on all other properties. Most differences directly stem from the difference in modularity.

Modularity is the key property that gives power to microkernels. In contrast to monolithic systems, microkernel operating systems are usually realized as a set of cooperating servers, each dedicated to its task. This gives a clear separation of responsibilities with all related benefits.

Examples of properties that may be found in microkernel operating systems follow. Microkernels are flexible and can easily be extended because it

	Monolithic kernel	**Microkernel**
Modularity	-	✓
Complexity	-	✓
Flexibility	-	✓
Maintainability	-	✓
Security	-	✓
Performance	✓	?
Compatibility	-	✓

Figure 1.2: Comparison of operating systems properties for monolithic kernels and microkernels. The performance of a microkernel-based operating system is not necessarily bad, as is discussed in Subsection 1.2.3.

is possible to replace servers or problem solving strategies. They are easier to maintain because small components are less complex and more manageable. The separation of responsibilities also is beneficial for security and robustness of the operating system because faults are isolated and malfunctioning components may be replaced on the fly. More details on security can be found in Subsection 1.2.2.

A property that has long been used against microkernels is *performance*. Monolithic system often provide a good performance because all services are part of the kernel and thus can directly access each other. A microkernel operating system, in contrast, requires additional communication to let system servers cooperate. For this reason, microkernels are often said to be slow, while this is not necessarily the case. Subsection 1.2.3 discusses this issue in more detail.

An interesting property of microkernels is that they make it possible to preserve a UNIX environment while experimenting with novel applications. They thus provide backwards compatibility while making a transition to a new computing environment. This is further discussed in Subsection 1.2.1.

While the potential benefits are tremendous, examples of successful microkernel operating systems are hard to find. QNX perhaps fulfills most promises, but, unfortunately, is a closed, commercial system. This master's project—and future work that will be based on it—may help MINIX to occupy a niche for open source microkernel operating systems.

1.2 Microkernel operating systems

While the previous section introduces general operating system principles, this section focuses on microkernel operating systems. Subsection 1.2.1 treats three different ways to structure applications that are built on top of microkernels. Two important operating system properties—security and performance— are discussed in Subsection 1.2.2 and 1.2.3, respectively.

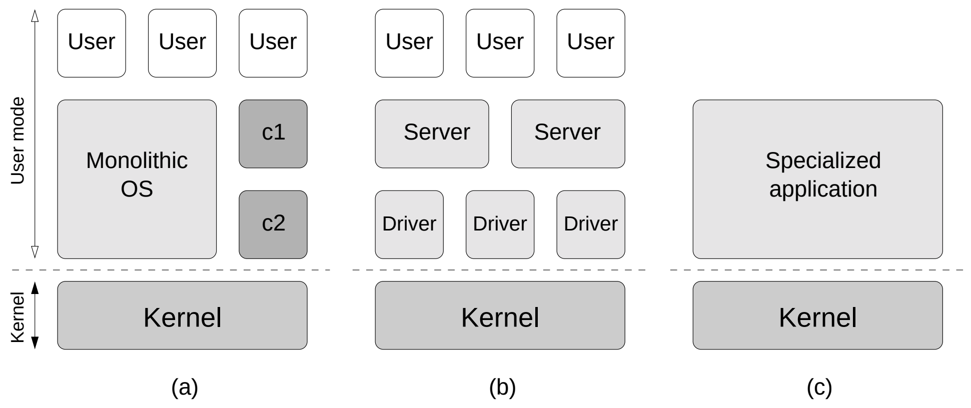

Figure 1.3: A classification of microkernel applications based on the structure of the programs that run in user mode. Different structures are (a) a single-server operating system with specialized components, (b) a multiserver operating system, and (c) a dedicated system.

1.2.1 Microkernel applications

In Subsection 1.1.2, it is outlined that hybrid kernels and microkernels provide a reduced set of abstractions compared to monolithic kernels. Therefore, microkernels rely on user-space servers to provide additional functionality for the application programs that are to run on top of the operating system. This functionality can be realized in several different ways.

Single-server operating system In a single-server OS, the microkernel runs an entire monolithic OS as an ordinary user program. This is illustrated in Figure 1.3(a). This setup does not change any of the properties of the monolithic OS and thus means that there still is a single point of failure. The main advantage of this setup is that is allows to preserve a UNIX environment while experimenting with a microkernel approach. Legacy applications that are targeted towards the monolithic OS often can coexist with novel applications.[1] The combination of legacy applications and real-time or secure components allows for a smooth transition to a new computing environment.

Many examples of this approach exists. Mach was one of the first microkernels around and allowed to run multiple 'OS personalities' next to each other, including BSD UNIX and OSF/1 [8]. Another example is L^{4}Linux [9], which runs on top of the L4 microkernel. The PERSEUS project [10] is an effort to run specialized components that enable secure digital signatures next to L^{4}Linux.

[1]The usual approach to use legacy applications on top of a single-server OS is to recompile the applications with a new system call stub library. In specific systems, such as Mach, trap redirection [7] can be used to realize binary compatibility.

Multiserver operating system In a multiserver OS, the operating system environment is formed by a number of cooperating servers. This is illustrated in Figure 1.3(b). The increased modularity brings many benefits, as is discussed in Subsection 1.1.3, including improved robustness, maintainability, flexibility. Depending on the functionality provided by the multiserver environment, legacy applications may still be usable if they are linked with an emulation library. Novel applications can simply be realized by writing a new system server.

Several examples of multiserver operating systems exist. The GNU Hurd, for example, is a multiserver environment that runs on top of a modified Mach microkernel. SawMill Linux [11] is a multiserver environment on top of the L4 microkernel. Subsection 1.3 discusses yet another example, MINIX.

Dedicated system Another organization is to use a specialized application that directly runs on top of the microkernel. This is illustrated in Figure 1.3(c). Many variations of such a system can be thought of. This is especially useful for mobile and embedded devices with reduced computing power.

An example of a dedicated system is a Java virtual machine (JVM) that runs directly on top of the microkernel. Compared to a traditional approach where an operating system hosts the JVM, this realizes a more secure environment for Java applications, because the host no longer needs to be trusted.

1.2.2 Microkernel security

Microkernels have good security properties, as already mentioned in Subsection 1.1.3. This is especially true for a multiserver OS where all system servers are encapsulated in address space. Servers then are physically protected from each other by the memory management unit (MMU) hardware. If a process illegally tries to access another process' memory, this is detected by the MMU and an exception will be raised. The exception is caught by the kernel, which can take any action deemed necessary.

While a server cannot directly corrupt other servers, dependent servers may be indirectly affected. If server A relies on server B to perform a task, server A may be affected by a malfunctioning in or malicious action by server B. An important concept relating to this is the *trusted computing base* (TCB), that is, the minimal set of components whose correct functionality is a precondition for security [12]. For microkernels, the TCB can be very small. In a dedicated system, for example, a specialized application's TCB would only be the microkernel and the underlying hardware.

An important security issue that cannot easily be solved is that components with harware control have the ability to corrupt the entire system.

For example, device drivers that can use direct memory access (DMA) may corrupt the memory of arbitrary processes be providing an invalid address to the DMA controller. Some PCI chip sets have an I/O MMU that offer protection by mapping a PCI address space onto a known area of physical memory. A more general solution to this problem is not yet available.

1.2.3 Performance issues

Microkernel operating systems often are claimed to suffer from inherent performance problems because multiple processes must cooperate to perform a task. The reason for the alleged performance loss is that extra interprocess communication (IPC) and thus extra context switches are needed between user processes, system servers and the kernel. Furthermore, copying of data between cooperating servers causes additional overhead.

The first microkernels around indeed had a substantial performance penalty. BSD UNIX on top of Mach, for example, is well over 50% slower than the normal version of BSD UNIX. Modern microkernels, however, have proven that high performance actually can be realized. L^4Linux on top of L4, for example, only has a performance loss of 2 to 4%.

Performance turns out to be an implementation problem [13]. The approach that was taken in L4 is to define a portable microkernel interface, but to make optimal use of hardware capabilities for the implementation.[2] Each platform thus requires a different kernel implementation, but applications that adhere to L4's kernel API can simply be recompiled and reused.

Another important lesson is that microkernels should be small. A performance breakdown of L4 revealed that cache misses are more important than IPC itself [15]. A context switch from a user-space process to the kernel requires only changing a few bits at the CPU to switch to kernel mode and to restore the kernel's state. The address space switch, however, may cause costly cache misses. This penalty becomes more severe if virtual memory is used because the entries of the translation look-aside buffer (TLB) are also invalidated by an address space switch.

Finally, software should be explicitly (re)designed for microkernels. Microkernels are different from monolithic and thus may require a different design to achieve a good performance. Research on SawMill Linux, for example, was aimed at designing a multiserver protocol that reduces IPC and prevents unnecessary data copies. The performance of microkernel-based operating systems thus cannot be measured by IPC costs alone [16].

[2]L4, for example, makes optimal use of CPU registers for IPC. Messages are defined as a set of virtual registers that are mapped to real registers whenever possible. IPC stub-generation supports the programmer in selecting the optimal implementation [14].

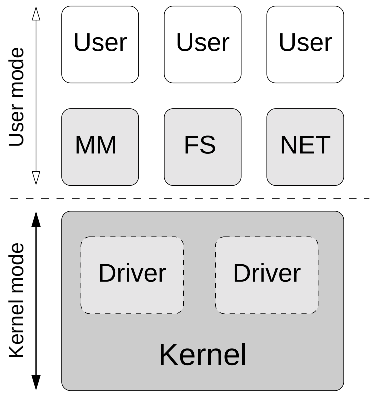

Figure 1.4: MINIX 2.0.4 can characterized as a multiserver operating system with a hybrid microkernel. All device drivers are part of the kernel, while important system functionality is realized by user-space servers, such as the file system (FS), memory manager (MM), and network server (INET).

1.3 Introduction to MINIX

This master's project focuses on microkernel operating systems in general and MINIX [1] in particular. MINIX is a free microkernel-based operating system that comes with complete the source code, mostly written in C. It was written by Andrew S. Tanenbaum in 1987 as an educational operating system that is easy to learn and maintain. MINIX is accompanied by a book that explains its design in great detail.

The design of MINIX can be characterized as a multiserver operating system with a hybrid microkernel. Its structure is illustrated in Figure 1.4. Major components such as the file system (FS) and memory manager (MM) are set up as separate servers that run in user mode. MINIX has a hybrid microkernel because device drivers are compiled as part of the kernel.

Because the device drivers are part of the kernel, they run at the highest privilege level. In effect, this means that MINIX' device drivers are fully trusted, while they should not be. Therefore, one of the goals of this master's project is to transform device drivers into independent, user-space servers so that MINIX becomes a multiserver operating system with a true microkernel. A more precise problem statement is given in Subsection 1.4.

Since the initial version of MINIX the source tree has gradually evolved, but its kernel has been rather stable. Development on the second version of MINIX, which is POSIX-conformant, has been done since 1996.

This project is based upon MINIX 2.0.4, which was released at November 2003 as a 'mid development expert-only snapshot' for the sake of this project. The kernel of MINIX 2.0.4 roughly comprises 20,000 lines of code (LoC), including comments. MINIX 2.0.4 and other versions of MINIX are publicly available from *http://www.cs.vu.nl/pub/minix/*.

Appendix A.1 provides an overview of all files that are part of the kernel in MINIX 2.0.4. In total, the kernel has 77 files and is 878 KB in size with all drivers included. The microkernel portion of the code only is 220 KB. The listing shows that only a small fraction (25%) of the source code belongs to MINIX' microkernel, whereas the majority (75%) of the code is made up by device drivers. Removing the device drivers from the kernel thus results in a huge reduction of code that runs at the highest privilege level.

1.4 Problem statement

The goal of this master's project is to revise MINIX 2.0.4 to get a more reliable operating system. This is realized by enhancing MINIX' kernel so that it becomes smaller and better. The problem statement thus is twofold:

Kernel reductions: MINIX' kernel can be reduced by transforming device drivers that run as kernel tasks into independent, user-space programs. Although MINIX' device drivers are designed as separately scheduled processes, called tasks, they are compiled into the kernel. This means that they run at the highest privilege level—with all the related risks. By removing the drivers from the kernel and having them to runs as ordinary user-space processes, MINIX will become more secure, because the device drivers and the kernel will be physically protected from each other.

Quality enhancements: This step concerns improving the quality of the operating system, in general. MINIX' kernel contains several functions that are not strictly required within the kernel. Nevertheless, this code has access to important kernel data structures and thus may endanger the system. All such code also should be removed from the kernel, for example, by setting up additional user-space servers. Furthermore, some parts of the kernel's code can be improved by redesigning the code or adding new security features.

The anticipated outcome of the project can be characterized as a multiserver operating system with a true microkernel. MINIX' new structure is shown in Figure 1.5. Because of the gross restructuring the version number will be raised from MINIX 2.0.4 to MINIX 3.0.0.

1.5 Approach followed

Getting familiar with MINIX. The project started with thoroughly studying the book [1] that accompanies MINIX 2.0.0. The chapters on the implementation of processes and device I/O were given the most attention. After this, the source tree of MINIX 2.0.4 was studied and some initial experiments with user-space I/O were undertaken.

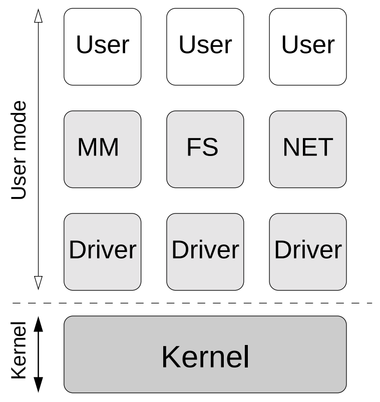

Figure 1.5: One of the goals of this project is to remove device drivers from the kernel, so that MINIX 3.0.0 can characterized as a multiserver operating system with a true microkernel.

Timing measurements. The restructuring of MINIX introduces some communication overhead because user-space device drivers and the kernel have to cooperate to perform the same tasks as before. Timing measurements were conducted on two different machines to assess the performance penalty. The test setup and the results can be found in Section 2.1.

Comparison of design options. The study of MINIX' source code turned out that user-space system services can be started in two ways. While most servers are statically configured into the system image, a dynamic approach also exists. A comparison of these possibilities and the approach that was chosen for this project is discussed in Section 2.2.

Feasibility study. This step consisted of moving the Centronics printer driver out of the kernel. The successful transformation of the kernel-space PRINTER task into a user-space servers built confidence for the rest of the project. The outcome of the feasibility study and the problem that were encountered are provided in Section 5.1.

Analyzing dependencies. A dependencies matrix was created by compiling all kernel tasks in isolation and gathering all missing symbols. Different ways to remove dependencies were analyzed and a functional classification was created to find a general approach for related dependencies instead of ad hoc solutions for individual device drivers.

Transformation of device drivers. The general idea that allows to have user-space device drivers, is to support them with system calls that execute privileged operations within the kernel. Many new system calls were added to the kernel and some parts of MINIX were completely redesigned to cleanly remove all dependencies. Once all dependencies were resolved, several device drivers were removed from the kernel. The results are described in detail in Chapter 3 to Chapter 5.

Improving the code quality. While removing the dependencies, much of MINIX' code was analyzed. This revealed certain shortcomings and room for improvement. Therefore, general improvements were made to the code whenever possible.

1.6 Outline of this thesis

This thesis is organized as follows. The next chapter provides a problem analysis and briefly discusses different design options. The problem analysis mainly focuses on the dependencies that play a role when device drivers are moved out of the kernel.

Chapter 3 gives an overview of the improvements that were made to MINIX' kernel. The improvements include various new system calls to support user-space device drivers, changes to MINIX' interprocess communication (IPC) facilities, and a full revision of MINIX' shutdown sequence.

Chapter 4 treats various new applications and illustrates the usefulness the various kernel improvements. New applications include a new server for obtaining system information, a simpler and generic way to manage watchdog timers, and new approaches to deal with unresponsive hardware.

Chapter 5 discusses how the kernel was strongly reduced in size. This mainly concerns the transformation of entangled, kernel-space device drivers into independent, user-space programs. The discussion covers the PRINTER, MEMORY, AT_WINI, FLOPPY, and TTY device drivers.

Chapter 6 surveys related work in microkernel operating systems. Three typical microkernels—Mach, QNX and L4—and some of their applications are discussed. Mach is interesting from a historical perspective. QNX Neutrino is a successful commercial microkernel for embedded systems. L4 can be characterized by is small size and high performance.

Finally, Chapter 7 concludes this thesis. It provides an overview of the major contributions by summarizing the results, it briefly looks back to what was accomplished and draws some conclusions, and then it describes possible areas of future work.

In the end, several appendices cover the details that did not fit in the main text. Appendix A provides an overview of the files that belong to MINIX' kernel and outlines the new source tree; Appendix B tells how to make certain changes; and Appendix C lists all system calls of MINIX.

Chapter 2

Problem analysis

This chapter presents the problem analysis that was conducted before the actual programming started. Section 2.1 presents timing measurements that were done to get an impression of the overhead incurred by user-space device drivers. This is followed by a discussion of the different design options for loading system services in Section 2.2. Finally, Section 2.3 provides a detailed analysis and classification of the dependencies that have to be dealt with when kernel tasks are transformed into user-space device drivers.

As mentioned in Subsection 1.5, the transformation of the PRINTER task into a user-space device driver served as a feasibility study for this project. While this can be considered as a part of the problem analysis, the results are discussed in Chapter 5, together with the other user-space drivers.

2.1 Timing measurements

The transformation of kernel tasks into independent user-space servers requires additional communication because the servers have to cooperate with the kernel (and possibly with other servers) to perform their work. Device drivers, for example, no longer have privileges to directly perform device I/O. Therefore, they must request the kernel to read or write a certain device register on their behalf.

The restructuring of MINIX thus requires extra context switches and will introduce some performance overhead. The time needed for a typical request-response sequence was measured to get an impression of the performance penalty. The test setup and the results are discussed below.

2.1.1 Test setup

The measurements are done within the kernel because microsecond precision readings of the clock counter are required. During the test an illegal request message is sent from the CLOCK task to the FS server, which immediately

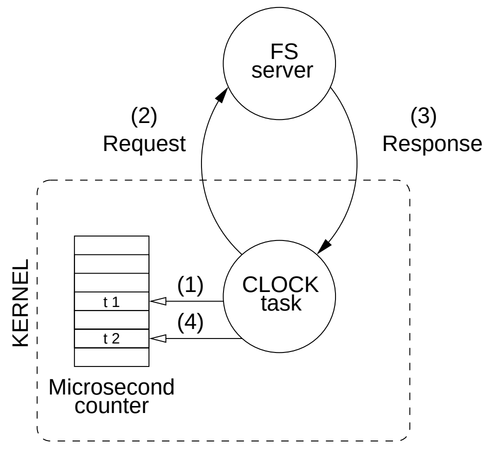

Figure 2.1: Test setup for measuring the incurred overhead of a typical request-response sequence. The timing measurements includes two message copies, two context switches, and the minimal logic to evaluate the request.

responds with an error message. The clock counter is read just before sending the request message and just after receiving the response message. The test setup is illustrated in Figure 2.1.

The measurement gives an impression of the incurred overhead when processes must cooperate to perform a certain task. The test setup includes two message copies, two context switches, and the minimal logic[1] to evaluate the request. Since an illegal request is made an error is immediately returned, and the actual work at the server is excluded. This means that the pure overhead of a typical request-response sequence is measured.

The implementation of this test required some small changes to the CLOCK task. A new function was defined in *src/kernel/clock.c* to do microsecond precision reading of the 8253A timer. Furthermore, a temporary system call was implemented at the CLOCK task to initiate the test sequence. No changes were required to the FS server.

2.1.2 Test results and discussion

The tests were conducted on two Intel machines with different CPU speeds. Thirty tests were run on each machine. The results are shown in Figure 2.2. They show that additional request-response sequences only cost a few microseconds and that the overhead relates to the CPU speed. Since the test machines are quite old, the overhead will much lower on modern computers.

[1]Most servers in MINIX are set up in a similar way. In general, there is a main loop that blocks until a request message is received, evaluates the request and dispatches to the handler function, and returns the result to the caller. For illegal requests no handler function is called, but an error is directly returned.

Machine (CPU)	**Minimum**	**Maximum**	**Average**
Pentium I, 166 MHz	28 μs	30 μs	29.4 μs
Pentium III, 450 MHz	13 μs	16 μs	14.2 μs

Figure 2.2: Test results showing the request-response overhead in a configuration as shown in Figure 2.1. Thirty tests were run on each machine. All results are in microseconds.

Unfortunately, the timing measurements alone do not provide enough information to assess the precise costs. The request-reponse frequency is needed to calculate the actual overhead, but is hard to determine in advance. Such measurements can only be accurately made when one or more device drivers have been moved out of the kernel. Moreover, the results are likely to depend on the type of driver.

To give a rough estimate of the performance penalty, suppose that a typical request-response sequence takes 5 μs on a modern CPU and that 1000 extra sequences per second are required by a user-space device driver. This implies a performance penalty of 5 ms per second, which means that the incurred overhead is only 0.5% of the CPU. The performance overhead thus seems very reasonable compared to the benefits that are gained by having user-space drivers.

While the time measurements presented here are fixed values, the number of request-response sequences may be subject to all kinds of optimizations. Furthermore, copying data from one process to another may be a bigger problem for certain classes of device drivers. Therefore, future work will focus on a more detailed analysis of the performance of user-space device drivers. This is outside the scope of this master's project.

2.2 Design options for system services

System services in MINIX can be loaded in two ways. The standard approach is to statically included them in the system image, but a dynamic approach also exists. These options are compared below and the approach taken for this project is briefly discussed.

2.2.1 Inclusion in the system image

The standard approach to start system services in MINIX 2.0.4 is to include them in the boot image that is produced by the program '`installboot`'. All services in the boot image are automatically started during system initialization. Currently, the boot image includes the kernel, the MM and FS servers, and the INIT process. Because of the way the programs are loaded,[2] there currently is space to include up to 12 other programs in the image.

Including extra programs in the system image requires updating the kernel data structures that are used to initialize the process table. The kernel, for example, must know about the number of programs included in the system images and requires a name for each program. Appendix B describes the precise steps that must be taken.

Since new system services affect the kernel's source code, they cannot be easily distributed and applied to working systems. To make changes the user must obtain the updated source code, apply all patches, and recompile the kernel. This requires some expertise and can only be done by the system administrator. Such updates may not be needed by embedded or stand-alone systems, but in a more dynamic environment this may become a problem.

2.2.2 Load system services on demand

Another possibility is to load system services after the operating system had been loaded. Since system services are treated as special processes in MINIX 2.0.4, this requires some extra steps compared to loading an ordinary user program. The system call SYS_SVRCTL allows a process with superuser privileges to transform itself into a system process during initialization. The steps that are taken to load the INET server are described below.

Dynamic control over system services is beneficial for several reasons. First, it simplifies development and debugging of system services because it does not require kernel modifications nor recompilation or rebooting of the entire system to install a new service. Another interesting benefit is that it facilitates the distribution of new or update system services. System services can easily be started by privileged users or by any user when the setuid bit is enabled.

Another advantage is that there are no practical limits to the number of servers that can be started. Simply increasing the NR_PROCS variable in <*minix/config.h*> would reserve extra resources, that is, space in the process table, to run additional servers. In contrast, the approach that is described above would require to modify the low-level startup code.

Unfortunately, the SYS_SVRCTL system call of MINIX 2.0.4 has several shortcomings that makes it hard to take full advantage of dynamically loading system services. The call, for example, it does not provide a clean way to abort system services and release resource again.

[2]The boot monitor extracts the size of the code and data segment of each program from the system image and stores these values in the _sizes array that used by MINIX to set the memory map of each process. The _sizes array is defined as 64 bytes of space at the beginning of the kernel's data segment. Because each program requires 4 bytes, a maximum of 16 programs can be included in the system image.

INET is playing tricks

In contrast to the FS and MM servers, the network server, INET, is separately distributed and not included in the system image. Thus the kernel does not know that INET is a special process. INET is started like any other user process, but it uses the SYS_SVRCTL system call during its initialization to become a system service. There are three steps in this process:

1. Register with the memory manager. The MM simulates an exit by releasing a waiting parent and disinheriting any child processes. When this is done, the process' ID and group are changed to become a server as far as the MM concerns.

2. Register with the SYSTEM task in the kernel. The SYSTEM task updates the process' type and priority. When this is done, INET no longer is a user process, but has become a real server.

3. The last step is to register itself with the FS in order to manage a device. The FS keeps a table mapping between devices and processes that handle them. The INET server requests to manage the */dev/ip* device. Once the new mapping is in place INET is addressed like any other device driver.

2.2.3 Approach taken in this project

During this project all system services, such as user-space device drivers, were included in the boot image. The reasons for choosing this static configuration are simplicity and time constraints.

The preferred method, however, is to dynamically start and stop system services. Therefore, the design of a proper interface for dynamically controlling system services is part of future work.

2.3 Analysis of dependencies

To get an impression of what needed to be done to move device drivers out of the kernel, their dependencies were analyzed. A dependency means that a symbol, that is, a variable or function, can no longer be directly referenced by a device driver that is compiled as a separate program. In most cases this means that the device driver depends on kernel functionality to do its work, but other kinds of dependencies exist as well. This is discussed in Subsection 2.3.2.

The dependencies were determined by copying all files[3] of a given task to a separate directory and trying to compile the task isolated from the kernel.

[3]The copying of files belonging to a given tasks excluded header files local to the kernel. Depending on the results, such header files or individual definitions may be relocated at a later time. Variables and functions, in contrast, usually cannot be relocated.

Symbol		AHA1540	AT_WINI	BIOS_WINI	ESDI_WINI	XT_WINI	FLOPPY	FATFILE	DOSFILE	MEM	RTL8139	DP8390	PRINTER	SB16_DSP	SB16_MIX	TTY	SYSTEM	CLOCK	Prototype	Implementation
_bad_assertion	F							x			x	x					x		assert.h	misc.h
_cause_sig	F															x		x	proto.h	system.c
_cons_stop	F															x			proto.h	clock.c
_clock_stop	F															x			proto.h	clock.c
_current	V															x			glo.h	-
_data_base	V	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x		glo.h	-
_disable_irq	F												x						proto.h	klib.s
_dosfile_stop	F															x			proto.h	dosfile.c
_dp8390_stop	F															x			proto.h	clock.c
_ega	V															x			glo.h	-
_enable_iop	F									x									proto.h	protect.c
_enable_irq	F	x	x			x	x				x	x	x	x		x	x	x	proto.h	klib.s
_env_panic	F										x								proto.h	misc.c
_env_parse	F	x									x	x							proto.h	misc.c
_env_prefix	F										x	x							proto.h	misc.c
_floppy_stop	F															x			proto.h	floppy.c
_get_uptime	F		x				x				x	x				x	x		proto.h	clock.c
_int86	F			x					x										proto.h	klib.s
_interrupt	F	x	x			x	x				x	x	x	x		x	x	x	proto.h	proc.c
_intr_init	F															x			proto.h	i8259.c
_level0	F			x					x	x						x			proto.h	klib.s
_mem	V									x	x						x		glo.h	-
_mem_rdw	F											x							proto.h	klib.s
_mem_vid_copy	F															x			proto.h	proto.h
_micro_delay	F	x	x			x						x	x			x			proto.h	clock.c
_micro_elapsed	F	x	x				x				x								proto.h	clock.c
_micro_start	F	x	x				x				x								proto.h	clock.c
_mixer_set	F													x					sb16.h	mixer.c
_monitor	F															x			proto.h	klib.s
_mon_params	V															x	x		glo.h	-
_mon_return	V															x			glo.h	-
_numap	F	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x			proto.h	system.c
_panic	F	x	x	x	x	x	x	x	x	x	x	x				x	x	x	proto.h	main.c
_pc_at	V			x												x			glo.h	
_phys2seg	F											x				x			proto.h	protect.c
_phys_copy	F	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x		proto.h	klib.s
_phys_insb	F											x							proto.h	klib.s
_phys_insw	F		x									x							proto.h	klib.s
_phys_outsb	F											x							proto.h	klib.s
_phys_outsw	F		x									x							proto.h	klib.s
_pproc_addr	V	x	x	x		x	x		x	x			x			x	x	x	proc.h	-
_proc	V									x						x	x		proc.h	-
_proc_ptr	V	x		x	x	x	x	x	x	x	x	x						x	glo.h	-
_protected_mode	V									x						x			glo.h	-
_pr_restart	F																	x	proto.h	printer.c
_ps_mca	V																	x	glo.h	-
_put_irq_handler	F	x	x			x	x				x	x	x	x		x	x	x	proto.h	i8259.c
_putk	F																x		proto.h	console.c
_reg86	V			x					x										ibm/int86.h	
_reset	F															x			proto.h	klib.s
_rtl8139_dump	F															x			proto.h	rtl8139.c
_rtl8139_stop	F															x			proto.h	rtl8139.c
_rtl_probe	F											x							proto.h	rtl8029.c
_tasktab	V	x	x	x	x	x	x	x	x	x									glo.h	-
_tmr_exptimers	F	x	x	x	x	x	x	x	x	x						x			proto.h	clock.c
_tmr_settimer	F		x				x				x					x			proto.h	clock.c
_tty_table	V															x			tty.h	-
_tty_timelist	V															x			tty.h	-
_tty_timeout	V															x		x	glo.h	-
_tty_wakeup	F																	x	proto.h	tty.c
_vid_vid_copy	F															x			proto.h	klib.s
_vir_copy	F	x								x								x	proto.h	system.c
_wreboot	F																x		proto.h	keyboard.c

Figure 2.3: Dependencies matrix based on missing symbols after isolated compilation of kernel tasks. Symbols are either a function (F) or a variable (V). The marked drivers were moved to user space; the last two tasks will stay in the kernel.

The task's dependencies then can be found by inspecting the compiler and linker's warnings relating to missing symbols. These steps were taken for all kernel tasks to obtain a list of dependencies for each.

The results of this procedure are listed in the dependencies matrix that is shown in Figure 2.3. The dependencies matrix was used to make a classification of dependencies. By making a classification of different kinds of kernel dependencies, it becomes easier to find general solutions and to realize the migration at a later time. This is discussed in Subsection 2.3.2.

Undetected dependencies

There is one important class of dependencies that was not found this way, and which is not listed on the dependencies matrix. Dependencies relating to device I/O were not found because the device I/O functions are part of the standard system libraries and thus were automatically linked with the device drivers. Device I/O, however, requires special privileges and thus cannot be used by user-space drivers. Any attempt to read or write a register would directly result in a CPU exception.

2.3.1 How to remove dependencies

Different kinds of dependencies require a slightly different approach in removing them. The classification below describes what needs to be done to remove a dependency. Five cases can be distinguished:

Driver-to-kernel. All device drivers depend on kernel functionality. Most drivers, for example, use a kernel function to copy chunks of memory between arbitrary, physical addresses. User-space processes, however, are not permitted to do so. Such dependencies typically can be replaced with a system call that requests the kernel to perform some task on behalf of a user-space process. The new system calls can be implemented in the system library where MINIX' existing system calls also reside.

Driver-to-driver. Drivers can also depend on each other. All system services, for example, rely on the TTY driver to output diagnostic messages. These dependencies can be replaced by defining new message types that allow drivers to communicate with each other. Similar to MINIX' system calls, these interprocess calls can be implemented in a separate library.

Kernel-to-driver. The kernel may also depend on the device drivers. All device drivers, for example, are stopped with a function call from the kernel when MINIX shuts down. Furthermore, interrupt handlers must be part of the kernel, but typically use device driver data structures. The approach for these dependencies similar to what is discussed above, that is, the kernel may send a message to notify the driver of the event for further handling. The exact ordering of messages may pose some problems, though, as is discussed in Subsection 2.3.3.

Bad design. There are several dependencies that are caused by bad design. These are unnecessary dependencies that exist because symbols were globally declared while they were only locally needed. This mainly concerns variables that belong to the TTY task. The CLOCK task, for example, directly sets an event flag at the TTY driver when a TTY alarm goes off. The solution simply is to make such declarations local.

False positives. Finally, some symbols that show up in the dependency matrix are not real dependencies. These symbols are on the list because the SYSTEM and CLOCK task were also separately compiled to find kernel-to-driver dependencies. Of course, these tasks will remain in the kernel to provide low-level services to the drivers, so these dependencies can be ignored. False positives were excluded from the dependencies matrix shown in Figure 2.3.

2.3.2 A functional classification

The dependencies matrix that is shown in Figure 2.3 was analyzed to find functionally related dependencies. Several functional classes could be formed by grouping symbols. The functional classification was useful to find general approaches for entire classes of dependencies, instead of ad hoc solutions for individual dependencies.

A rough functional classification of dependencies, together with the approach to remove the dependencies, is given below. Unfortunately, some dependencies could not easily be grouped and still required an individual approach. Furthermore, it turned out that the different functional classes could not always be dealt with independent from each other.

Handling assertions and panics. These will be handled locally. If a server panics only the server should be aborted. In many cases the rest of the system is not directly affected and can continue to work.

Copying of data. Most copying in MINIX 2.0.4 is done using physical addressing. Copying should be done by the SYSTEM task on behalf of the requesting process. A new, generic copy function that uses virtual addressing will be defined for this.

Process table information. While the process table at first sight seems to cause many dependencies, this in fact is not a real problem. Most references are made to calculate physical addresses for copying data and will no longer be needed when virtual copying is in place. Other fields may be obtained in a way that is discussed next.

Getting system information. Since drivers can no longer directly access important system information, such as kernel environment variables, they should be able to request a copy of this information. A new system call will be created for this.

Use of clock functionality. Instead of directly calling on CLOCK task functionality, such as alarm timers and delays, this should be done by sending a request message. A number of clock library functions will be created, similar to the system library.

Debug dumps. A separate information server will be set up to handle debug dumps that are done from within the kernel in MINIX 2.0.4. The new system call to request system information will be used to retrieve the kernel's data structures.

Interrupt handling. Interrupt handling requires many privileges and thus should be left to generic interrupt handler at the kernel. A new system call to enable or disable interrupts will be added. Furthermore, drivers should be able to set an interrupt policy to be executed by the generic interrupt handler.

Shutting down MINIX. Shut down is a global event that somehow was handled from within the TTY task. Therefore, the code should be moved to a central location, for example, to the file that also contains MINIX' main program. Instead of directly calling other drivers to stop, this will be done with a notification message.

Variables declarations. Many TTY variables are globally allocated declared in MINIX 2.0.4 while this actually is not necessary. This design problem will be corrected by moving the variables to a different header file and changing the way storage is allocated for them.

Device I/O. As mentioned above, device I/O forms a class of dependencies that is not listed on the dependencies matrix. Because device I/O requires special privileges, several system calls will be added to have the kernel perform the I/O on behalf of a user-space device driver.

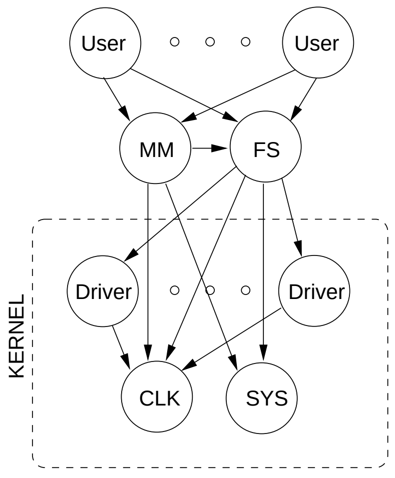

Figure 2.4: The partial message ordering of MINIX 2.0.4. All request messages are top-down to prevent deadlocks. Bottom-up responses are allowed, but are not shown in this figure.

2.3.3 Message ordering problems

Subsection 2.3.1 discusses different kinds of dependencies. If all dependencies were in a top-down direction, that is, driver-to-kernel, a generally applicable approach to remove them would be to add extra system calls.

Unfortunately, bottom-up, that is, kernel-to-driver dependencies also exist. A serious problem is that cyclic dependencies exist. A device driver may not only depend on the kernel, but the kernel may also depend on that same device driver, which may cause a deadlock if both are sending to each other.

The original setup of MINIX has a partial message ordering for the exchange of regular messages. This ordering is top-down. User processes may send to the MM and FS server, the MM may send to the FS server, the servers may send to the kernel tasks, and the kernel device drivers may send to the CLOCK task. This is illustrated in Figure 2.4. To prevent deadlocks bottom-up messages may not be sent in a regular way, unless the message concerns a response to a top-down request. Therefore, asynchronous events such as interrupts and timers are handled in a special manner.

Analysis of the source code of MINIX 2.0.4 turned out that the approach that for dealing with asynchronous events has certain shortcomings and is not suitable for removing all dependencies that require bottom-up communication. Therefore, a more general solution to notify user-space processes of system events was thought of. In short, a new notification construct carefully checks if a process is ready to receive a bottom-up message before sending it. If the receiving process is not waiting for a message, the notification is postponed. The treatment of various asynchronous events is discussed in detail in Section 3.3.

Chapter 3

Kernel improvements

This chapter introduces various improvements to the MINIX kernel. Section 3.1 presents the most important system calls that were added to support user-space device drivers. Section 3.2 discusses improvements to MINIX' interprocess communication (IPC) facilities, including a rewrite of MINIX' system call handler. Section 3.3 treats a mechanism that is built around MINIX' IPC facilities to notify user processes about kernel events without risking a deadlock. Finally, Section 3.4 treats MINIX' new shutdown sequence that was created with the new notification construct.

Note that the following chapters also introduce several kernel improvements. For example, Chapter 4 discusses changes relating to MINIX' timer management and Chapter 5 includes kernel modifications that were initiated by the transformation of specific kernel tasks into user-space drivers.

3.1 Supporting user-space device drivers

When kernel-space device drivers are transformed into independent, user-space programs they lose many of the privileges that they previously had. Although it turned out that various functions could be managed by the device drivers themselves, not all functionality could be relocated to user space. User-space device drivers, for example, can no longer perform device I/O. Another problem is that kernel data structures, such as the process table and environment variables, can no longer be directly accessed.

Therefore, numerous new system calls were added to MINIX to support user-space device drivers in doing their work as before. The drivers simlply request the kernel to do the things that they can no longer do themselves.

This section discusses the most important calls that were added. Appendix C provides a detailed overview of all new and existing system calls. It also provides a short introduction to system call implementation, so this is not discussed below.

Listing 3.1: Handler function for the SYS_DEVIO system call. This system call simply reads or writes a single I/O port on behalf of a user-space device driver.

```
PUBLIC int do_devio(m_ptr)
register message *m_ptr;                                /* request message */
{
  /* Perform actual device I/O for byte, word, and long values. */
  if (m_ptr->DIO_REQUEST == DIO_INPUT) {                /* read I/O port */
      switch (m_ptr->DIO_TYPE) {                        /* port granularity */
      case DIO_BYTE:  m_ptr->DIO_VALUE = inb(m_ptr->DIO_PORT);  break;
      case DIO_WORD:  m_ptr->DIO_VALUE = inw(m_ptr->DIO_PORT);  break;
      case DIO_LONG:  m_ptr->DIO_VALUE = inl(m_ptr->DIO_PORT);  break;
      default: return(EINVAL);
      }
  } else if (m_ptr->DIO_REQUEST == DIO_OUTPUT) {        /* write I/O port */
      switch (m_ptr->DIO_TYPE) {                        /* port granularity */
      case DIO_BYTE:  outb(m_ptr->DIO_PORT, m_ptr->DIO_VALUE);  break;
      case DIO_WORD:  outw(m_ptr->DIO_PORT, m_ptr->DIO_VALUE);  break;
      case DIO_LONG:  outl(m_ptr->DIO_PORT, m_ptr->DIO_VALUE);  break;
      default: return(EINVAL);
      }
  } else { return(EINVAL); }                            /* illegal request */
  return(OK);
}
```

3.1.1 System calls for device I/O

User-space programs cannot directly read from or write to the registers of a device's controller. On some computers I/O is done using special instructions that can be executed only in kernel mode. On the Pentium, I/O is done using I/O registers that, in theory, can be made accessible by user-space programs. However, this mechanism is not used for security reasons.

Three new system calls were created to have the kernel perform the device I/O on behalf of a user-space driver. The simplest call, SYS_DEVIO, allows to read or write a single I/O port at a time. Listing 3.1 shows the kernel's handler function for this system call. A variant of this call, SYS_VDEVIO, allows to read or write a series of I/O ports by passing a vector with (port, value)-pairs. Finally, SYS_SDEVIO, allows to read or write a buffer from or to a given port.

When multiple, consecutive device operations are required a single call to SYS_VDEVIO or SYS_SDEVIO should be used to minimize the system call overhead. Although repeatedly calling SYS_DEVIO is also possible, this requires more context switches and thus should be avoided.

The interface for the device I/O system calls was designed to resemble the device I/O functions that are contained in the system libraries as much as possible. For example, the statement outb(port,value); can be replaced by s=sys_outb(port,value); where s is the return value of the system call.

3.1.2 Generic virtual copying

Many kernel dependencies are caused by device drivers that copy data around. In MINIX 2.0.4 this is mostly done using physical addressing. While this may be tolerated for kernel tasks, user-space device drivers should not be trusted as much and thus should be not be allowed to copy from or to arbitrary memory addresses. Instead, only virtual addressing should be allowed because this provides more control.

MINIX 2.0.4 already has a virtual copy function, but this can only be used to copy between processes' text, data and stack segments. Analysis of MINIX 2.0.4, however, learned that phys_copy() was used to copy from or to user processes, to do BIOS I/O, and to perform RAM disk operations. Therefore, a new generic virtual copy function, virtual_copy(), was created to support all three types of virtual addresses. This function is shown in Listing 3.2. The associated system call, SYS_VIRCOPY, was fully revised.

Several macros were defined in the system library header, <*minix/syslib.h*>, to provide a convenient system call interface. Instead of explicitly providing the segment of the virtual addresses, the programmer can use a macro. To read from the BIOS or to copy data between two processes, for example, sys_biosin() and sys_datacopy() were defined, respectively.

3.1.3 Interrupt handling

When a hardware interrupt occurs the processor automatically traps to the kernel and calls the interrupt service routine. This call is done at the highest CPU privilege level so that the service routine can take all necessary actions to handle the interrupt. In MINIX 2.0.4, device driver tasks are part of the kernel so that their interrupt handlers can directly be called by the interrupt service routine. This process is explained in more detail in Subsection 3.3.1.

Because interrupts cannot be handled directly by a user-space driver, a generic interrupt handler and a new system call, SYS_IRQCTL, were added to the kernel. The call's precise parameters and options are given in Appendix C. User-space device drivers typically use the SYS_IRQCTL call to install an appropriate interrupt handling policy in their initialization phase and then enable interrupts. Whenever an interrupt occurs it is serviced by the generic handler on behalf of the device driver.

To determine what interrupt policies should be supported, the actions taken by the interrupt handlers of the device driver tasks in MINIX 2.0.4 were analysed. Most drivers, including the PRINTER and AT_WINI drivers, only read a status register. Other drivers, including the TTY driver, read a port value and strobe it back to acknowledge the interrupt. Finally, some drivers, including the FLOPPY driver, do not perform any device I/O at all, but leave all interrupt handling to the device driver. Although some drivers do not directly fit in the above scheme, they can be rewritten to do so.

Listing 3.2: The generic virtual copying function that handles the SYS_VIRCOPY system call. Virtual addresses can be in LOCAL_SEG, REMOTE_SEG, or BIOS_SEG. The function copies bytes from src_addr to dst_addr using virtual addressing.

```
PUBLIC int virtual_copy(src_addr, dst_addr, bytes)
struct vir_addr *src_addr;                          /* source address */
struct vir_addr *dst_addr;                          /* destination address */
vir_bytes bytes;                                    /* # of bytes to copy */
{
  struct vir_addr *vir_addr [2];                    /* virtual addresses */
  phys_bytes phys_addr[2];                          /* absolute addresses */
  int seg_index;                                    /* memory segment */
  int i;                                            /* _SRC_ or _DST_ */

  /* Copy count should be greater than zero. */
  if (bytes <= 0) return(EDOM);

  /* Convert virtual addresses to physical addresses. */
  vir_addr[_SRC_] = src_addr;                       /* source virtual */
  vir_addr[_DST_] = dst_addr;                       /* destination virtual */
  for (i=_SRC_; i<=_DST_; i++) {                    /* convert to physical */

      /* Apply different mapping for different segment types. */
      switch((vir_addr[i]->segment & SEGMENT_TYPE)) {
      case LOCAL_SEG:                               /* text, stack or data */
          seg_index = vir_addr[i]->segment & SEGMENT_INDEX;
          phys_addr[i] = umap_local( proc_addr(vir_addr[i]->proc_nr),
              seg_index, vir_addr[i]->offset, bytes );
          break;
      case REMOTE_SEG:                              /* far memory areas */
          seg_index = vir_addr[i]->segment & SEGMENT_INDEX;
          phys_addr[i] = umap_remote( proc_addr(vir_addr[i]->proc_nr),
              seg_index, vir_addr[i]->offset, bytes );
          break;
      case BIOS_SEG:                                /* BIOS memory area */
          phys_addr[i] = umap_bios( proc_addr(vir_addr[i]->proc_nr),
              vir_addr[i]->offset, bytes );
          break;
      default:                                      /* illegal segment */
          return(EINVAL);
      }

      /* Check if mapping succeeded. */
      if (phys_addr[i] <= 0) return(EFAULT);
  }

  /* Now copy bytes between physical addresseses. */
  phys_copy(phys_addr[_SRC_], phys_addr[_DST_], (phys_bytes) bytes);
  return(OK);
}
```

Listing 3.3: The generic interrupt handler at the kernel. When a hardware interrupt occurs it is handled according to the interrupt policy set by a user-space device driver with the SYS_IRQCTL system call.

```
PUBLIC int generic_handler(hook)
irq_hook_t *hook;
{
  /* Get interrupt policy shorthands for convenience. */
  irq_policy_t policy = irqtab[hook->irq].policy;        /* IRQ policy flags */
  int proc_nr = irqtab[hook->irq].proc_nr;               /* process to notify */
  long port = irqtab[hook->irq].port;                    /* register for I/O */
  phys_bytes addr = irqtab[hook->irq].addr;              /* address at driver */
  long mask_val = irqtab[hook->irq].mask_val;            /* bit mask or value */

  /* Read a value from the given port. Possibly echo or strobe it back. */
  if (policy & (IRQ_READ_PORT|IRQ_STROBE|IRQ_ECHO_VAL)) {
      switch(policy & (IRQ_BYTE|IRQ_WORD|IRQ_LONG)) { /* port granularity */
      case IRQ_BYTE: {                                   /* byte values */
          u8_t byteval = inb(port);
          if (policy & IRQ_STROBE) outb(port, byteval | mask_val);
          if (policy & IRQ_ECHO_VAL) outb(port, byteval);
          if (policy & IRQ_READ_PORT)
              phys_copy(vir2phys(&byteval), addr, sizeof(u8_t));
          break;
      } case IRQ_WORD: {                                 /* word values */
          ...                                            /* like above */
          break;
      } case IRQ_LONG: {                                 /* long values */
          ...                                            /* like above */
          break;
      } default: /* do nothing */ ;                      /* wrong type flags */
      }
  }

  /* Write a value to some port. Cannot both read and write. */
  else if (policy & (IRQ_WRITE_PORT)) {
      switch(policy & (IRQ_BYTE|IRQ_WORD|IRQ_LONG)) { /* port granularity */
      case IRQ_BYTE: outb(port, (u8_t) mask_val); break;
      case IRQ_WORD: outw(port, (u16_t) mask_val); break;
      case IRQ_LONG: outl(port, (u32_t) mask_val); break;
      default: /* do nothing */ ;                        /* wrong type flags */
      }
  }

  /* Send a HARD_INT notification to allow further processing. */
  notify(proc_nr, HARD_INT);

  /* Possibly reenable interrupts depending on the policy given. */
  return(policy & IRQ_REENABLE);
}
```

The generic interrupt handler is shown in Listing 3.3. When an interrupt occurs it first looks up the policy from the global irqtab and then executes it. The supported interrupt policies are (1) do nothing, (2) read a port value, (3) optionally echo the value or strobe back with a supplied bit mask, or (4) write a value to some port. In all cases a HARD_INT notification is sent to the device driver for further handling. Finally, the policy may or may not reenable interrupts.

3.1.4 Getting system information

Some device drivers need to know about information that is only available within the kernel. Examples include process table information, boot monitor parameters, and the list of free memory chunks. A new system call, SYS_GETINFO, was created so that user processes can retrieve a copy of such information. By copying entire data structures user processes can safely perform operations on the data.

To get a copy of a data structure a process must supply a key for the data structure and a pointer to the location where the copy must be placed. No size is needed because all data structures have a fixed size. The kernel's handler function for the SYS_GETINFO call is shown in Listing 3.4. Figure 3.1 provides an overview of kernel information that can be obtained.

Convenient shorthands for the SYS_GETINFO system call were defined in <*minix/syslib.h*>. A copy of the process table (GET_PROCTAB) or the monitor parameters (GET_MONPARAMS), for example, can be requested with sys_getproctab() and sys_getmonparams(), respectively.

The SYS_GETINFO system call was not only used to remove kernel dependencies, but also allowed several new applications. Subsection 4.1, for example, discusses a new information servers that uses this call for debugging dumps. Subsection 5.4.3 discusses how kernel diagnostics are stored in local buffer and can be obtained with SYS_GETINFO.

Key	**Data structure**
GET_PROCTAB	kernel process table
GET_MONPARAMS	monitor parameters
GET_KMESSAGES	kernel messages
GET_IRQTAB	interrupt policies
GET_KENVIRON	kernel environment
GET_PROC	single process slot
GET_IMAGE	system image table
GET_MEMCHUNKS	free memory chunks

Figure 3.1: The most important data structures that are supported by the SYS_GETINFO system call. A convenient shorthand function is defined for each key in <*minix/syslib.h*>.

Listing 3.4: Handler function for the SYS_GETINFO system call. This system call request a kernel data structure to be copied to a given address in the caller's address space.

```
PUBLIC int do_getinfo(m_ptr)
register message *m_ptr;                              /* request message */
{
  phys_bytes src_phys, dst_phys;                      /* abs. copy addresses */
  size_t length;                                      /* # bytes to size */
  int proc_nr;                                        /* process to copy to */

  /* First get the process number and verify it. */
  proc_nr = (m_ptr->I_PROC_NR == SELF) ? m_ptr->m_source : m_ptr->I_PROC_NR;
  if (! isokprocn(proc_nr)) return(EINVAL);

  /* Set source address and length based on request type. */
  switch (m_ptr->I_REQUEST) {
  case GET_PROCTAB: {                                 /* process table */
      src_phys = vir2phys(proc);                      /* convert to physical */
      length = sizeof(struct proc) * (NR_PROCS + NR_TASKS);
      break;
  } case GET_MONPARAMS: {                             /* monitor parameters */
      src_phys = mon_params;                          /* already is physical */
      length = mon_parmsize;
      break;
  } case GET_KMESSAGES: {                             /* kernel messages */
      src_phys = vir2phys(&kmess);                    /* convert to physical */
      length = sizeof(struct kmessages);
      break;
  } case ... :                                        /* and so on ... */
  ...
  default:                                            /* illegal request */
      return(EINVAL);
  }

  /* Try to make the actual copy for the requested data. */
  if (m_ptr->I_VAL_LEN > 0 && length > m_ptr->I_VAL_LEN) return (E2BIG);
  dst_phys = numap_local(proc_nr, (vir_bytes) m_ptr->I_VAL_PTR, length);
  if (src_phys == 0 || dst_phys == 0) return(EFAULT);
  phys_copy(src_phys, dst_phys, length);
  return(OK);
}
```

3.1.5 Other support functions

Although this section covers the most important system calls to support user-space device drivers, several other system calls were needed as well. These are briefly discussed below. The reader is referred to Appendix C for a complete overview of MINIX' system calls.

The SYS_EXIT system call was added to cleanly shut down a user-space system service. This call is mainly used for the new shutdown sequence, which is discussed in Section 3.4. Furthermore, it may be used in case of errors and panics that should be handled locally.

The MEMORY device driver uses the new SYS_KMALLOC system call to allocate a chunk of memory for a RAM disk when MINIX is boot. This concerns a one-time, static allocation. The call returns a REMOTE_SEG selector that can be used with the SYS_VIRCOPY system call. The user-space MEMORY driver is discussed in Section 5.2.

The SYS_PHYS2SEG call is used by the TTY device driver to use the video RAM. The call adds a segment descriptor for the video memory to the TTY driver's local descriptor table (LDT) to grant direct access from user space. This is explained in Section 5.4.

Finally, several existing system calls were also updated to provide better support for user-space device drivers. For example, the CLOCK's alarm functionality was fully revised and provides a new type of alarm, which is discussed in Section 4.3.

3.2 Interprocess communication

Interprocess communication (or IPC for short) allows system servers to cooperate with each other and the kernel. IPC in MINIX is characterized by a client-server approach based on message passing. It is done by copying a request message from one process to another and awaiting the response. This section discusses MINIX' low-level IPC facilities and presents various improvements that were made.

3.2.1 Rendezvous message passing

The exchange of messages in MINIX is characterized by rendezvous message passing. Rendezvous is a two-way interaction without intermediate buffering. The interaction is fully synchronous, which means that the first process that is ready to interact, blocks and waits for the other [17]. When both processes are ready the message is copied from the sender to the receiver, and both can resume execution.

Although rendezvous message passing is easier to implement than a buffered message passing scheme, it is less flexible and sometimes even inadequate because the sender has to wait for the receiver to accept a message. While it is OK to block a user-space process when system services are requested, it is unsatisfactory to block a kernel task when it wants to send a message and the receiver is not ready. MINIX 2.0.4 takes special measures to prevent the latter, as is discussed in Subsection 3.3.

A nuisance related to this is that the exact ordering of messages between processes is important to prevent deadlocks. Although the kernel checks for deadlocks by scanning the send queue of the destination process in mini_send(), this does not solve the nature of the problem, that is, dealing with cyclic dependencies. The error ELOCKED is returned if a deadlock is found, but it would be better to structurally prevent deadlocks from occurring. MINIX does this by using a partial message ordering in which communication is mostly driven from the top.

In general, messages are sent in one direction, so that if process X calls Y, then Y may never call X. User processes, for example, may only send to the MM and FS; servers, including the MM and FS, can send to device drivers and tasks; device drivers, in turn, may call on the tasks, such as the CLOCK or SYS task. This is shown in Figure 2.4 in Subsection 2.3. Exceptions to this rule are allowed and exist, but cyclic dependencies are carefully prevented.[1]

3.2.2 Implementation of rendezvous IPC

Rendezvous message passing is implemented by means of a SEND, RECEIVE or BOTH system call. The calls are done with a *software interrupt*, that is, by trapping to the kernel with an INT instruction. This is conveniently hidden by the library functions send(), receive() and sendrec() of the run-time system (RTS). The RTS library can be found in *src/lib/i*86/rts/.* The trap is caught and handled in the kernel by the assembly routine s_call() defined in *src/kernel/mpx.s.*

The assembly code saves the machine state and calls the C function sys_call() in *src/kernel/proc.c* for further handling. This function verifies the system call parameters and calls mini_send(), mini_rec() or both to do the actual message passing. This may affect the scheduling queues in several ways. In mini_rec(), the caller is queued and blocked when no message from the desired source is available. In mini_send(), the same happens when a message cannot be delivered; if it can be delivered, the destination process is unblocked. When the system call is finished the assembly routine s_call() restarts the process that is scheduled next.

Problems with MINIX' original system call implementation

Analysis of the system call implementation in MINIX 2.0.4 revealed several shortcomings and room for improvement. First of all, a serious security flaw was found in the function sys_call(). Although it makes sure that user processes can only do a sendrec() system call so that they stall waiting for a reply from the MM or FS, all other processes can also use the send() and receive() functions. While it is not a bad thing to protect the MM and FS from malicious user processes, it is much more important to protect the kernel

[1]The FS, for example, normally only receives from the MM, but one message is exchanged in the opposite direction to synchronize the MM and FS when MINIX boots. Because this message is expected it does not interfere with MINIX' partial message ordering and cannot cause a deadlock.

from being blocked. The setup of MINIX 2.0.4, however, allows a server to block a kernel task by calling send() and not doing the corresponding receive() because servers are trusted. As more servers gets written this policy needs to be changed.

Furthermore, it is not possible to restrict interprocess communication in MINIX 2.0.4. Apart from a hard-coded restriction that ensures that user processes can only send to the MM and FS, all communication is allowed. System services thus should be able to deal with unexpected requests from arbitrary processes.

Another issue with sys_call() is that the function does not check for illegal system call numbers. Because of the current setup, an illegal call number is interpreted a RECEIVE system call. This merely blocks the caller if no message is available, but more serious scenarios can be thought of. If a request message is received while the caller was not expecting it, the request may be dropped without sending a response, effectively blocking the process that sent the request.

The fact that rendezvous message passing is synchronous makes it hard to use and sometimes even inadequate if processes must be sure that a message is delivered instantly. This is an issue of asynchronous trust [18]. Subsection 3.3 discusses how the kernel circumvents this issue with ad hoc implementations to deal with asynchronous events. Processes that are not part of the kernel cannot apply the same tricks, though, and thus can only hope that their system call succeeds.

3.2.3 A revision of MINIX' system call handler

The sys_call() function was fully rewritten to fix the shortcomings described above, to realize nonblocking system calls, to restrict communication between processes in a generic way, and to return proper error codes. The new sys_call() function is shown in Listing 3.5.

A small but important check now makes sure that kernel tasks can no longer be blocked. It is now required to use sendrec() for all calls to the kernel, so that kernel tasks can always reply. This is similar to the check whether the caller is a user process. Using send() or receive() is no longer allowed and results in an ECALLDENIED error. The check to see if src_dst is a valid process number was retained, but now returns EBADSRCDST if a problem is detected.

The next step is to see if the system call is known and to try to perform the request. This is done in a switch statement that makes the code more readable and easily allows for new system calls—should this be necessary. As in MINIX 2.0.4, the only system calls that exist are sending and receiving messages. However, the *nonblocking* variants NB_SEND and NB_RECEIVE are now supported. This is discussed below. Furthermore, illegal system calls are now detected and result in the error EBADCALL.

Listing 3.5: The sys_call() function was rewritten to fix several shortcomings and to support nonblocking system calls. It now now returns proper error codes, prevents kernel tasks from being blocked, and restricts communication between arbitrary processes. Nonblocking system calls have a NON_BLOCKING flag.

```
PUBLIC int sys_call( call_nr , src_dst , m_ptr)
int  call_nr ;                                        /* SEND, RECEIVE, BOTH */
int  src_dst ;                                        /* source or destination */
message *m_ptr;                                       /* message in caller's space */
{
  register struct proc * caller_ptr = proc_ptr;
  int req_function = call_nr & SYSCALL_FUNC;
  int may_block = ! ( call_nr & NON_BLOCKING);
  int mask_entry;                                     /* bit in send mask */
  int result ;                                        /* system call result */

  /* Protect the system by requiring caller to await the result . */
  if (( iskernel(src_dst )|| isuserp( caller_ptr )) && req_function != BOTH) {
      result = ECALLDENIED;                           /* BOTH was required */
  }
  /* Verify that source/ destination process is valid . */
  else if (! isoksrc_dst(src_dst)) {
      result = EBADSRCDST;                            /* nonexistent process */
  }
  /* Check if the request is known and try to perform it . */
  else {
      switch(req_function) {
      case SEND:                                      /* send a message */
          /* Fall through, SEND is done in BOTH. */
      case BOTH:                                      /* send and receive */
          if (! isalive (src_dst)) {
              result = EDEADDST;                      /* cannot send if dead */
              break;
          }
          mask_entry = isuser(src_dst) ? USER_PROC_NR:src_dst;
          if (! isallowed( caller_ptr ->p_sendmask, mask_entry)) {
              result = ECALLDENIED;                   /* denied by send mask */
              break;
          }
          result = mini_send( caller_ptr , src_dst , m_ptr, may_block);
          if ( req_function == SEND || result != OK) {
              break;                                  /* done, or SEND failed */
          }                                           /* BOTH falls through */
      case RECEIVE:                                   /* receive a message */
          result = mini_rec( caller_ptr , src_dst , m_ptr, may_block);
          break;
      default:
          result = EBADCALL;                          /* illegal system call */
      }
  }
  return( result );                                   /* system call status */
}
```

A new feature of sys_call() are send masks that provide fine-grained control over which processes may communicate. Send masks are per-process bit masks that indicate to which processes a given process may send. They are defined in *src/kernel/sendmask.h*. If a SEND or BOTH system call is done, it is checked whether the bit for src_dst is enabled in the caller's send mask. The error ECALLDENIED is returned otherwise. Below this protection mechanism is discussed in more detail.

Non-blocking system calls

While MINIX does not provide asynchronous message passing to prevent all troubles of buffering messages, it turned out that nonblocking message passing was relatively easy to implement. The implementation is based on a flag, NON_BLOCKING, that can be applied to the system call number. This flag as well as two bit masks for taking the system call number apart, SYSCALL_FLAGS and SYSCALL_FUNC, are defined in *<minix/com.h>*.

The nonblocking variants of send() and receive() are called nb_send() and nb_receive(), respectively. Function prototypes for these functions were defined in *<minix/syslib.h>*, and the RTS library, *src/lib/i*86/rts/_sendrec.s*, was updated to include the implementation.

In sys_call(), the requested message passing function is determined by masking the system call number with SYSCALL_FUNC. A Boolean indicating whether the flag is set, is passed to mini_send() or mini_rec(). If the NON_BLOCKING flag is set and the source or destination process is not ready, mini_send() and mini_rec() return ENOTREADY instead of blocking the caller.

The nonblocking system call variants can be used in specific cases where blocking is no option. This is especially useful for user-space device drivers and servers that cannot check another process' state in the process table. The alternate approach that they can now use is *polling*. Examples of how the nb_send() and nb_receive() calls are used for MINIX' new shutdown sequence can be found in Section 3.4 and 4.1.

System call protection

Instead of restricting only user processes in whom they can send to, a more generic protection mechanism was realized. All processes now have a new process table entry to store a send mask that determines to which processes they are allowed to send. If a processes wants to send to some other process, it is first verified that the corresponding bit in the process' send mask is set.

Note that there is no separate receive mask to indicate from which processes a given process may receive messages. This is implied by the send mask. On the one hand, if process A may send to process B, it is assumed that B is allowed to receive A's messages. On the other hand, if process B is not allowed to receive from process A, the bit for B in A's send mask simply can be unset so that A cannot send to B.

The send mask has distinct bits for all system processes, that is, kernel tasks, device drivers, servers and INIT, as well as one bit for user processes. Because all system services are statically included in the system image and have a known process number, the send mask definitions are fixed.

The actual send mask definitions for all system services are kept in a separate file, *src/kernel/sendmask.h*. The send masks are defined with help of two default masks, ALLOW_ALL_MASK and DENY_ALL_MASK, and two bit operations to allow or deny sending to individual processes.

In *src/kernel/table.c*, the newly defined send masks are placed into the image table, which is used to build the initial process table in main() when MINIX starts up. This way all system services automatically have their send mask installed.

The send mask for user processes is set in a different way. User processes are always created via a SYS_FORK system call. The send mask for them is set to USER_PROC_SENDMASK in the function do_fork() in the SYS task. Similarly, if a user process exits, the process table slot is cleared in do_xit() and the send mask is set to DENY_ALL_MASK.

One temporary exception was created to make the network server, INET, work. This server starts as an ordinary user process, but upgrades itself to a server by means of a SYS_SVRCTL system call. Because the send mask for user processes is too strict for the INET server, more rights were given in the system call's handler function. As this exception is temporary, the ALLOW_ALL_MASK was set.

The definition of a better interface for dynamic control of system services is part of future research. The SYS_SVRCTL function, for example, must be updated with a mechanism to dynamically set send masks.

Changes to the message passing functions

The functions mini_send() and mini_rec() also were updated in several ways. First of all, the functions accept another parameter from sys_call() that tells whether the system call may block. If the source or destination is not ready and blocking is not allowed, ENOTREADY is returned.

In MINIX 2.0.4, mini_send() contains a hard-coded check to ensure that user processes only send to FS and MM. This has been replaced by the generic check in sys_call(), as discussed above.

The function mini_rec() was modified in two ways. It contained some code that was only executed if the MM blocks, to check whether there are pending kernel signals that should be delivered to the MM at that point. The check is no longer necessary because of a new way to notify the MM about pending signals. This is discussed below. Furthermore, a check for blocked 'interrupts' was generalized to check for blocked notifications of any kind. Both modifications are discussed in more detail in the next section.

3.3 Dealing with asynchronous events

While most communication in MINIX is driven from the top, the kernel has to deal with events such as expired timers and hardware interrupts that must be communicated to processes at a higher level. Because these system events are inherently asynchronous—in contrast to MINIX' rendezvous message passing—special measures are taken to prevent problems. In the previous section it is already discussed that MINIX uses a partial message ordering to prevent deadlocks. In addition, a special construct is used in the kernel to make sure that a bottom-up message cannot block the kernel.

The asynchronous events that require message passing to a higher level process include hardware interrupts,[2] exceptions, kernel signals, alarms, and MINIX shutdown. The old and new implementation of each of these events is discussed below.

3.3.1 Original implementation

Hardware interrupts originate in I/O devices when they must be serviced. Once the processor receives an interrupt request (IRQ) it finishes its current instruction, and calls the interrupt service routine that corresponds to the request vector. These routines are located in *src/kernel/mpx.s* and loaded into the interrupt vector table during startup. The service routine saves the entire state on the stack, and executes the interrupt handler that is registered by the device driver that handles the IRQ line. Interrupt handlers generally do little work and notify the device driver for further processing. Because of the asynchronous nature of hardware interrupts *race conditions* may exist when passing messages within the interrupt handler. Hardware interrupts, for example, can be nested and may interfere with other process switching functions, such as sys_call() and sched(). Therefore, a special function, interrupt(), is used to notify the device driver of the interrupt by means of a HARD_INT message. If a race condition is found in interrupt() the call is put on the 'held' queue to be flushed by unhold() at the next noncompeting restart. If the driver is ready to receive the HARD_INT message it is directly delivered, and the driver is scheduled to run next. Otherwise, the message is marked 'blocked' and delivered as soon as when the driver does a receive() call with source HARDWARE or ANY. Once the interrupt handler has finished, the service routine returns by restarting the process that is scheduled next.

Exceptions are handled similarly to hardware interrupts. When an exception is encountered, the processor switches context and executes one of the exception handlers in *src/kernel/mpx.s*. The handler, in turn, calls the function exception() in *src/kernel/exception.c* with the exception number as an argument. Exceptions in user processes are converted to signals, while

[2]Software interrupts are synchronous events in that the system call that causes the INT instruction to trap to the kernel is explicitly executed by the caller.

exceptions in system processes cause a panic() and shutdown MINIX. Both signal handling and MINIX' shutdown are discussed below.

Signals can originate in various places,[3] but are always forwarded to the kernel where they are handled by sending a message to MM. Regular rendezvous message passing is used for this, but only once it has been verified that the MM is ready to receive the message. Most of the work is done by two functions in *src/kernel/system.c*. If a process is signaled, the function cause_sig() updates the process' signal mask and sets the PENDING flag. Then it carefully checks whether the MM can be informed about the pending signals. The function inform() is called directly from cause_sig() if the MM is idle and waiting for a message. Otherwise, this is done from mini_rec() as soon as the MM blocks waiting for a message and signals are pending. The function inform() pushes all pending signals to the MM by sending a message for each process with the flag PENDING.

Alarms originate in the CLOCK task when an alarm timer expires. MINIX 2.0.4 has two types of alarms, namely user alarms and synchronous alarms. The functions to set and expire alarms are contained in *src/kernel/clock.c*. The CLOCK causes a SIGALRM signal when a user alarm expires. This is handled like all other kernel signals. When a synchronous alarm expires the CLOCK delegates the work to the synchronous alarm task, SYN_AL, that sends a CLOCK_INT message to the process that requested the alarm. The CLOCK task notifies the SYN_AL task with help of the interrupt() function so that it cannot block. Unfortunately, the SYN_AL task cannot use this function to notify the requesting process of the alarm and uses an ordinary blocking send() call instead.[4]

Finally, a shutdown or reboot also is an asynchronous event that, in principle, should be communicated to all processes to allow them to cleanly exit. MINIX 2.0.4 does not do so, however. The shutdown code is contained in the function wreboot() in *src/kernel/keyboard.c*. It stops a handful of tasks that are part of the kernel by calling their stop_task() functions, and then directly brings down MINIX. User-space system services are simply ignored.

Problems with the original approach

Unfortunately, MINIX 2.0.4 handles each of the asynchronous events described above in a different way. The events that are described are solved with ad hoc implementations while a uniform approach is possible. Moreover, there are several shortcomings that pose serious problems.

[3]Examples of signals and their origin: A segmentation violation by a user process results in a SIGSEGV, the FS causes a SIGPIPE on a broken pipe, the TTY driver causes a SIGINT upon getting a 'delete', the CLOCK task sends SIGALRM when a user alarm expires, and user processes can use kill() to send signals to other processes.

[4]The interrupt() function of MINIX 2.0.4 can only deliver a single message type, HARD_INT and only works kernel tasks, because it directly assigns values to the message buffer of the receiver, which is not possible for processes in a different address space.

Because the SYN_AL task uses ordinary send() calls, it will be blocked until a receive() is called by the process that requested the alarm. This situation can easily delay synchronous alarms requested by other processes, or, even worse, could block the SYN_AL task forever when the corresponding receive() is not done at all. This is a problem of *asynchronous trust* [18], which becomes more severe with the amount of processes that rely on synchronous alarm. The setup works for MINIX 2.0.4 where only the INET server uses synchronous alarms, but is not suitable when numerous device drivers also rely on the synchronous alarm functionality—the latter is explained in detail Section 4.2.

The shutdown code in MINIX 2.0.4 does not notify user-space device drivers and servers, because direct function calls are not possible across address spaces, and MINIX 2.0.4 does not have a message passing construct to safely to do. This is problematic because important system processes are not informed about MINIX' shutdown, and thus cannot run their cleanup code. The FS, for example, cannot synchronize when MINIX is brought down due to a panic() and thus is likely to loose data.

The next subsection describes a uniform approach for handling asynchronous events that solves all of these problems.

3.3.2 A new notification construct

A new message passing construct, notify(), was designed to inform all types of processes about asynchronous system events in a unified way. Initially, the notify() function existed next to the interrupt() function, but these were merged into a single function that can be used for all system notifications. The resulting function was named notify() because that name better covers its purpose. It is shown in Listing 3.6.

The new notify() function has a similar semantics as the interrupt() function that is discussed in Subsection 3.3.1. It provides kernel tasks with a mechanism that realizes nonblocking message passing without buffering messages.[5] The new notify() function offers important extended functionality compared to the interrupt() function of MINIX 2.0.4, however. First, it can be used to notify any type of process and not just kernel tasks. Second, in addition to hardware interrupts, it supports notifications for any kind of system event, including the ones that are discussed above.

To prevent any confusion it has to be noted that the notify() construct is not a system call like send() or receive(). Instead notify() is built around send() and makes sure it does not block by inspecting the status of the receiving process. Therefore, it can only used within the kernel.

[5]This is possible because similar notifications that are sent to the same process in a short period of time may be united. If a notification message cannot be delivered this is recorded in a bit mask, which has only a single bit for each notification type.

Listing 3.6: The notify() function that allows kernel tasks to safely inform all types of processes about any kind of system event without the risk of being blocked. Race conditions are handled by putting the call on a held queue. If a process is not ready to receive the notification the blockage is recorded.

```
PUBLIC void notify(proc_nr, notify_type)
int proc_nr;                                          /* process to notify */
int notify_type;                                      /* type of notification */
{
  register struct proc *rp;                           /* pointer to process */
  message m;                                          /* notification message */
  unsigned int notify_bit;                            /* bit in bit mask */

  /* See if notification type is known. */
  notify_bit = (unsigned int) (notify_type - NOTIFICATION);
  if (notify_bit >= NR_NOTIFICATIONS) {               /* check validity */
      panic("Incorrect notification type", notify_bit);
      return;
  }
  rp = proc_addr(proc_nr);

  /* Check for races with other process switching functions. */
  if (k_reenter != 0 || switching) {
      lock();                                         /* disable interrupts */
      if (! rp->p_notify_held) {                      /* add to held queue? */
          if (held_head != NIL_PROC) held_tail->p_nextheld = rp;
          else                       held_head = rp;
          held_tail = rp;
          rp->p_nextheld = NIL_PROC;
      }
      set_bit(rp->p_notify_held, notify_bit);         /* update held mask */
      unlock();                                       /* reenable interrupts */
      return;                                         /* retry later */
  }
  switching = TRUE;                                   /* protect rest of call */

  /* If process is not ready to receive HARDWARE message, record the blockage. */
  if ( ! isreceiving(rp->p_flags) || ! isrxhardware(rp->p_getfrom)) {
      set_bit(rp->p_ntf_blocked, notify_bit);         /* update blocked mask */
      switching = FALSE;                              /* end protection */
      return;                                         /* handle in mini_rec */
  }

  /* Destination is awaiting message. Send notification and announce process ready. */
  m.m_source = HARDWARE;                              /* construct message */
  m.m_type = notify_type;                             /* set notification */
  CopyMess(HARDWARE, proc_addr(HARDWARE),&m,rp,rp->p_messbuf);
  rp->p_flags &= ~RECEIVING;                          /* message delivered */
  clear_bit(rp->p_ntf_blocked, notify_bit);           /* update blocked mask */
  ready(rp);                                          /* receiver now can run */
  pick_proc();                                        /* schedule new process */
  switching = FALSE;                                  /* end protection */
}
```

Implementation details

The implementation required several small changes to MINIX source code. The process structure in *src/kernel/proc.h* was updated to include bit masks for 'held' and 'blocked' notifications. All notification types are defined as incrementing numbers in <*minix/com.h*>. They are relative to NOTIFICATION to prevent interference with existing message types. The implementation is found the functions notify(), unhold(), and mini_recv() in *src/kernel/proc.c*. The most important differences compared to MINIX 2.0.4 are that different types of notifications are distinguished and that CopyMess() is used to copy the notification message to any type of process.

The notify() function circumvents potential race conditions so that it can be used safely from both interrupt handlers and the task level. If a notification competes with other process-switching functions, such as system calls and hardware interrupts, a bit is set in the 'held' bit mask of the destination process, and the call is put on the 'held' queue. The unhold() function flushes the held notifications at the next noncompeting restart. It was generalized to support different kinds of notifications.

Furthermore, the notify() function is nonblocking. If the receiver is not ready to receive a notification message, a bit is set in the 'blocked' bit mask of the destination process. This bit mask is checked later when the destination process calls receive(). If a blocked notification is found a notification message is locally constructed and copied to the receiver.

Although the 'held' and 'blocked' bit masks obviate the need for a buffered message passing scheme, there is a minor trade-off. With only one bit per type of notification it is not possible to store additional information such as the sender of the notification message or parameters. Therefore, the only information that can safely be passed is the notification type and all notification messages have HARDWARE as their message source. This property turns out to be very useful for exception handling within user-space device drivers as is discussed in Section 4.3.

3.3.3 Handling of asynchronous events

Hardware interrupts basically are handled as before. The HARD_INT message type was redefined to become a valid notification type. Instead of calling interrupt(proc_nr), the interrupt handlers now call notify(proc_nr, HARD_INT) to alert a device driver about the hardware interrupt. As a positive side-effect the code has become more readable.

MINIX' signal handling was greatly simplified by removing several exceptional cases. A new notification type KSIG_PENDING was defined to replace the KSIG message type. The code that checks whether the MM is ready to receive a message could be removed from both cause_sig() and mini_recv(). A single notify(MM_PROC_NR, KSIG_PENDING) call in cause_sig() now does the job.

The function inform() that pushes pending signals to the MM once it is ready to receive them was also removed. In the new approach the MM repeatedly polls the kernel for a pending signal when it receives a KSIG_PENDING notification. This places the responsibility for handling signals at the MM—where it belongs.

The problem with the synchronous alarm task was solved by removing the SYN_AL task altogether. The CLOCK task no longer calls on the SYN_AL task to make a potentially blocking call to send() when an alarm expires. Instead, because the new notify() function cannot block, the CLOCK now dares to alert the process that requested the alarm itself by sending a SYN_ALARM notification.

Finally, MINIX shutdown sequence was fully revised. A HARD_STOP notification type was defined for this. Instead of making direct function calls, the process table is now scanned for system services that are still active. Each active process is notified about the upcoming shutdown, and is allowed some time to clean up. The notifications are ordered according to the dependencies that exist between different types of processes, so that higher level processes can still be serviced by the lower level ones. The new shutdown sequence is described in detail in Section 3.4.

3.4 A new shutdown sequence

MINIX' shutdown sequence is briefly discussed in the previous section as an example of an asynchronous event that requires bottom-up communication. Here, the details are further studied and a new shutdown sequence that cleanly brings down MINIX and all system services is explained.

3.4.1 Original implementation

The shutdown sequence of MINIX 2.0.4 is straightforward. It can be triggered by several events. The user, for example, can issue a `'Ctrl-Alt-Del'` or type `'shutdown'` or `'reboot'` to bring MINIX down. A kernel panic also causes MINIX to shutdown. If the event that triggers the shutdown is detected outside the kernel, the SYS_ABORT system call is used to notify the kernel. The MM, for example, uses this call to process a `'shutdown'`.

In all cases, the kernel gets to run the shutdown code in wreboot() in *src/kernel/keyboard.c*. The function first masks all IRQ vectors so that device drivers can no longer be interrupted. Then it allows several device drivers to run their cleanup code by calling their stop_task() functions. For example, floppy_stop() stops the motor of the floppy disk drive and cons_stop() selects the primary console to be visible.

If the reason to shut down was a panic, the user is allowed to make debug dumps before rebooting MINIX. The debug dumps are easily handled in the wreboot() function, because all needed functions and data structures are in the kernel address space. The functions that make the actual debug dumps are contained in *src/kernel/dmp.c*.

Finally, wreboot() brings down MINIX. The action that is taken depends on the system's environment and the reason to shutdown that contained in the argument that is passed to wreboot(). Known values are contained in *<unistd.h>*. RBT_MONITOR, RBT_PANIC, RBT_REBOOT, and RBT_HALT all try to return to the boot monitor with level0(monitor) and possibly run some code at the boot monitor. If it is not possible to return to the boot monitor because the environment variable mon_return is 0 or if how is RBT_RESET, a hard reset is issued with level0(reset).

Problems with the original design

The most important issue with the original setup has already been discussed in Section 3.3. The shutdown code in MINIX 2.0.4 does not notify user-space device drivers and servers. This is problematic because important system services are not informed about MINIX' shutdown and cannot run their cleanup code. The FS, for example, cannot synchronize when MINIX is brought down due to a kernel panic and thus is likely to loose data.

Another problem is that the shutdown code in MINIX 2.0.4 is responsible for quite some dependencies when device drivers are moved out of the kernel. The stop_task() functions can no longer be called for user-space drivers, because function calls across address spaces are not possible without bypassing the MMU's normal protection mechanisms.

Finally, the location of the shutdown code is not logical at all. Although shutting down is an important, system-wide event, the shutdown code in MINIX 2.0.4 is part of the TTY driver. The historic reason for this may be that the the `'Ctrl-Alt-Del'` command is issued via the keyboard, but this does not reflect the other ways to shutdown MINIX. In most cases, the MM initiates a shutdown via a sys_abort() system call.

3.4.2 New setup to cleanly bring down MINIX

MINIX' shutdown was completely redesigned to solve all problems that are listed above. The new shutdown sequence heavily relies on the new notify() message passing construct that was discussed in Section 3.3. This solves the problem relating to the dependencies and allows to inform all system services about MINIX' shutdown, instead of kernel tasks only. The new shutdown sequence is illustrated in Figure 3.2.

The shutdown sequence based on three functions that are executed in turn: prepare_shutdown(), stop_sequence() and finally shutdown(). These functions are discussed below. Because shutting down is the opposite of starting MINIX the new code was placed in *src/kernel/main.c*.

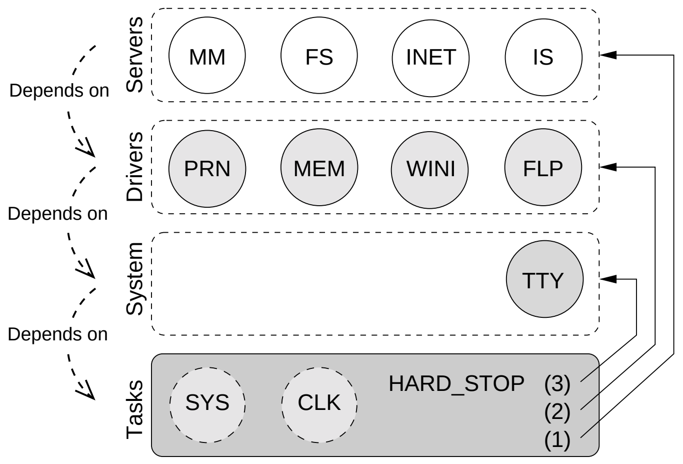

Figure 3.2: MINIX' new shutdown sequences notifies all user-space system services in order of possible dependencies. The TTY is notified last so that it can output diagnostic messages during the shutdown sequence.

Preparing MINIX' shutdown

The function prepare_shutdown() that initiates MINIX' shutdown sequence is shown in Listing 3.7. The function first checks a global flag, shutting_down, to prevent recursive shutdowns. This could, for example, happen when the MM or FS panics during their cleanup. It then disables most interrupts, like wreboot() does in MINIX 2.0.4. The CLOCK_IRQ is excluded, however, because the hardware clock is needed by the CLOCK task to keep track of the watchdog timers that are used for the stop sequence.

The next step is to check whether MINIX is brought down due to a panic. If this is the case and if the TTY driver is ready the user is allowed to make debug dumps before shutting down. Debug dumps are discussed in Section 4.1. The kernel carefully verifies that the TTY driver is still alive and ready by using the new, nonblocking nb_send() function.[6] The function prepare_shutdown() returns after sending a PANIC_DUMPS request message so that the TTY can run. Control is only given up if the message is successfully sent. The TTY is responsible to restart the shutdown sequence with a SYS_ABORT system call when the user is done. More information on debugging dumps on a panic can be found in Subsection 4.1.4.

Finally, prepare_shutdown() sets the flag shutting_down to inform other kernel part about the upcoming shutdown. Then it decides if stop_sequence() should be run. Naturally, the stop sequence should be run whenever possible. However, it is skipped to prevent doing more harm once a CPU

[6]The new notify() function is not useful here, because it does not provide status information on the receiver while it must be verified that the TTY driver is still alive and ready to handle the request.

Listing 3.7: The new function prepare_shutdown() disables most interrupts, checks if a panic occurred and whether debugging dumps should be allowed, and starts the stop sequence unless a CPU exception was raised.

```
PUBLIC void prepare_shutdown(how)
int how;                                        /* reason to shut down */
{
  if (shutting_down) return;                    /* shutdown yet active */

  /* Mask interrupts, but keep clock ticking for timers. */
  outb(INT_CTLMASK, ~(1<<CLOCK_IRQ));

  /* See if a panic occured and debug dumps are possible. */
  if (how == RBT_PANIC) {                       /* a panic occurred */
      message m;                                /* assemble a message */
      m.m_type = PANIC_DUMPS;
      if (nb_send(TTY, &m) == OK)               /* try, but don't block */
          return;                               /* await abort from TTY */
  }

  /* Now start MINIX' shutdown. Try to stop system services. */
  notify(TTY, HARD_STOP);                       /* to primary console */
  shutting_down = TRUE;                         /* set shutdown active */
  tmr_arg(&shutdown_timer)->ta_int = how;       /* pass shutdown reason */

  if (skip_stop_sequence) {                     /* set in exception() */
      kprintf("Fatal exception; skipping stop sequence.\n", NO_ARG);
      shutdown(&shutdown_timer);                /* directly shut down */
  } else {
      kprintf("Notifying services about MINIX' shutdown.\n", NO_ARG);
      stop_sequence(&shutdown_timer);           /* run stop sequence */
  }
}
```

exception was raised for a kernel process. Segmentation faults, for example, tend to persist when the stop sequence gets to run because kernel tasks are nonpreemptable and are scheduled with the highest priority.

Stopping all system services

The function stop_sequence() that tries to cleanly stop all system services before shutting down is shown in Listing 3.8. All system services are notified with a HARD_STOP message and given some time to clean up and exit. The notifications are sent in the order of possible dependencies, starting at the highest level and ending at the lowest. This means that the FS server, for example, can still use the AT_WINI driver to cleanly shutdown. This is illustrated in Figure 3.2. Unfortunately, dependencies between similar types of processes are not resolved this way. Therefore, the TTY driver was placed at a lower level than other other device drivers.

Listing 3.8: The new function stop_sequence() tries to cleanly stop all system services before shutting down. Only when all processes have been exited—either gracefully or forcibly—MINIX is really shut down.

```
PUBLIC void stop_sequence(tp)
timer_t *tp;
{
  static int level = PPRI_SERVER;                    /* highest level first */
  static struct proc *p = NIL_PROC;                  /* next process to stop */
  static char *types[] =                             /* all process types */
      {"task","system","driver","server","user"};

  /* See if previous process exited. Else force it to exit. */
  if (p != NIL_PROC) {                               /* skip first time */
      kprintf("[%s]\n", isalivep(p) ? "FAILED" : "OK");
      if (isalivep(p))                               /* check if exited */
          clear_proc(p->p_nr);                       /* force process exit */
  }

  /* Lookup next process to exit. Shutdown when all done. */
  if (p == NIL_PROC) p = BEG_PROC_ADDR;
  while (TRUE) {                                     /* stop all processes */
      if (isalivep(p) && p->p_type == level) {       /* next process found */
          kprintf("- Stopping %s ", p->p_name);
          kprintf("%s ... ", types[p->p_type]);
          shutdown_process = p;                      /* used in sys_exit() */
          notify(proc_number(p), HARD_STOP);         /* alert the process */
          set_timer(tp,get_uptime()+STOP_TICKS,stop_sequence);
          return;                                    /* allow it to stop */
      }
      p++;                                           /* check next process */
      if (p >= END_PROC_ADDR) {                      /* this level done? */
          level = level - 1;                         /* go to next level */
          p = BEG_PROC_ADDR;                         /* restart at begin */
          if (level == PPRI_TASK) {                  /* tasks remain alive */
              set_timer(tp, get_uptime()+HZ, shutdown);
              return;                                /* display output */
          }
      }
  }
}
```

When stop_sequence() is run for the first time, it starts at the beginning of the process table to find the next process that must be stopped. If a process is found its name and type are printed for the user's interest. A global variable, shutdown_process, is set to indicate which process is being stopped. This makes it possible to immediately continue the stop sequence once the process has exited. A watchdog timer with stop_sequence() as a watchdog function is set to make sure that the stop sequence continues within STOP_TICKS ticks if the process does not exit voluntarily. Then stop_sequence() returns so that the process to be stopped can be scheduled and can run its cleanup code.

Now two things can happen. If the process that is being stopped notices the HARD_STOP message it can clean up and stop with a SYS_EXIT system call. The function do_exit() that handles this system call continues the stop sequence if the shutting_down flag is set and the process that exited indeed is the shutdown_process. If the process does not exit within the timeout interval, the watchdog timer expires and will run its watchdog function.

In both cases, the stop_sequence() function is run again. If the process to be stopped exited gracefully this is reported to the user. If a disobedience is detected, the failure is reported and the process is forcibly exited. Then the stop sequence looks up the next process to stop. It continues where the last search ended and proceeds to next level if there are no more active process at the current level.

Finally, if the last level has been processed, that is, if all processes have exited, MINIX is brought down with a call to shutdown(). This is done by setting a watchdog timer to give the user some time to inspect the status of the entire shutdown sequence. The watchdog timer's argument is used to pass the shutdown status as before.

It has to be noted that the TTY driver must be the last process in this sequence so that it can display the stop status of each process at the primary console. The TTY thus resides at the end of the lowest level of user processes. When the TTY driver finally receive a HARD_STOP notification it directly displays its own stop status before exiting.

Shutting down MINIX

The last function, shutdown(), deals with returning to the boot monitor or doing a hard system reset. It contains the majority of the code that was part of wreboot() in MINIX 2.0.4. This code was not changed compared to the discussion in Subsection 3.4.1. Therefore, it is not treated here.

3.4.3 Future modifications

The new shutdown sequence is part of the kernel because system services have a special status in MINIX and cannot be managed in the same way as ordinary user processes. System services, for example, cannot do an ordinary exit() call, which is handled by the MM. Instead, they use the SYS_EXIT system call to directly tell the kernel that they want to exit.

As discussed in Section 7.3, there is no obvious reason for this special treatment. Therefore, the transformation of system services into ordinary user processes is part of future work. Once this has been done, a large part of the shutdown code can be removed from the kernel, for example, by setting up a new server that controls the shutdown sequence. The design that is presented in this section need not to be changed, though.

Chapter 4

New applications

This chapter discusses several new user-space applications. Section 4.1 describes a new server, IS, that allows the user to make debug dumps of various kernel data structures. Section 4.2 introduces a new library that allows user-space device drivers to use watchdog timers. Finally, Section 4.3 shows two new approaches for dealing with unresponsive hardware.

4.1 A new information server

MINIX allows the user to make to types of debug dumps of the kernel's process table fields. This can be done at any time by pressing a function key and when MINIX is aborted due to a kernel panic. `'F1'` shows the process table including program counter and stack pointer, user and system times, flags, messages being sent or received and process names. `'F2'` shows memory usage for text, data and stack segments of each process.

In MINIX 2.0.4 the debug dumps are entirely handled within the kernel. The code to make the actual debug dumps as well as the logic to determine that the user requested a debug dump are part of the TTY driver. Since this driver is transformed into a user-space device driver the kernel data structures can no longer be directly accessed.

4.1.1 Debug dumps in MINIX 2.0.4

In MINIX 2.0.4 debug dumps are a hard-coded feature of the TTY task that resides in the kernel. For each key that is struck, the function func_key() in *src/kernel/keyboard.c* is called to check whether is has a special purpose. A switch-statement based on the key's scan code checks for known function keys and dispatches the associated handler function if it detects one.

The handler functions for the debug dumps of the kernel's data structures are contained in *src/kernel/dmp.c*. Debug dumps of the process table and memory maps can be requested with `'F1'` and `'F2'`, which are mapped onto p_dmp() and map_dmp(), respectively. Debug dumps of other data structures are provided by the owner of the data structure. 'F5', for example, triggers a debug dump of the network statistics at the RTL8139 or DP8390 device drivers. If the RTL8139 driver is enabled 'F5' is mapped onto rtl8139_dump() which is contained in *src/kernel/rtl8139.c*.

The TTY driver thus touches kernel data structures and makes direct function calls to other drivers. This is possible in MINIX 2.0.4 because all drivers are part of the kernel and thus share the same address space.

Shortcomings of the original approach

The implementation of debug dumps in MINIX 2.0.4 has several shortcomings. First of all, the debug dumps are an integral part of the TTY device driver, while its primary task is to manage terminals. The TTY driver thus should not know about kernel data structures and functions of completely unrelated device drivers.

Another problem is that the current approach directly accesses crucial kernel data structures for noncrucial system services. A reasonable amount of code is in place to format and output the actual debug dumps with all risk of bugs. As a general rule, kernel data structures should only be accesses by trusted kernel task.

A final problem is that the TTY driver can no longer directly access kernel data structures and debug functions of other device drivers when it is moved to user space.

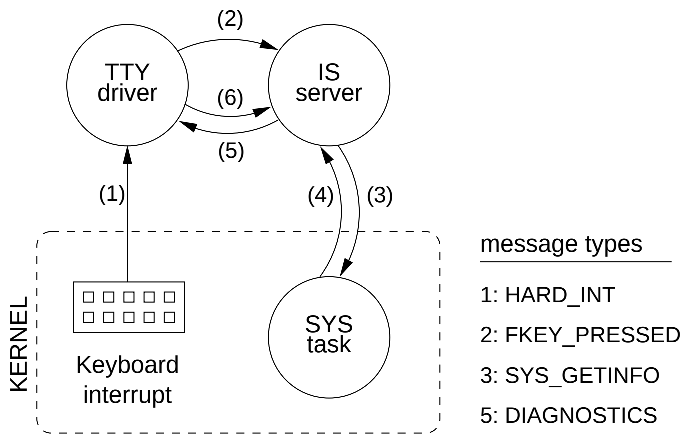

Figure 4.1: A distributed approach for making debug dumps. If a user presses a function key (1) an interrupt notification is sent to the TTY driver, (2) a function key notification is sent to the IS server, (3/4) a system call to the SYS task is done to get a copy of some kernel data structure, and (5/6) the TTY driver is asked to output the diagnostics. The precise message types are shown in the figure.

4.1.2 A distributed approach

The new, distributed approach for making debug dumps is illustrated in Figure 4.1. It can best be explained by tracing what happens if the user presses a function key to request a debug dump. As soon as key is pressed a hardware interrupt occurs which is caught by the kernel. The kernel's generic interrupt handler informs the TTY driver about the interrupt with a HARD_INT notification message in step 1.

The TTY driver notices that a function key has been pressed, so it calls the function func_key() in *src/drivers/tty/keyboard.c* to see if any process is interested. Subsection 4.1.3 discusses how processes can register for notifications of function key events. If a key in the `'F1'`-`'F12'` range is depressed, the IS server is alerted with a FKEY_PRESSED message in step 2.

When the IS server receive a function key notification, it dispatches to the function that handles the requested debug dump based on the function key's scan code. Since kernel data structures can no longer be directly accessed, the IS server makes a SYS_GETINFO system call to get a copy of the required kernel data. This is shown in steps 3 and 4. If the call was successful, the actual debug dump is made with help of the local copy.

Finally, the IS server uses the TTY driver to output the debug dump to the primary console. Redirecting output to the TTY driver is transparently done by the printf() function that is linked with all system servers. This uses a DIAGNOSTICS message as discussed in Subsection 5.4.3. The final steps, 5 and 6, are repeated until the entire dump has been printed.

Debug dumps available

This distributed approach made it relatively easy to create debug dumps and therefore was applied to several kernel data structures during the course of this master's project. Figure 4.2 provides an overview of the most important dumps that are currently supported by the IS server. Futhermore, the FS and MM provide debug dumps of their own data structures.

Key	**Debug dump**	**Information shown**
F1	Process table	including PC, SP, times, IPC flags, and names
F2	Memory usage	text, data and stack segments and total size
F3	System image	initial values for processes in the boot image
F4	Send masks	bits maps used to restrict SEND system calls
F5	Environment	(key,value)-pairs that are set at boot monitor
F6	IRQ policies	hardware interrupt policies per IRQ vector
F7	Kernel messages	diagnostic messages outputted by the kernel

Figure 4.2: Overview of the most important debug dumps supported by the new IS server. The function keys `'F1'` to `'F12'` have been registered by the IS server.

One advantage of this new scheme is that by moving the debugging dumps out of the kernel, there is less pressure to keep the code small. Consequently, additional dumps can easily be added and the dumps can be formatted better now to make them more useful.

4.1.3 Observing function keys

In MINIX 2.0.4, function key depresses are detected and directly handled within the TTY task. Since the TTY task can directly access all data structures and functions that are part of the kernel, the event handlers for known function keys can be called directly. This approach is not possible for the user-space TTY driver, because it cannot make function calls and access data structures across address spaces. Therefore, all processes are made responsible for making their own debug dumps, and the IS server was created to handle the debug dumps of kernel data structures.

Since debug dumps are initiated by the user, other processes must be notified about this event. The *observer design pattern* was used for this. Arbitrary processes can register with the TTY driver to receive notifications for a given function key. A new message type, FKEY_CONTROL, was defined for this. Observers are stored in a global array at the TTY driver that allows one observer per function key.

Currently, it is possible to sign up for `‘F1-F12’` and `‘Shift+F1-F12’` notifications. The RTL8139 driver, for example, has registered the `‘Shift+F9’` event. The modifiers `‘Ctrl’` and `‘Alt’` are reserved by the TTY driver as shown in Figure 4.3. Other combinations of modifiers and function keys are still available, though. The combination `‘Alt+Shift+F1-F12’`, for example, is already defined in <*minix/keymap.h*>.

When a function key is pressed, the function func_key() at the TTY driver is called. This function is shown in Listing 4.1. It is first verified that the function key can be observed. Then it is checked whether there is a process that is interested in the event. If an observer is found a FKEY_PRESSED notification is sent. This is done with the nonblocking variant of SEND to prevent the TTY driver from being blocked by a busy observer.

Using a nonblocking variant of SEND implies that some notifications may be lost when the IS server is not ready. This is not a big problem, though,

Function key	Meaning at the TTY
`‘F1-F12’`	reserved for debug dumps by the IS server
`‘Shift + F1-F12’`	free for others processes, like MM and FS
`‘Alt + F1-F12’`	reserved for switching between consoles
`‘Ctrl + F1-F12’`	reserved for special TTY driver functions

Figure 4.3: Function key assignments at the TTY driver. Other combinations of modifiers and function key are still available for future use.

Listing 4.1: The TTY driver function that checks whether there are observers for function key event. If an observer is found a nonblocking notification is sent.

```
PRIVATE int func_key(scode)
int scode;                                          /* key scan code */
{
  message m;                                        /* notification message */
  int index;                                        /* function key number */
  int observer;                                     /* index of observer */
  unsigned fkey;                                    /* key including modifiers */
  int s;

  if (scode & KEY_RELEASE) return(FALSE);           /* ignore key releases */
  fkey = map_key(scode);                            /* include modifiers */

  /* Only F1-F12 and Shift+F1-F12 can be observed. */
  if (F1 <= fkey && fkey <= F12) {                  /* F1-F12 */
      index = fkey - F1;
      observer = fkey_obs[index];
  } else if (SF1 <= fkey && fkey <= SF12) {         /* Shift+F1-F12 */
      index = fkey - SF1;
      observer = sfkey_obs[index];
  } else {                                          /* not observable */
      return(FALSE);
  }

  /* Send notification if an observer is registered. */
  if (observer != NONE) {                           /* observer registered ? */
      m.m_type = FKEY_PRESSED;                      /* assemble notification */
      m.FKEY_NUM = index+1;
      m.FKEY_CODE = fkey;
      if (OK != (s=nb_send(observer, &m)))          /* try to send, don't block */
          printf("F%d key notification to %d failed: %d.\n", index+1, observer, s);
  }
  return(TRUE);
}
```

because the user will be alerted by the TTY driver and can simply retry. If a reliable notification mechanism is needed the code would have been much more complex. This would require a queuing mechanism and the use of timers to periodically retry to deliver the failed notification.

4.1.4 Debug dumps after a panic

A panic is a rare event that is usually caused by a kernel exception. The debug dumps of the process table and other kernel data structures may be helpful to find the problem that caused the panic. As discussed in Subsection 4.1.1, MINIX 2.0.4 handles all debug dumps from within the kernel, which makes the process straightforward.

Listing 4.2: Making debug dumps after a panic must be done with care in order not to be blocked and hang the system. This function waits for keystrokes for printing debugging information and then aborts MINIX.

```
PUBLIC void do_panic_dumps(m)
message *m;                                        /* TTY request message */
{
  int quiet, code;
  (void) scan_keyboard();                          /* ack any old input */
  quiet = scan_keyboard();                         /* quiescent value */

  printf("Hit ESC to reboot, DEL to shutdown, F-keys for debug dumps.\n");
  for (;;) {                                       /* stop on ESC or DEL */

      /* Check for new diagnostics or kernel messages. */
      clk_tickdel(10);                             /* prevent fast looping */
      while (nb_receive(ANY, m) == OK) {
          switch(m->m_type) {                      /* expect output request*/
          case NEW_KMESS:    do_new_kmess(m);    break;
          case DIAGNOSTICS:  do_diagnostics(m);  break;
          }
          clk_tickdel(1);                          /* wait for more */
      }

      /* Check for user actions on the keyboard. */
      code = scan_keyboard();                      /* get last key scan code */
      if (code != quiet) {                         /* a key has been pressed. */
          switch (code) {                          /* possibly abort MINIX */
          case ESC_SCAN:  sys_abort(RBT_REBOOT);  return;
          case DEL_SCAN:  sys_abort(RBT_HALT);    return;
          }
          func_key(code);                          /* check for function key */
          quiet = scan_keyboard();
      }
  }
}
```

In the distributed approach that is described in Subsection 4.1.2, however, several user-space processes must cooperate. This makes debug dumps during a panic more complicated because the kernel must give up control to the user-space TTY driver.

When the kernel aborts due to a panic the function prepare_shutdown() detects this and checks whether the TTY driver is ready to handle debug dumps. This function is discussed in Section 3.4 and is shown in Listing 3.7. If debug dumps are possible control will be passed to the TTY driver by means of PANIC_DUMPS message. When the TTY driver receives the PANIC_DUMPS request it dispatches to the handler function.

The handler function, do_panic_dumps(), is shown in Listing 4.2. It ignores the last keyboard input and then starts its main loop. At the beginning of the loop the nonblocking variant of RECEIVE is used to poll for pending output requests. Because the debug dumps can span multiple printf() calls this is repeatedly done until all messages have been processed. Only kernel messages and diagnostic output are handled; other request are simply ignored. The small delay in the inner loop is meant to give up control so that, for example, the IS server can continue outputting diagnostics.

The next step is to check whether the user gave new keyboard input. The keys `'ESC'` and `'DEL'` cause the TTY driver to abort MINIX. Experience learned that rebooting on a panic is not always helpful—MINIX may directly panic again—so the user can now choose whether MINIX should reboot or halt. The function func_key() is used as before to check if an observer should be notified about for the event. The function keys `'F1'` to `'F12'`, for example, will send a notification and trigger a debug dump at the IS server.

4.2 Generic management of watchdog timers

Timers are important to keep the operating system responsive. Not surprisingly, they are used throughout MINIX' kernel. All timers in MINIX are so called *watchdog timers* that cause delayed execution of a procedure specified by the caller. If a timer expires its associated *watchdog function* is executed. Because the watchdog function is provided by the caller it can do whatever is necessary when the timeout interval expires.

Watchdog timers are used, for example, to schedule wakeup calls, to handle exceptions and to set alarms. User alarm calls are handled in the CLOCK task by setting a timer to run a watchdog function that causes a SIGALRM signal after the specified interval. Synchronous alarms work similarly, but send a SYN_ALARM notification instead of a signal. Many device drivers rely on timers to handle exceptions when the hardware is not responding within a predefined period. If a command times out, the watchdog function sets a flag and wakes the device driver. Finally, timers are also used within the microkernel. For example, the new shutdown sequence that is discussed in Subsection 3.4 uses a timer to forcibly exit processes that fail to shutdown within the given time.

4.2.1 Original implementation

All code relating to watchdog timers is contained in *src/kernel/clock.c*. In MINIX 2.0.0, execution of the watchdog functions is automatically done by the CLOCK task. In MINIX 2.0.4, the process that sets a timer can tell who should be alerted to run the watchdog function if the timer expires. At first sight, the latter setup seems cleaner because the CLOCK task no longer needs to dispatch other processes' watchdog functions, but the approach is not consequently used and thus only obfuscates the code. In fact, only the TTY driver executes its own watchdog functions; the FLOPPY driver uses a mixed approach where some timers are under its own control and some are under the CLOCK's control; the watchdog functions of all other device drivers are dispatched by the CLOCK task. Apart from the process that executes the watchdog function, most dependencies relating to timer management are very similar in both versions of MINIX.

Processes that want to set a timer are required to maintain a timer variable that is passed to one of the timer management functions of the CLOCK task. Timers can be enabled with tmr_settimer() or disabled with tmr_clrtimer(). The CLOCK task is responsible for maintaining lists of active and expired timers. If a timer expires the CLOCK either dispatches the watchdog function itself or alerts the owner of the timer to do so. Watchdog functions of expired timers are dispatched by calling the tmr_exptimer() function. Thus, all processes that use timers strongly depend on the timer management functions in the CLOCK task.

In fact, all device drivers that use the device independent driver code contained in *src/kernel/driver.c* depend on the CLOCK's timer management functions—regardless of whether they actually use timers or not. The reason for this is that the CLOCK's function to dispatch expired timers is automatically called from the main loop of those drivers.

Problems with the original approach

The current design has several shortcomings. First, the dependencies pose a serious problem when the device drivers that use timers are moved out of the kernel. Because user-space device drivers do not share an address space with the CLOCK task, they can no longer directly call the CLOCK's timer management functions and thus cannot set timers as before.

Second, there is the issue who is responsible for dispatching watchdog functions of expired timers. The current mixed approach where the CLOCK task executes the watchdog functions of arbitrary processes clearly obfuscates the code. It would be better if the owner of a timer is responsible for running the associated watchdog function if it expires.

Third, the timer management functions are written towards the CLOCK task and use the CLOCK's private variables, while they, in principle, perform generic operations. Naturally, the CLOCK will always be important to provide a stimulus, namely the current time, for expiring active timers, but this can easily be passed as an argument.

4.2.2 Generic timer management

Because the low-level timer management functions are not accessible from user-space device drivers, they were replaced by a more generic approach. All watchdog timers are now managed in user space—with help of the CLOCK task—by the process that uses them.

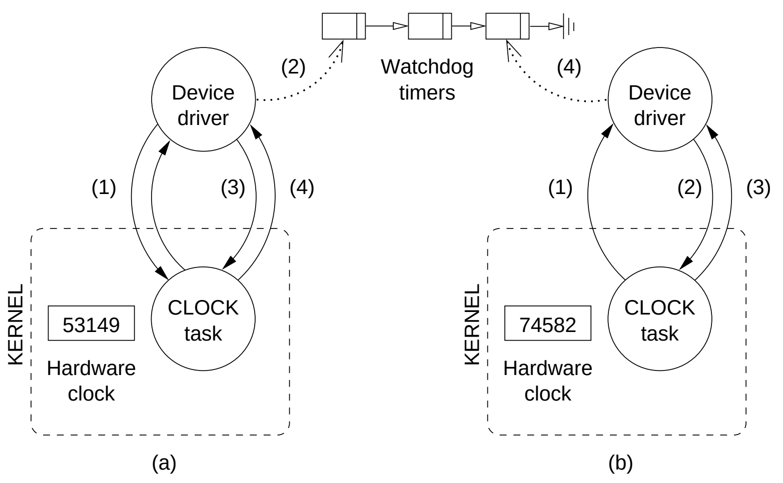

Figure 4.4: User-space watchdog timer management. The figure shows the steps that are taken by a device driver to (a) set a new watchdog timer and (b) expire a watchdog timer. The precise steps are explained in Subsection 4.2.2.

The timer management functionality works roughly as follows. Figure 4.4(a) illustrates the steps to set a new timer. In step 1, the device driver requests the time from the CLOCK task with a CLK_GETUPTM system call. It uses the uptime to calculate the expiration time and then adds the new watchdog time to its local queue of watchdog timers in step 2. Finally, the driver schedules a synchronous alarm for the new timer with a CLK_SYNALRM system call in steps 3 and 4.

Figure 4.4(b) shows what happens if a watchdog timer expires. When the synchronous alarm goes off, the CLOCK task notifies the device driver with a SYN_ALARM message in step 1. The notification is directly delivered when the driver is blocked waiting for a message. If the driver is busy, the notification is delivered as soon as it does a receive() call. In steps 2 and 3, the driver calls back with a CLK_GETUPTM system call to retrieve the current time. The time is required to determine which watchdog timers have expired and is passed as an argument to library call that checks the local queue in step 4. The watchdog function of expired watchdog timers are automatically run. The latter is not shown in the figure.

Problems encountered

Several problems were encountered during implementation of the user-space timer management functionality. The synchronous alarm functionality in MINIX 2.0.4 has several shortcomings that affect its reliability when used by multiple processes at a time. Because the SYN_AL task uses the send() function to notify processes about an expired alarm, it may be blocked for an indefinite period of time. Section 3.3 discusses a solution for this problem.

Listing 4.3: Function prototypes of the timer management library. There are two functions to add or remove a watchdog timer to or from the queue of timers and one to check for expired timers and run their watchdog functions. Furthermore, two macros exist to initialize a timer and to set an optional argument.

```
#define tmr_inittimer (tp) (void)((tp)->tmr_exp_time=TMR_NEVER)
#define tmr_arg(tp) (&(tp)->tmr_arg)

_PROTOTYPE( void tmrs_settimer,    (timer_t **tmrs, timer_t *tp,
                              clock_t exp_time, tmr_func_t watchdog));
_PROTOTYPE( void tmrs_clrtimer,    (timer_t **tmrs, timer_t *tp));
_PROTOTYPE( void tmrs_exptimers,  (timer_t **tmrs, clock_t now));
```

Another problem is that execution of a watchdog function by the CLOCK task is similar to running a signal handler in that the process' thread of execution is interrupted but not altered. The watchdog function causes some side-effects, such as setting a timeout flag, that allow the process to notice the timeout, but the process resumes execution where it left off when the timer expired. In contrast, receiving a SYN_ALARM message requires an explicit receive() call and inspection of the incoming message, which is directly reflected in the code because the flow of control changes.

Yet another issue that complicated replacing the watchdog timers is that some device drivers require multiple timers that can be simultaneously active. The reason for this is that some drivers manage more than one device. The FLOPPY driver, for example, uses two timers for each disk drive it handles: one for error handling and one to stop the motor a few seconds after the last access. The TTY uses one timer per terminal, as well as one to control beeping. This behavior cannot be directly simulated by synchronous alarms, because each process can only have a single outstanding synchronous alarm.

Implementation of a new timers library

The problems were solved by extracting the timer functionality of MINIX 2.0.4, and transforming it into generic code that does not depend on the CLOCK task. Instead of using the CLOCK's private variables to manage the timer lists, the current time as well as a pointer to the queue of timers to be managed must now be provided by the caller. The new, generic timer management functions are available in a new system library.

The library comprises three functions for maintaining queues of watchdog timers and two macros that operate on a timer structure. The function prototypes and macro definitions are shown in Listing 4.3. The function prototypes and (type) definitions can be found in *<timers.h>*. The implementation is contained in *src/lib/timers/*.

The macros tmr_inittimer() and tmr_arg() can be used to initialize a timer structure or to set an optional argument that is passed to the watchdog

Listing 4.4: Generic timer management in the FLOPPY driver. Two helper functions were defined to set a new watchdog timer and to check the queue for expired timers if a synchronous alarm message arrives.

```
PRIVATE timer_t f_timers;                                   /* queue of timers */
PRIVATE clock_t f_next_timeout;                             /* next sync alarm */

PRIVATE void f_set_timer(tp, delta, watchdog)               /* set new watchdog */
timer_t *tp;                                                /* pointer to timer */
clock_t delta;                                              /* in how many ticks */
tmr_func_t watchdog;                                        /* watchdog function */
{
  clock_t now;                                              /* current time */
  int s;

  /* Get the current time to calculate the expiration time. */
  if ((s=clk_getuptm(&now)) != OK)
      server_panic("FLOPPY","Couldn't get uptime.", s);

  /* Add the timer to the local queue of timers. */
  tmrs_settimer(&f_timers, tp, now + delta, watchdog);

  /* Reschedule an alarm call, if the front of the active
   * timers list has changed, i.e., updated or renewed.
   */
  if (f_timers.tmrs_active->tmr_exp_time != f_next_timeout) {
      f_next_timeout = f_timers.tmrs_active->tmr_exp_time;
      if ((s=clk_syncalrm(SELF, f_next_timeout, 1)) != OK)
          server_panic("FLOPPY","Couldn't set alarm.", s);
  }
}

PRIVATE void f_tmrs_expire(struct driver *dp)               /* expire watchdog */
{
  clock_t now;                                              /* current time */
  timer_t *tp;                                              /* timer pointer */
  int s;

  /* Get the current time to compare the watchdog timers against. */
  if ((s=clk_getuptm(&now)) != OK)
      server_panic("FLOPPY","Couldn't get uptime.", s);

  /* Run watchdog functions of expired timers. Possibly reschedule an alarm. */
  (void) tmrs_exptimers(&f_timers, now);                    /* expire timers */
  if (f_timers.tmrs_active == NULL) {                       /* no more timers */
      f_next_timeout = TMR_NEVER;
  } else {                                                  /* reschedule alarm */
      f_next_timeout = f_timers.tmrs_active->tmr_exp_time;
      if ((s=clk_syncalrm(SELF, f_next_timeout, 1)) != OK)
          server_panic("FLOPPY","Couldn't set alarm.", s);
  }
}
```

function. The first argument to all functions is a pointer to a locally declared queue of timers. The functions tmrs_settimer() and tmrs_clrtimer() can be used to add a new timer to and to remove a timer from the timers queue, respectively. The function tmrs_exptimers() takes the current time to check the timers queue for expired timers, and runs their watchdog functions.

User-space device drivers can use multiple watchdog timers by keeping a local queue, and requesting a single synchronous alarm for the first timer to expire. This approach is illustrated in Figure 4.4. When a SYN_ALARM message is received the current system time is looked up to check and expire the local timers with the function tmrs_exptimers(). Unless all timers are expired, a new alarm must be scheduled for the next active timer.

The user-space FLOPPY driver, for example, uses the new timer management library. Two functions, f_set_timer() and f_tmrs_expire(), were locally declared to conveniently set new timers and to handle synchronous alarms. These are shown in Listing 4.4. The functions operate on the global variables f_timers and f_next_timeout that keep track of the timers queue and the next timer to expire, respectively. Both functions have a similar structure. They first retrieve the current time from the CLOCK task. Then the queue of timers is updated by adding a new timer or running the watchdog function of expired timers, respectively. Finally, a new synchronous alarm for the next timer to expire is scheduled. Subsection 4.3.2 describes how this is used to detect different kinds of timeouts.

Using the new timer management functions in the CLOCK task is particularly convenient because the current time is directly accessible. Instead of requesting synchronous alarms, the current time can be used directly to check for expired timers at each clock tick. A substantial part of the CLOCK task was refurbished by using the new timer management functionality. All old timer management functions were removed, the alarm functionality was rewritten to use separate timer variables for different types of alarms, and new functions were added to set or reset watchdog timers from within the microkernel. The latter means that the CLOCK task now is responsible for all watchdog functions used in the microkernel.

4.3 Dealing with unresponsive hardware

Device drivers have to deal with all kinds of peculiarities of hardware devices. An important issue is how to deal with *timeouts*. A timeout occurs when a device fails complete a requested operation within a predefined period of time. Timeouts can, for example, be caused by a hardware failure. The exact underlying reason is not of interest for this discussion, though.

Timeouts must be detected to prevent being blocked by unresponsive hardware. When a timeout is detected a driver can take various actions. It may, for example, give up or reset the hardware and retry the operation.

4.3.1 Exception handling in MINIX 2.0.4

This subsection discusses two different approaches to detect timeouts that are used by the device drivers in MINIX 2.0.4, and outlines the shortcomings.

Using the 8253A timer The first approach is to use the microseconds counter of the 8253A timer to keep track of the time elapsed since the operation started. This functionality is contained in *src/kernel/clock.c*. The counter can be initialized with a call to micro_start(). The number of microseconds that have passed can be obtained with micro_elapsed(). The latter function must be polled rapidly to get accurate timer values, because the clock counter counts down and resets when it reaches zero.

Device drivers can use this functionality to detect a timeout by doing the requested operation within a loop that breaks when either the request succeeds or the number of microseconds elapsed exceeds the timeout interval. The code within the loop can set a timeout flag or can use different return values to inform the driver about the result.

Using watchdog timers The second approach is to use watchdog timers to cause delayed execution of a function provided by the caller. Running a watchdog timer in MINIX 2.0.4 is similar to running a signal handler in that the process' thread of execution is interrupted but not altered. Section 4.2 discusses watchdog timers and timer management in more detail. The focus here is how they are used to deal with unresponsive hardware.

Device drivers such as the FLOPPY and AT_WINI frequently issue a command to the controller and then expect a hardware interrupt to occur. The driver awaits the interrupt with a blocking call to receive() to get a message with source HARDWARE. The interrupt handler provides such a message if the expected interrupt occurs by calling interrupt(). This function sends a HARD_INT message with source HARDWARE to the device driver.

Before issuing the command, however, the device driver sets a watchdog timer to prevent being blocked by unresponsive hardware. Because the driver is blocked by the receive() call, the CLOCK task is made responsible for running the associated watchdog function when the timer expires. If a timeout occurs the watchdog function sets a timeout flag and then fakes a hardware interrupt by calling interrupt() to wake up the device driver.

The device driver thus will always receive a HARD_INT message that unblocks it. Once it is awake, it uses the timeout flag to tell a successfully performed operation and a timeout apart, and takes further action.

Problems with the original implementation

When device drivers are moved out of the kernel they can no longer detect timeouts as is done in MINIX 2.0.4, since the microseconds counter of the

8253A timer cannot be rapidly polled by user-space process without privileges to perform device I/O. A possible solution is to make a SYS_DEVIO system call and have the kernel read the counter, but this requires two extra context switches per loop iteration and thus gives very inaccurate readings.

Watchdog timers are still available for user-space processes, but require a different approach than the watchdog timers of MINIX 2.0.4. It is, for example, no longer possible to have the CLOCK task run the associated watchdog function if the timer expires, because function calls across address spaces are not possible. The new approach places the responsibility for managing a queue of watchdog timers and running the watchdog function of expired timers at the process that needs them. The precise details are described in Section 4.2.

As an aside, the use of watchdog timers in MINIX 2.0.4 is not very elegant from a conceptual point of view. Because the watchdog function forges a hardware interrupt, the information that is conveyed becomes meaningless. The HARD_INT message is merely used to wake up the device driver, and the status flag must always be inspected to see what happened. It would be better to send a proper timeout notification so that the driver can decide what to on basis of the information is was given.

4.3.2 New approaches to detect timeouts

This subsection presents two new approaches to detect timeouts that can be applied by user-space device drivers, and gives an example of an advanced timer management scheme that is used by the FLOPPY driver.

Using the new timeout flag alarm To overcome the problem that microsecond counter cannot easily be used by user-space processes, a new type of alarm, CLK_FLAGALRM, was 'invented' to set a timeout flag in the caller's address space. The alarm can be activeted with library function clk_flagalrm() by providing the timeout interval and pointer to a timeout flag. The flag is set to 1 if the alarm goes off.[1]

This approach alarm is convenient to use and the resulting code is easy to understand. Listing 4.5, for example, shows how the new type of alarm is used in the AT_WINI driver.

Using the synchronous alarm Because the CLOCK task cannot run the watchdog functions of user-space device drivers, the driver now requests a synchronous alarm before calling receive() to prevent being blocked by unresponsive hardware. This is discussed in Section 4.2. If no hardware

[1]A similar result could be obtained by requesting an ordinary alarm() call and setting a timeout flag to 1 in the alarm signal handler, but unfortunately MINIX does not allow system services to be signaled. This is part of future work.

Listing 4.5: Detecting a timeout with the new CLK_FLAGALRM alarm. This code is used in the AT_WINI device driver to wait until the controller is in the required state. The timeout alarm is not reset upon success because a static flag is used so that leftover timeout cannot do any harm.

```
PRIVATE int w_waitfor(mask, value)
int mask;                                         /* status field mask */
int value;                                        /* the required status */
{
  static int timeout = 0;                         /* timeout flag alarm */
  clk_flagalrm ((TIMEOUT_TICKS, &timeout);

  do {
      w_status = inb(w_wn->base + REG_STATUS);    /* check controller status */
      if ((w_status & mask) == value)
          return(OK);                             /* stop if status is ok */
  } while (! timeout);                            /* or if alarm expired */

  w_need_reset();                                 /* controller gone deaf */
  return(E_TIMEOUT);
}
```

interrupt occurs within the timeout interval, that is, if no HARD_INT notification arrives, the synchronous alarm goes off and a SYN_ALARM notification is received instead. In both cases, the device driver is unblocked and can resume execution.

This allows user-space device drivers to detect timeouts and results in cleaner code compared to MINIX 2.0.4, because it is no longer needed to fake hardware interrupts and decide what happened on basis of some status flag. The notification message now simply tells what has happened. Listing 4.6 shows how this is applied in the FLOPPY driver.

This approach was made possible by the new notify() construct that is used to handle asynchronous events such as interrupts and alarms. It works because all notification messages have the same source, namely HARDWARE. The notify() construct is discussed in Section 3.3.

Combining different alarms The two approaches for detecting timeouts can safely be combined, because the kernel now uses separate timers for each type of alarm.[2] This, for example, is required by the RTL8139 device driver that simultaneously uses a synchronous alarm to periodically check for missed interrupts and a flag alarm to read port values within a short timeout interval.

[2]MINIX 2.0.4 uses a single timer per process so that only one alarm can be active at a time. Setting a new alarm always cancels a previously set alarm, regardless of its type.

Listing 4.6: Handling timeouts with a synchronous alarm. This code is used in the FLOPPY device driver to wait for a task completion interrupt that sets the status to 'not busy'. The variable f_busy is set to BSY_WAKEN if a timeout occurs.

```
PRIVATE int f_intr_wait ()
{
  message m;                              /* storage for message */

  do {
      receive(HARDWARE, &m);              /* receive a message */
      switch (m.m_type) {
          case SYN_ALARM:                 /* synchronous alarm */
              f_expire_tmrs ();           /* check for timeout */
              break;
          default:                        /* expect a interrupt */
              f_check_status ();          /* get f_busy status */
      }
  } while (f_busy == BSY_IO);

  if (f_busy == BSY_WAKEN) {              /* set on a timeout */
      need_reset = TRUE;                  /* reset the floppy */
      return(E_TIMEOUT);
  }
  return(OK);
}
```

Advanced timer management

With the basic approach that allows user-space device drivers to detect timeouts in place, a more advanced scheme that is discussed in Section 4.2 can be illustrated. In this scheme, a user-space device driver locally manages multiple watchdog timers that can be active at the same time. This may be required if the driver services multiple devices.

The device driver no longer relies on the CLOCK task to keep track of watchdog timers, but maintains a local queue of timers. When a time critical operation must be performed, the driver adds a watchdog timer to the local queue and sets a synchronous alarm. Then it awaits a message with source HARDWARE as before. When the alarm went off it checks the local queue to see if one or more watchdog timers expired.

This approach is used in the function f_intr_wait() that is shown in Listing 4.6. The function f_tmrs_expire() is called on a timeout to check the queue of watchdog timers. The code is rather subtle, because the arrival of a SYN_ALARM notification does not necessarily mean that a timeout for f_intr_wait() occurred. The notification may as well expire other timers that happen to have a shorter timeout interval. The function f_tmrs_expire() carefully checks if there are remaining watchdog timers and reinstalls a synchronous alarm if needed. Its code is shown in Listing 4.4 in Section 4.2.

Chapter 5

Kernel reductions

This chapter describes how the PRINTER, MEMORY, AT_WINI, FLOPPY and TTY tasks were transformed into independent, user-space device drivers. This reduction of the kernel can be regarded as the most important result of this master's project. The transformation was made possible by general kernel improvements and new applications that are described in the previous two chapters.

The general approach that was followed when removing a kernel task was to remove all dependencies while the task still was part of the kernel. This allowed to replace all kernel dependencies with similar alternatives one by one. Most dependencies were replaced by a system call to let the kernel perform some task on behalf of the user-space driver.

Once all kernel dependencies were removed the driver was copied to a separately directory in *src/drivers/* to make it function as an independent, user-space device driver. The kernel modifications that were needed to include the new driver program in MINIX' system image are described in Appendix B.

The following sections describe how each driver was transformed into a user-space driver and what problems were encountered. Furthermore, some general improvements and modifications to MINIX that were initiated by the transformation of specific device drivers are discussed.

5.1 A user-space PRINTER driver

The PRINTER driver in MINIX supports a single Centronics compatible printer. Its transformation into an user-space device driver served as a test case during the problem analysis. Although the PRINTER driver was successfully removed from the kernel, several modifications to the original PRINTER task were needed before the actual work could start. Furthermore, an unexpected problem with MINIX' low-level process scheduling code was encountered.

Symbol		Purpose
_data_base	V	Base of kernel's data segment to get physical addresses
_disable_irq	F	Disable interrupts on restart when printer is hung
_enable_irq	F	Enable interrupts during initialization or after restart
_inb	F	Input a byte from the printer's status register
_interrupt	F	Interrupt handler notifies the driver about interrupt
_micro_delay	F	Delay to meet Cetronics interface timing demands
_numap	F	Map a virtual address to physical address for copying
_outb	F	Output a byte to the printer's data or control register
_phys_copy	F	Copy data to be printed using physical addressing
_pproc_addr	V	Pointers for fast access into the process table
_put_irq_handler	F	Driver sets interrupt handler during initialization

Figure 5.1: Dependencies of the PRINTER task in MINIX 2.0.4. Symbols are either a function (F) or variable (V). See Figure 2.3 for the full dependencies matrix.

5.1.1 Modification of the original PRINTER task

The setup of the PRINTER task in MINIX 2.0.4 is to a large extent dependent on its interrupt handler, pr_handler(). Basically all accounting and printing, that is, the actual device I/O, is done in a loop within the interrupt handler. Because interrupt handlers are to stay within the kernel, it was impossible to directly move the driver to user-space. Hence, the PRINTER task was first remodelled to only send an interrupt notification in the interrupt handler and to do the actual work in a separate function.

A second problem with the original PRINTER task is that it is badly designed and depends on the CLOCK task to properly function. On each clock tick, the PRINTER driver requires a restart by the CLOCK task, with pr_restart(), to prevent it from being idle when there is work to do. The redesigned setup of the interrupt handler obviated the need for this.

A third problem is that the original PRINTER task immediately returns an error after issueing a print request for the first time. Reissueing the request usually works. The reason is that the printer hardware needs some time to come 'online', that is, to move the print head to the starting position, while the PRINTER task does not wait for this. This problem was fixed by waiting for this event before actually trying to write to the printer.

5.1.2 Setup of a user-space PRINTER driver

Once the interrupt handler of the original PRINTER task was modified and the driver behaved correctly on startup, it was moved from kernel-space to user-space. The approach followed was to remove all kernel dependencies while the PRINTER task was still part of the kernel. All dependencies are shown in shown in Figure 5.1.

The dependencies were replaced by system calls that let the kernel perform some task on behalf of the user-space driver. This process was

Listing 5.1: MINIX' low-level scheduling code was completely rewritten to support generic multilevel scheduling. The ready() function updates the scheduling queues and implicitly picks a new process when it has a higher priority to the currently active process.

```
PRIVATE void ready(rp)
register struct proc *rp;                          /* process is now runnable */
{
  int q = rp->p_priority;                          /* scheduling queue to use */

  /* User processes may be I/O-bound and are added in front. */
  if (isuserp(rp)) {                               /* user processes */
        if (rdy_head[q] == NIL_PROC)               /* empty queue */
            rdy_tail[q] = rp;                      /* update rear of queue */
        rp->p_nextready = rdy_head[q];             /* rp goes before others */
        rdy_head[q] = rp;                          /* update front of queue */
  }
  /* All other processes are added to the end of the queue. */
  else {                                           /* system services */
      if (rdy_head[q] != NIL_PROC)                 /* nonempty queue */
          rdy_tail[q]->p_nextready = rp;           /* rp goes after others */
      else                                         /* empty queue */
          rdy_head[q] = rp;                        /* update front of queue */
      rdy_tail[q] = rp;                            /* update rear of queue */
      rp->p_nextready = NIL_PROC;                  /* rp is last entry */
  }

  /* Run 'rp' next if it has a higher priority than 'proc_ptr'. */
  if (rp->p_priority < proc_ptr->p_priority) proc_ptr = rp;
}
```

straightforward. Copying of data to the printer task is now done with the SYS_VIRCOPY system call. Interrupts are handled by setting a policy for the kernel's generic interrupt handler with the SYS_IRQCTRL system call. Finally, device I/O is handled with the SYS_DEVIO and SYS_VDEVIO calls. These calls are explained in Section 3.1. An overview of the precise system call paramaters is given in Appendix C.

5.1.3 Generic multilevel scheduling

A time-consuming problem was encountered once the printer task was moved out of the kernel and ran as a server program. The printing functionality itself worked, but long periods of inactivity were experienced. Although the different process scheduling priorities for servers and tasks were immediately under suspicion, the solution was not quickly found.

A bug in *src/kernel/proc.c* turned out to be responsible. In MINIX 2.0.4, a side-effect of ready() is that kernel tasks are directly scheduled because they run at the highest priority. User-space processes, however, are merely

Listing 5.2: The function pick_proc() is no longer repeatedly checks the queues of different process types, but simply checks all scheduling queues from high to low priority. The IDLE process resides in the queue with the lowest priority.

```
PRIVATE void pick_proc()
{
  register struct proc *rp;                            /* process to run */
  int q;                                               /* iterate over queues */

  /* Select the highest priority , runnable process by setting ' proc_ptr '. */
  for (q=0; q < NR_SCHED_QUEUES; q++) {                /* check all queues */
      if ((rp=rdy_head[q]) != NIL_PROC) {              /* find ready process */
          proc_ptr = rp;                               /* run process 'rp' next */
          if (isuserp(rp) || isidlep(rp)) bill_ptr = rp; /* possible bill 'rp' */
          return;
      }
  }
}
```

added to the ready queue and not directly scheduled. The interrupt() function of MINIX 2.0.4 relies on the mentioned side-effect and does not call pick_proc() to schedule a new process. Therefore, the user-space PRINTER driver was not scheduled when it became ready due to an hardware interrupt. Instead, it would remain inactive until another event caused pick_proc() to be called. Calling pick_proc() after announcing a process ready solved the problem. This way user-space device drivers also get a fair chance to be scheduled.

While the low-level scheduling code of MINIX 2.0.4 was thoroughly studied, several other shortcomings were found. The most serious problems are that it directly couples process priorities to a process types and verbosely repeats the same checks for different types of processes. The scheduling code was completely rewritten to fix these problems. In the new setup process types and priorities are fully decoupled and MINIX' scheduling now can be characterized as generic multilevel scheduling. Furthermore, the IDLE process is no longer treated as an exception, but simply resides in the queue with the lowest priority. The new ready() and pick_proc() functions are shown in Listing 5.1 and 5.2, respectively.

The new ready() function immediately schedules a process when it is ready and has a higher priority than the currently running process. Several low-level process management functions, including mini_send() and mini_rec() rely on this side-effect. Although it may be cleaner to call pick_proc() in all cases, saving a function call is beneficial for MINIX' IPC performance.

All process types and priorities are defined in *src/kernel/proc.h*. The number of scheduling queues, for example, can be changed by updating NR_SCHED_QUEUES. Process types and priorities of individual system services can be set in the image table in *src/kernel/table.c*. This allows fine-grained control over which process has precedence over another.

5.2 A user-space MEMORY driver

The MEMORY driver is the second driver that was transformed in a user-space device driver. This driver is responsible for handling the null device (*/dev/null*), physical memory (*/dev/mem*), kernel memory (*/dev/kmem*), and the RAM disk (*/dev/ram*). The null device acts as a data sink and simply discards all data written to it. Physical memory and kernel virtual memory allow operations on the entire memory and kernel memory, respectively. Two uses of the latter are discussed below. The RAM disk that is required by the FS probably is the most important functionality of the MEMORY driver.

5.2.1 Setup of a user-space MEMORY driver

The kernel dependencies of the MEMORY task are given in Figure 5.2. Most dependencies directly relate to concern the driver's core functionality, that is, copying memory from one place to another. Like all other tasks in MINIX 2.0.4 the MEMORY task uses physical addressing. This was changed into virtual addressing so that all copying could be done with help of the SYS_VIRCOPY system call that is discussed in Section 3.1.2.

MINIX' memory management is done by the MM server, which requests the array of free memory areas when MINIX boots. The MEMORY driver, however, needs to allocate a RAM disk before the MM grabs all free memory. In MINIX 2.0.4 this is done by directly updating the array with free memory areas. The user-space MEMORY driver temporary uses the SYS_KMALLOC system call to do the same. The definition of a better resource management framework is part of future work.

Since the MEMORY driver uses the device independent device driver interface it automatically inherits the dependencies contained in the device

Symbol		**Purpose**
_data_base	V	Base of kernel's data segment to get physical addresses
_enable_iop	F	Enable CPU's IOPL bits to allow user-space device I/O
_mem	V	Array with free memory areas to get RAM disk memory
_numap	F	Map a virtual address to physical address for copying
_panic	F	Panic if there is not enough memory for the RAM disk
_phys_copy	F	Copy from/ to RAM disk using physical addressing
_pproc_addr	V	Pointers for fast access into the process table
_proc	V	Process table has memory ranges to get physical address
_proc_ptr	V	Use pointer to active process to get process number
_tasktab	V	Used by the device independent code to get name
_tmr_exptimers	F	Automatically called by device independent code
_vir_copy	F	Use virtual copy for psinfo structure of `'ps'` utility

Figure 5.2: Dependencies of the MEMORY task in MINIX 2.0.4. Symbols are either a function (F) or variable (V). See Figure 2.3 for the full dependencies matrix.

independent code. Since the device independent code is shared with other device drivers, such as FLOPPY and AT_WINI, a revised version without kernel dependencies was placed in a separate directory, *src/drivers/libdriver*. Because the timer functionality is not needed by the MEMORY driver it was made optional. This is further discussed in Subsection 5.3.2.

All in all, the transformation MEMORY task into a user-space driver was relatively easy because the MEMORY driver does not require any interrupt handling. The _enable_iop dependency that enables the CPU's IOPL bits to allow user-space device I/O caused most problems and is discussed below.

Limitations of the user-space MEMORY driver

The MEMORY task of MINIX 2.0.4 has some features that are only used by the MINIX VMD distribution. It can execute BIOS calls on behalf of user processes and allows them to access far memory by installing a descriptor for it in their local descriptor table (LDT). These features are realized by the I/O control requests MIOCINT86 and MIOCSLDT86, respectively.

The user-space MEMORY no longer supports the above I/O controls because they are not strictly needed and would give the MEMORY driver to much power. Moreover, they actually cannot be implemented in user-space because the features require access to privileged functions and data structures in the kernel.

5.2.2 Reading the real time clock

When MINIX boots, it dynamically sets the system date and time with the `'readclock'` utility that is defined in *src/command/ibm/readclock.c*. This program retrieves current real time clock (RTC) stored in the CMOS RAM by reading from the BIOS and makes a stime() system call to set the system time. This call is handled by the FS, which forwards the system time to kernel's CLOCK task.

User programs normally cannot perform device I/O, but `'readclock'` uses a feature[1] of MINIX 2.0.4 to get additional privileges. When a process opens the */dev/mem* device its CPU's I/O protection level (IOPL) bits are enabled by the MEMORY driver. This side-effect allows user programs that run as superuser to perform device I/O. The `'readclock'` program thus can directly access the RTC registers to get the system time.

The user-space MEMORY driver no longer enables the CPU's IOPL bits which makes that the `'readclock'` program does no longer work. The MMU's protection mechanisms turned out to work fine and `'readclock'` produced a 'memory fault' as soon as MINIX was booted. The SYS_DEVIO

[1]All processes that open */dev/mem* and */dev/kmem* get I/O privileges as a side-effect. This feature is intended for systems with memory-mapped I/O, but can be considered bad design because it gives too much privileges to user processes.

system also cannot be used because user program are not allowed make kernel calls. Therefore, a new FS call CMOSTIME was created to let the FS perform the work. This call is found in *src/servers/fs/cmostime.c*.

Another change is that the FS no longer forwards the system time to the kernel, but stores the boot time in an internal variable, boottime. The kernel does not need to know about the real time clock, but only keeps track of the number of ticks since boot time. When a user program retrieves the current time with a time() system call the FS requests the number of clock ticks from the kernel and adds this to its local boot time.

5.2.3 Problems with the 'ps' utility

The transformation of the MEMORY driver into a user-space program also affected the `'ps'` utility that prints process table information. Since MINIX' process model is distributed over the kernel, MM and FS the `'ps'` utility must gather process information from several places in main memory. It does so by opening */dev/mem* and reading from the process table addresses provided by the psinfo structure of the MEMORY driver.

A minor change was required to make this work for the user-space MEMORY driver. When MINIX boots, the MM and FS report their process table addresses as in MINIX 2.0.4. The SYS_GETINFO system call is used, however, to obtain the address of the kernel's process table.

As an aside, the process table structures of the MM and FS and kernel were subject to frequent changes that affected the offset of process table fields used by the `'ps'` utility. This sometimes resulted in unexpected and garbled output. Fortunately, the problem could easily be solved with a simple recompilation of the `'ps'` utility.

5.3 User-space AT_WINI and FLOPPY drivers

The third and fourth task that were removed from MINIX' kernel are the AT_WINI and the FLOPPY device driver, respectively. They are discussed together in this section because their design is very similar. The AT_WINI driver is the default disk driver in MINIX. It can handle two AT Winchester hard disks and provides ATAPI CD-ROM support. The FLOPPY driver supports up to two floppy disk drives.

The approach to transform the AT_WINI and FLOPPY tasks into user-space drivers was very similar to what was done for the PRINTER and MEMORY driver. As before, most dependencies were removed while the tasks were still part of the kernel.Therefore, this will not not be discussed again. A complete overview of the dependencies is given in Figure 5.3. The following subsections explain some other issues that were encountered.

Symbol		**Purpose**
_data_base	V	Base of kernel's data segment to get physical addresses
_enable_irq	F	Enable interrupts during initialization or after restart
_inb	F	Input a byte from the controller's data or status register
_interrupt	F	Interrupt handler notifies the driver about interrupt
_micro_delay (*)	F	Delay a few microseconds before retrying an operation
_micro_elapsed	F	Check the microseconds counter of the 8253A timer
_micro_start	F	Initialize the microseconds counter to zero
_numap	F	Map a virtual address to physical address for copying
_outb	F	Output a byte to the controller's data or control register
_panic	F	Panic when controller (re)initialization fails
_phys_copy	F	Copy from or to the BIOS or user process
_phys_insw (*)	F	Input an array of words into a buffer from the controller
_phys_outsw (*)	F	Output an array of words into a buffer to the controller
_pproc_addr	V	Pointers for fast access into the process table
_put_irq_handler	F	Driver sets interrupt handler during initialization
_tasktab	V	Used by the device independent code to get name
_tmr_exptimers	F	Automatically called by device independent code
_vir_copy	F	Use virtual copy for psinfo structure of `'ps'` utility

Figure 5.3: Dependencies of the AT_WINI and FLOPPY tasks in MINIX 2.0.4. An asterix (*) indicates that a dependency only exist for AT_WINI. Symbols are either a function (F) or variable (V). See Figure 2.3 for the full dependencies matrix.

5.3.1 Detecting controller timeouts

The AT_WINI and FLOPPY tasks have several dependencies that are used to detect controller timeouts. The approach that is used in MINIX 2.0.4 to deal with unresponsive hardware, however, cannot be applied by user-space drivers. Therefore two new mechanisms were thought of before the AT_WINI and FLOPPY tasks could be moved to user space.

The new timeout mechanisms for the user-space AT_WINI and FLOPPY drivers are discussed in detail in Section 4.2 and 4.3. Key examples are given in Subsection 4.3.2. Listing 4.5, for example, illustrates how the AT_WINI driver uses the new timeout flag alarm and Listing 4.6 shows how the FLOPPY driver detects timeouts with help of a synchronous alarm.

Especially the FLOPPY driver is demanding in this respect because it supports multiple devices and maintains multiple watchdog timers per device; one to stop the motor and one for error handling. Subsection 4.2.2 discusses how multiple watchdog timers can be maintained in user-space and Listing 4.4 shows how this is done by the FLOPPY driver.

5.3.2 Changes to the device independent code

The use of device independent device driver code brings several benefits. It, for example, improves the structure of device drivers and makes sure that they all adhere to the same interface—as expected by the FS. Another

Listing 5.3: The driver structure of the device independent code must be initialized by device driver specific handler functions.

```
struct driver {
  _PROTOTYPE( char *(*dr_name),      (void ) );
  _PROTOTYPE( int (*dr_open),        ( struct driver *dp, message *m_ptr) );
  _PROTOTYPE( int (*dr_close),       ( struct driver *dp, message *m_ptr) );
  _PROTOTYPE( int (*dr_ioctl),       ( struct driver *dp, message *m_ptr) );
  _PROTOTYPE( struct device *(*dr_prepare), (int device ) );
  _PROTOTYPE( int (*dr_transfer),    ( int proc_nr, int opcode,
                                off_t position, iovec_t *iov, unsigned nr_req ) );
  _PROTOTYPE( void (*dr_cleanup),    (void ) );
  _PROTOTYPE( void (*dr_geometry),   ( struct partition *entry ) );
  _PROTOTYPE( void (*dr_stop),       ( struct driver *dp ) );
  _PROTOTYPE( void (*dr_alarm),      ( struct driver *dp ) );
};
```

advantage in MINIX 2.0.4 is that the code is not linked separately with each task. The kernel images contains a single copy of the executable code, which is shared by multiple drivers. Unfortunately, this property is lost when device drivers are moved to user space.

MINIX uses device-independent code for many device drivers, including the MEMORY, AT_WINI and FLOPPY drivers. The drivers must first initialize a driver structure with pointers to specialized handler functions and then pass this structure to the function driver_task(). This function starts the driver's main loop that repeatedly waits for a requests. When a known request is received it automatically dispatches one of the specialized handler functions provided by the driver. Otherwise an error is returned.

To make the device independent code work for user-space drivers all kernel dependencies had to be removed, but some other changes were needed as well. These are discussed below. The new code is contained in the files *driver.h* and *driver.c* in *src/drivers/libdriver/*.

The driver structure had to be updated with two new request types. A hook for HARD_STOP notifications was added to the driver structure to support the shutdown sequence that is discussed in Section 3.4. Furthermore, the new approach for detecting timeouts required a hook for SYN_ALARM notifications. As discussed in Section 4.2 user-space drivers can still use watchdog timers, but this requires scheduling a synchronous alarm call. The new driver structure is shown in Listing 5.3.

In addition two new functions, nop_stop() and nop_alarm(), were defined to provide a default implementation for device drivers that do not require specific actions for HARD_STOP and SYN_ALARM notifications. The function nop_alarm() simply ignores leftover alarm notifications and directly returns. This, for example, is used in the MEMORY driver. The function nop_stop() does not run any cleanup code, but directly exits the driver.

Listing 5.4: The FS function map_driver() installs a new device driver mapping in the dmap table. Provided that correct arguments are given, this only works if the entry is mutable and the currently installed driver is not busy.

```
PUBLIC int map_driver(major, proc_nr, dev_style)
int major;                                         /* major device number */
int proc_nr;                                       /* driver's process nr */
int dev_style;                                     /* style of the device */
{
  struct dmap *dp;                                 /* pointer into table */

  /* Get pointer to device entry in the dmap table. */
  if (major >= max_major) return(ENODEV);
  dp = &dmap[major];                               /* set table entry */

  /* See if updating the entry is allowed. Immutable entries can never be updated. */
  if (! (dp->dmap_flags & DMAP_MUTABLE)) return(EPERM);
  if (dp->dmap_flags & DMAP_BUSY) return(EBUSY);   /* driver is busy */

  /* Check if process number of new driver is valid. */
  if (! isokprocnr(proc_nr)) return(EINVAL);

  /* Almost done. Try to update the entry. */
  switch (dev_style) {
  case STYLE_DEV:      dp->dmap_opcl = gen_opcl;      break;
  case STYLE_TTY:      dp->dmap_opcl = tty_opcl;      break;
  case STYLE_CLONE:    dp->dmap_opcl = clone_opcl;    break;
  default:             return(EINVAL);
  }
  dp->dmap_io = gen_io;
  dp->dmap_driver = proc_nr;
  return(OK);
}
```

5.3.3 Dynamic controller-driver mappings

An important change to MINIX' kernel was required for the AT_WINI driver. In MINIX 2.0.4 up to four process table slots are reserved for the hard disk controllers CTRLR(N), where N indicates the controller number. The actual controller types are dynamically determined by the function mapdrivers() in *src/kernel/table.c*. This function maps the appropriate device drivers to the controller slots by inspecting the boot monitor variables `'cN'`. If the user, for example, sets the boot monitor variable `'c0=at'` the AT_WINI device driver will be loaded for CTRLR(0).

With this approach several disk drivers are compiled into a single kernel image, and the ones that are needed are selected at boot time. This allows using the same distribution of MINIX for different platforms. Once the user has determined which device drivers are needed MINIX can be recompiled without the unwanted drivers.

In the original design the FS does not know about different types of disk drivers, but merely distinguishes the different controller numbers and relies on the kernel to map the controller numbers to actual disk drivers. Unfortunately, this approach does no longer work for user-space disk drivers. Moreover, the mapping is implemented in the kernel while this typically is under the responsibility of the FS. Therefore, this mechanism was moved to the FS. A welcome side-effect is that the removal further reduced the size of MINIX kernel.

The new code relating to device driver mappings is contained in the file *src/servers/fs/dmap.c*. The mappings are stored in the FS table dmap. The function map_driver() that allows to dynamically update a mapping is shown in Listing 5.4. It returns normal error codes so that it can be used from a system call that tries to dynamically install a new driver. The function is not further discussed here, though.[2]

The function map_controllers() replaces the kernel code that maps disk drivers to controllers on startup. The function maintains a local table, drivertab, which associates controller types to device driver identifiers. For each controller, the boot monitor parameter `'cN'` is analyzed to determine the type of disk is attached. If a known controller type is found the identifier of the device driver that handles it is fetched from the local drivertab, and used to lookup the process number of the driver. Finally, the function map_driver() is called to update the dmap table to the user's selection.

5.4 A user-space TTY driver

The TTY task is the last device driver that was removed from the kernel in this project. MINIX' memory-mappped terminal driver actually consists of multiple drivers in one. Its primary responsibilities are handling keyboard input and screen display, but the TTY driver also provides optional support for RS-232 lines and pseudo-terminals. The optional components are not available in the user-space TTY driver due to time constraints. Therefore, support for RS-232 lines and pseudo-terminals is part of future work.

The TTY driver had by far most dependencies. This is partly caused by some design problems, but the number of dependencies also relates to the terminal driver's complexity. An overview of all dependencies is given in Figure 5.4. The dependencies that were previously encountered will not be treated again. Specific problems that complicated the transformation of the TTY task into a user-space driver are discussed below.

[2]Currently this feature is only used by the FS for the disk driver mappings and to dynamically load the INET server. Dynamic control over other system services such as device drivers is part of future work. All device drivers, for example, are still part of the boot image.

Symbol		**Purpose**
_cause_sig	F	Signal a process when the user types `'DEL'` to alike
_cons_stop	F	Switch to primary console when MINIX shuts downs
_clock_stop	F	Reset the clock to the BIOS rate during shutdown
_current	V	Currently active console (visible to the user)
_data_base	V	Base of kernel's data segment to get physical addresses
_ ... _stop	F	Stop various device drivers when MINIX shuts downs
_enable_irq	F	Enable interrupts during initialization or after restart
_get_uptime	F	Get current time to set a new watchdog timer
_inb	F	Input a byte from the controller's data/ status register
_interrupt	F	Interrupt handler notifies the driver about interrupt
_intr_init	F	Reinitialize the interrupt controller to BIOS defaults
_level0	F	Make function call at the highest CPU privilege level
_mem_vid_copy	F	Copy characters from TTY to video memory
_micro_delay	F	Delay a few microseconds before retrying an operation
_monitor	V	Return to the boot monitor when MINIX shuts down
_mon_params	V	Boot monitor parameters that are run upon returning
_mon_return	V	Kernel variable indicating whether return is possible
_numap	F	Map a virtual address to physical address for copying
_outb	F	Output a byte to the controller's data/ control register
_panic	F	Panic if something very bad happens
_pc_at	V	Inspect if bus is AT to determine size of video RAM
_phys2seg	F	Install LDT descriptor to allow access to video RAM
_phys_copy	F	Copy from or to the BIOS or user process
_pproc_addr	V	Pointers for fast access into the process table
_proc	V	Get process details and make debug dumps
_protected_mode	V	Determine if MINIX runs 'real' or 'protected'-mode
_put_irq_handler	F	Driver sets interrupt handler during initialization
_reset	F	Do a hard reset when MINIX shuts down
_tmr_exptimers	F	Check for and run expired watchdog timers
_tmr_settimers	F	Set a new watchdog timer for the TTY driver
_tty_table	V	The main data structure of the TTY driver
_tty_timeout	V	Set by the CLOCK if a TTY timeout is detected
_tty_timelist	V	List if watchdog timers maintained by the TTY driver
_vid_vid_copy	F	Copy video memory around to scroll or clear the screen

Figure 5.4: Dependencies of the TTY task in MINIX 2.0.4. Symbols are either a function (F) or variable (V). See Figure 2.3 for the full dependencies matrix.

5.4.1 Redesign of MINIX' shutdown code

A substantial number of dependencies are caused by the shutdown code of MINIX 2.0.4. The dependencies relating to MINIX shutdown, for example, include _level0, _monitor, _reset and _ ... _stop. The latter are the cleanup functions of various other device drivers that are part of the kernel.

Since MINIX' shutdown is an important, system-wide event the choice to put this code in the TTY driver can be considered bad design. Most dependencies were automatically solved when the shutdown code was moved to a more central location in the kernel. The _..._stop dependencies, however, posed several problems when other device drivers were tranformed into user-space programs. Therefore, a new design for MINIX shutdown sequence was required. This is discussed in detail in Section 3.4.

5.4.2 Making debug dumps

The TTY task in MINIX 2.0.4 can make debug dumps of the process table. This feature is useful for debugging the system, especially when an unexpected kernel panics occur. Originally, the debug dumps were directly processed by the TTY driver, but the transformation of the TTY task into a user-space program required a radically different design. This is discussed in detail in Section 4.1.

5.4.3 Outputting diagnostics

The TTY driver in MINIX 2.0.4 is a memory-mapped terminal driver. It directly operates on the video memory to display characters with help of the assembly support routines mem_vid_copy() and vid_vid_copy() that are defined in *src/kernel/klib.s*. In protected-mode the MMU normally only permits processes to access their own data segment. The TTY task, however, can also access the video memory because it installs a segment descriptor for it in the global description table (GDT) during its initialization.[3] This is done with a call to phys2seg(), which returns a segment selector that is used by the assembly support routines.

Diagnostic output is handled in a three different ways. User programs use the function printf() from the standard C library that sends all output to the FS, which, in turn, forwards the request to the TTY driver. System services use two different approaches. User-space system services use a simplified version of printf() that directly sends all output to the TTY task with a SYS_PUTS system call. Kernel tasks have their own version of printf(), which directly calls the TTY task's output routines.

The transformation into a user-space TTY driver caused three problems relating to diagnostic output. First of all, the user-space TTY driver no longer has the required privileges to directly write to the video memory. Second, user-space system services no longer can use the SYS_PUTS system call. Third, kernel-space services also cannot output diagnostics as before.[4] The solution to each problem is discussed below.

[3]Because the video segment descriptor is put in the GDT all kernel tasks can access the video memory. It would be better to install the video segment descriptor in the local descriptor table (LDT) of the TTY task so that only the terminal driver has access.

[4]Note that nothing changes for user processes. Their output still works because this still is done via the FS, which directly forwards the request to the TTY driver—regardless of whether it is in kernel-space or in user-space.

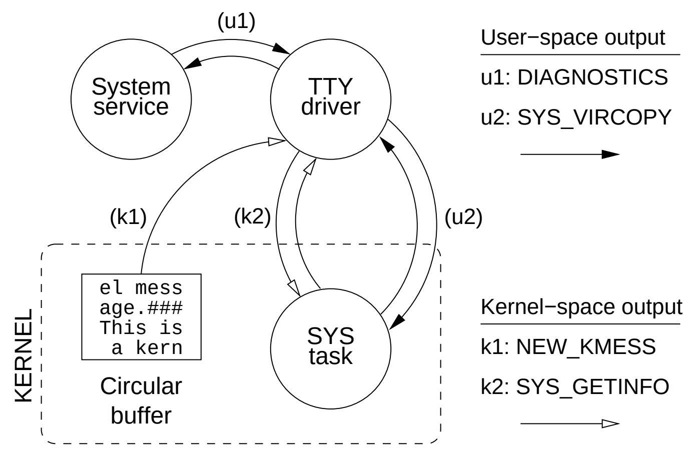

Figure 5.5: All diagnostic output by system services ends up at the user-space TTY driver. User-space system services make a direct DIAGNOSTICS request, whereas kernel tasks buffer their messages locally and send a NEW_KMESS the TTY driver. The TTY driver copies the data to printed in different ways as well.

Using the video memory

The user-space TTY driver can no longer directly update the protected-mode descriptor tables that are used by the MMU to validate all memory accesses. Therefore, a new system call, SYS_PHYS2SEG, was implemented to do this.[5] The user-space TTY driver can request the kernel to update its local descriptor table (LDT), which contains the memory segments that are specific to the TTY process. It is used instead of the GDT to restrict access by other user-space processes. The system call returns a segment selector so that the assembly support routines could be used by the user-space TTY driver without modifications.

Output from system services

Output from system services no longer goes through the SYSTEM task, but is directly sent to the user-space TTY driver. The implementation required a small modification to the printf() function in the system library. Instead of sending SYS_PUTS request to the SYSTEM task, a newly defined request, DIAGNOSTICS, is sent to the user-space TTY driver. The request includes the virtual address and the length of the string to be printed. When the TTY driver receives a DIAGNOSTICS message it copies the data to its own address space with a SYS_VIRCOPY system call and directly outputs the string to the video memory. This is shown in Figure 5.5.

[5]Although the functionality offered by the SYS_PHYS2SEG system call is required by the TTY driver, its interface may be subject to change because the design of a proper interface for resource management is part of future work.

Listing 5.5: The kernel uses a simplified version of printf() to output diagnostics.

```
PUBLIC void kprintf(fmt, arg)
const char *fmt;                                        /* string to be printed */
karg_t arg;                                             /* argument for format */
{
  int c;                                                /* next char in fmt */
  unsigned long u;                                      /* holder for number */
  int base;                                             /* base of number arg */
  int negative = 0;                                     /* print minus sign */
  static char x2c[] = "0123456789ABCDEF";               /* nr conversion table */
  char ascii[8 * sizeof(long) / 3 + 2];                 /* for ascii number */
  char *s = NULL;                                       /* string to print */

  /* Process a single character at a time, until at the end. */
  while((c=*fmt++) != 0) {
      if (c == '%') {                                   /* expect format '%?' */
          switch(c = *fmt++) {                          /* switch on key '?' */
          case 'd':                                     /* output decimal */
              u = arg < 0 ? -arg : arg;
              if (arg < 0) negative = 1;
              base = 10;
              break;
          case 'u':   ... ;   break;                    /* output unsigned long */
          case 'x':   ... ;   break;                    /* output hexadecimal */
          case 's':                                     /* output string */
              if ((s=(char *)arg) == NULL) s = "(null)";
              break;
          case '%':                                     /* output percent */
              s = "%";
              break;
          default:                                      /* echo back '%?' */
              s = "%?";
              s[1] = c;                                 /* set unknown key */
          }

          /* Convert number to ascii and do actual ouput for '%?'. */
          if (s == NULL) {
              s = ascii + sizeof(ascii)-1;
              *s = 0;                                   /* work backwards */
              do { *--s = x2c[(u % base)]; }
              while ((u /= base) > 0);
          }
          if (negative) kputc('-');                     /* print sign if negative */
          while(*s != 0) { kputc(*s++); }               /* print string / number */
      }
      else {                                            /* ordinary character */
          kputc(c);                                     /* print and continue */
      }
  }
  kputc(END_OF_KMESS);                                  /* terminate output */
}
```

Listing 5.6: The function kputc() is used by kprintf() to accumulate characters of a kernel message in a circular buffer. When the TTY receives a NEW_KMESS notification it requests a copy of the buffer with SYS_GETINFO to display the message.

```
PRIVATE void kputc(c)
int c;                                               /* char to append */
{
/* Accumulate a single character for a kernel message. */
  if (c != END_OF_KMESS) {                           /* normal character */
      kmess.km_buf[kmess.km_next] = c;               /* put char in buffer */
      if (kmess.km_size < KMESS_BUF_SIZE)            /* increment until full */
          kmess.km_size += 1;
      kmess.km_next = (kmess.km_next + 1) % KMESS_BUF_SIZE;
  } else {                                           /* end of message */
      notify(TTY, NEW_KMESS);                        /* notify the TTY */
  }
}
```

Kernel messages

Diagnostics messages from within the kernel pose a more serious problem. Replacing the direct TTY function calls with request messages like above is not an option, because this would block the kernel if the user-space TTY driver is not ready. The request may even cause a deadlock when the TTY did a system call that triggered the output. In this case there is a cyclic dependency between the TTY driver and the SYSTEM task.

The solution that was chosen is to let the kernel buffer its diagnostic messages until the TTY driver is ready to display them. To prevent buffer overflows in the kernel a simple circular buffer is used. The buffering scheme is implemented by the functions kprintf() and kputc() that are shown in Listing 5.5 and 5.6, respectively.

The function kprintf() proceses a kernel message and accumulates the output in a buffer by calling kputc(). It terminates all kernel message with a END_OF_KMESS character to trigger kputc() to send a NEW_KMESS notification to the TTY driver. When the TTY receives the notification it requests a copy of the buffer with the SYS_GETINFO and displays the message. This is shown in Figure 5.5

Chapter 6

Related work

This chapter surveys related work in microkernel operating systems. The study includes Mach, QNX and L4 because these systems covers a whole range of time and both open and commercial systems. Mach was one of the first microkernels around and is interesing from a historical perspective. Although its design is far from perfect, it has influenced many other systems. QNX is a commercial system that is targeted towards embedded systems. The QNX platform provides a full-featured environment on top of the Neutrino microkernel. Finally, L4 is a recent effort that has proven that microkernels do not necessarily have a bad performance. L4 actually is a kernel API with implementations for different hardware architectures. For each system a general introduction is given, the design is explained, and possible applications are highlighted.

6.1 CMU Mach

Mach is a microkernel operating system that was developed from 1985 to 1994 at Carnegie-Mellon University (CMU). It was one of the first microkernels around and introduced several new concepts that influenced many other projects. Most notable is the concept of a user-space pager. Mach was originally developed as a part of BSD UNIX, but became a microkernel when BSD UNIX-specific code was moved to user-space servers [3, 19].

Although the official Mach project was discontinued in 1994, development work on Mach continued at the Open Software foundation, University of Utah's Flexmach project, Helsinki University of Technology's LITES system and the Free Software Foundation's GNU/Hurd system. One of the most prominent and recent traces of Mach, for example, can be discovered in Apple's Mac OSX.[1]

[1]Max OSX is based on a version of BSD UNIX known as Darwin. According to Apple's web site (*http://developer.apple.com/darwin/*), "Darwin integrates a number of technologies, most importantly Mach 3.0, ..."

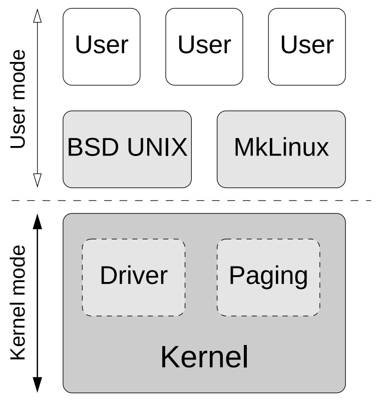

Figure 6.1: Mach 3.0 has a hybrid microkernel and is typically used to host multiple OS personalities. Devices drivers and the paging policy are part of the kernel. The single-server OS functions as user-space pager for all user processes.

6.1.1 Kernel properties

The discussion that follows focuses on Mach 3.0 that was released in 1986. Mach's kernel can be characterized as a hybrid microkernel because important system services, such as device drivers, are part of it. Later work on Mach [20] also provided a framework for user-space device drivers, though. The kernel is written in C and assembly and comprises about 75,000 lines of code (LoC).[2] Mach has been implemented on several architectures, including i386/486, Alpha and MIPS.

Mach's kernel provides the notion of tasks that group system resources. System resources in Mach are abstracted by means of ports, which can be thought of as one-way communication channels. Associated with each task are a set of port rights and one or more threads that run in the context of a task and share all its resources. UNIX' extended process model must be provided by user-space servers. Mach thus uses a distributed process model.

Mach introduced the concept of a user-level pager [21]. Page faults are detected by the kernel and forwarded to a user-space pager that is responsible for providing the requested data. Unfortunately, the Mach's pageout policy still is an integral part of the kernel. This fixed paging policy could not provide the flexibility required by specific types of applications.

6.1.2 Applications of Mach

Mach was typically used to host one or more single-server operating systems known as 'OS personalities,' including BSD UNIX and MkLinux. In this setup, the operating system server functions as a user-space pager for the

[2]This was measured with the `'sloccount'` utility, which was also used to estimate the size of MINIX' kernel. For MINIX 2.0.4 this number is about 20,000 LoC and the kernel of MINIX 3.0.0 has about and 7,500 LoC.

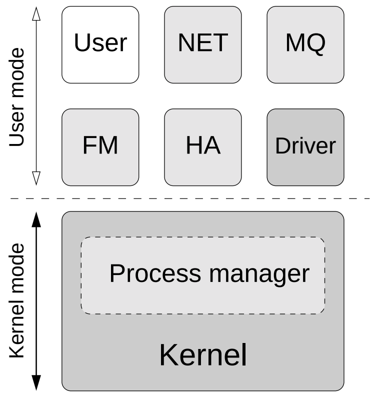

Figure 6.2: QNX is a multiserver operating system with a hybrid microkernel, Neutrino 6.3. All system services behave like ordinary user processes, except for the mandatory process manager that shares its address space with the kernel.

entire memory. Examples of OS personalities that have been hosted on top of Mach include MkLinux, BSD UNIX, OSF/1, HP-UX and OS/2 [8]. This is illustrated in Figure 6.1.

Mach provides binary compatibility with legacy applications that run on the hosted operating systems. This works by means of trap redirection [7]. Mach injects an emulation library in the address space of user processes, which causes system calls to trap to Mach's kernel. The kernel catches the trap and redirects the request to the operating system server.

Later work on Mach provided several advances. For example, Mach 4.0, which was developed at the University of Utah, supports user-space device drivers and thus has a smaller kernel. Furthermore, several multiserver operating systems, such Mach-US and GNU/Hurd, were also developed for variants of Mach.

6.2 QNX Neutrino RTOS

QNX is a commercial, real-time operating system (RTOS) with a microkernel architecture [4]. Because of its real-time properties it is widely used in embedded devices, including medical appliances and multimedia systems. QNX was originally created at the University of Waterloo, but has been commercialized and produced by QNX Software Systems since 1981.

Because QNX is a commercial system, no source code is available and only a few scientific publications on QNX exist. This makes it hard to compare QNX to other systems. The QNX web site, however, provides a 'system architecture guide' [22] that gives insights in the design goals, IPC facilities, system services, and so on. This guide was used for the discussion that follows.

6.2.1 System architecture

The core of the QNX platform is the Neutrino 6.3 microkernel, which has been developed since 2001. All operating system services, except for the mandatory process manager (PM), are standard processes. Because the process manager and the microkernel share the same address space, QNX can be characterized as a multiserver operating system on top of a hybrid microkernel. QNX has been implemented on various platforms, including Intel x86, MIPS, PowerPC and StrongARM, and is primarily written in the C programming language.

The PM's primary responsibilities are process management, memory management and pathname management. The latter is used to bind resource managers to the system's 'message passing bus.' QNX's IPC is driven by ordinary POSIX system calls that operate on the pathname space. When a process opens a file, the open() library routine first contacts the PM to look up the server that manages the pathname, and then transparently sends an OPEN message to the server. QNX provides mechanisms to dynamically control system services and the pathname space.

The structure of QNX is illustrated in Figure 6.2. Apart from the provided functionality, there is no distinction between user processes and system services. The figure show a selection of the system services provided by QNX. It includes a network server (NET), message queue manager (MQ), file manager (FM), a high availability manager (HA), and device drivers.

Although the size of QNX's microkernel could not be measured because it is a closed source system, it probably is much larger than the kernel of L4 or MINIX. QNX' kernel provides a limited set of kernel abstractions, but is full-featured at the same time. Process scheduling, for example, comes in many flavors such as FIFO priority, round-robin and sporadic scheduling. Furthermore, the process manager is also part of the kernel's address space and thus should be taken into account.

6.3 L4 microkernel API

L4 is a portable microkernel API that has been developed by Jochen Liedtke. The original L4 kernel for Intel x86 platforms has been developed at German National Center for Computer Science (GMD) and IBM TJ Watson Research Center since 1995 [2]. Later, L4 development continued at University of Karlsruhe as part of the L4Ka research project.

L4 has that demonstrated that microkernels are not necessarily slow. It has proven that high performance can be realized by implementing only a minimal set of abstractions and tuning each implementation to a particular platform [13, 15]. The performance of L^4Linux, for example, only shows a 2 to 4% slowdown on an industry benchmark compared to vanilla Linux.

While the microkernel API is portable, its implementation is not. The original L4 kernel, L4/x86, was implemented in assembly language to obtain a maximum performance. Unfortunately, this made the kernel hard to maintain and port to other platforms. Therefore, later L4 implementations were realized in a higher-level programming language [2].

There exist numerous applications of the L4 microkernel in various application domains. Single-server operating systems on top of L4 include L^4Linux—which is modeled after Mach's MkLinux—and DROPS [23] with a 'tamed' version of $L4^4$Linux [24]. Multiserver environments also exist, for example, IBM's SawMill Linux and L^4MINIX. There also exist specialized applications on top of L4, such as Perseus that provides secure digital signatures. The last two applications are discussed below.

6.3.1 L4 implementations

Currently, there exist three versions of the L4 microkernel API as well as several implementations for different hardware architectures. The different API versions are called V.2, X.0 and X.2. The most recent version is the experimental X.2 API that is designed for portability among 32-bit and 64-bit hardware architectures. The APIs were used for the following L4 implementations:

L4/x86. This is the original version of L4 by Jochen Liedtke, which was developed at GMD, IBM Watson and University of Karlsruhe since 1995. L4/x86 implements the V.2 and X.0 API in assembly language. While L4/x86 is targeted towards the Intel i468 and IA32, variants for other architectures include L4/MIPS, L4/Alpha, L4/PowerPC. Development has been discontinued.

L4Ka::Hazelnut. This is a reexamination of L4/x86 at University of Karlsruhe since 2000. L4Ka::Hazelnut basically is Liedtke's original L4/x86 kernel in a higher-level language for portability. It implements the X.0 API in C++ on the IA32 and ARM architectures. Development was discontinued late 2001.

L4/Fiasco. This is a L4 branch at the Technical University Dresden. L4/Fiasco is meant as a base for the Dresden Realtime OS (DROPS). The kernel has a binary L4 interface, but was enhanced with real-time properties for DROPS. It implements the V.2 and X.0 API in C++. Development on L4/Fiasco has been done since 2003.

L4Ka::Pistachio. This is the most recent version of L4. L4Ka::Pistachio is a pure L4 kernel that has been developed at University of Karlsruhe and University of New South Wales (UNSW) since 2003. It implements the X.2

API in C++ on a variety of architectures, including IA32, IA64, ARM, Alpha, AMD64, MIPS, and PowerPC. The latest version, L4Ka::Pistachio 0.4, was released in June 2004.

6.3.2 L4Ka::Pistachio

This subsection focuses on most recent implementation of the L4 microkernel API. L4Ka::Pistachio [25] is a minimal microkernel that provides only the most requires features. L4 provides basic task management. A task in L4 is composes of an address space and one or more threads of execution. It is up to the user-space applications to provide extended process models. The kernel implements POSIX threads with priority-based, preemptive scheduling. Interprocess communication is based on rendezvous message passing.

L4's resource manager (RMGR) is the first user-space task that is started at boot time when the microkernel has been initialized. It has control over all system resources, including main memory, IRQ lines and L4 task numbers, and has privileges to distribute resources to other processes. The resource manager loads the rest of system based on a configuration file.

In contrast to Mach, paging is completely done in user space. The L4 microkernel does not implement a paging policy within the kernel, but only provides mechanisms. Three simple memory management primitives, grant, map and unmap, allow to recursively construct address spaces. A process can, for example, grant a subspace to another process or map it into another process' address space.

6.3.3 Examples of L4 applications

L^{4}Linux and Perseus Perseus is an application that provides secure digital signatures 'in the real world' [10]. The architecture of this application is shown in Figure 6.3(a). Perseus is uses L^{4}Linux as a single-server operating system—to run legacy Linux applications—next to this a protected digital signature application (SA). Legacy software and viruses cannot interface with the SA module because it is physically protected by the MMU. The operating system effectively resides in a sandbox and cannot access the secrets at the SA.

A system-wide security policy is realized by the resource manager (RM), trusted user-interface (TUI) and application manager (AM)—as well as some other components that are not treated. The TUI and AM tell the user which process is currently running by means of a reserved 'secure line' on top of the screen. This way the user can check which application is in control, and can securely sign documents when the SA runs.

SawMill Linux Another interesting research project is SawMill Linux, which is a multiserver Linux operating system [11]. Its architecture is shown

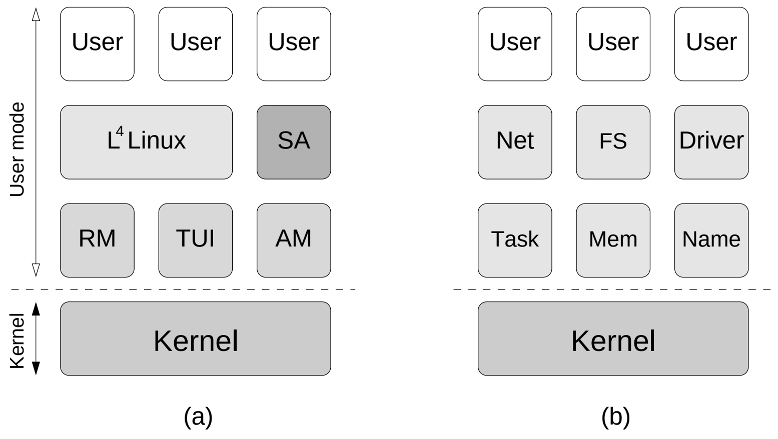

Figure 6.3: Two application structures on top of the L4 microkernel: (a) Perseus has as a single-server operating system with specialized component for secure digital signatures next to it, whereas (b) SawMill Linux is a multiserver operating system.

in Figure 6.3(b). As with Perseus, the lowest layer is responsible for resource management. It includes process and task management (TASK), memory management (MEM) and a name server (NAME). Other components that are found in SawMill Linux include a network server (NET), file system (FS) and device drivers.

The development of an efficient multiserver protocol was one of the research goals. The protocol tries to reduce IPC by directly calling a processing server whenever possible and by securely sharing data among servers to prevent copying. Measurements with an industrial benchmark indicated that L^4Linux and SawMill Linux can obtain a similar performance.

6.4 Comparison with MINIX

In this section, MINIX is compared to the microkernels that are discussed in the previous sections. The comparison will be based on MINIX 2.0.4 as well as on the new version, MINIX 3.0.0. More information about MINIX 2.0.4 is provided in Subsection 1.3.

Figure 6.4 provides an overall impression of the various microkernels. The table immediately shows the key differences between MINIX 2.0.4 and MINIX 3.0.0. The kernel has been transformed from a hybrid microkernel into a true microkernel by moving the device drivers to user space. This reduced the number of lines of code (LoC) that is part of the kernel from 20,000 to approximately 7,500 LoC in MINIX 3.0.0.

The main question is how MINIX compares to the other systems, though. Mach has been included because of its historical value, but the comparison between QNX, L4, and the new version of MINIX is most interesting.

	Mach	**QNX**	**L4**	**MINIX**	
Year	1986	2001	2003	2003	2005
Version	CMU Mach	Neutrino	Pistachio	‘old’	‘new’
Release	3.0	6.3	0.4	2.0.4	3.0.0
Microkernel	hybrid	hybrid	true	hybrid	true
- drivers	kernel	user	user	kernel	user
- manager	user	kernel	user	user	user
Language	C	C	C++	C	C
Kernel LoC	75,000	n.a.	10,000	20,000	7,500
Open source	yes	-	yes	yes	yes
Single-server	yes	-	yes	-	-
Multiserver	-	yes	yes	yes	yes

Figure 6.4: This figure compares MINIX 2.0.4 and the new version, MINIX 3.0.0, to the microkernels that are surveyed in this chapter. The ‘manager’ entry represents what is known as the process manager (PM) in QNX or the memory manager (MM) in MINIX. The number of kernel lines of code was measured with the `‘sloccount’` utility.

QNX is a reliable and full-featured system that is widely used for embedded system. Commercially, QNX is very successful. Compared to MINIX, however, QNX has the down side that it is not freely available and does not come with all source code available. Although MINIX does not have as many features, its simplicity, the amount of documentation that is available, and the fact that it has a liberal licensing model makes it an interesting competitor in certain markets. From a design perspective, MINIX 3.0.0 also is interesting because it has true microkernel, whereas QNX has hybrid microkernel.

L4 is the current state of the art in microkernel design. L4 is a relatively small and fast microkernel, which has gradually evolved for over a decade. Although the kernel of MINIX 3.0.0 is actually smaller, it compromises on functionality such as virtual memory. When L4 and MINIX’ microkernels are compared L4 probably is better because it is much more mature. Research on L4, however, has not really focused on multiserver applications on top of L4. The SawMill Linux project is interesting in this respect, but has been discontinued since 2001. MINIX on the other hand provides a complete, POSIX conformant multiserver operating system.

All in all, MINIX has a number of desirable properties and may well find its own niche next to QNX and L4. This master’s project is just a first step in this direction, though. MINIX 3.0.0 is still in an early stage of development and has to become more mature.

Chapter 7

Summary and conclusion

This chapter concludes this master's thesis. The first section starts with an overview of the major contributions of this project by summarizing the results presented in the previous chapters. Section 7.2 discusses whether the goal that was set forth in the title of this master's thesis, 'Towards a True Microkernel Operating System', is fulfilled and draws conclusions. Finally, Section 7.3 ends with a description of open issues and outlines directions for future research.

7.1 Contributions

Supporting user-space device drivers When device drivers are transformed to independent, user-space programs they lose privileges and can no longer directly access kernel data structures. Section 3.1 discusses new system calls and other kernel changes that are needed to support user-space device drivers.

First of all, several system calls to perform device I/O on behalf of a user-space driver were added. Variants include calls to read of write a single device register or a series of registers.

Interrupt handling is a typical device driver function, but user-space device drivers have no privileges to handle interrupts. A system call to enable or disable interrupts was added. The call also allows to add a policy to be executed by a generic interrupt handler at the kernel.

A generic virtual copy system call was added to copy between processes, from or to the BIOS, and from or to remote memory areas. This system call replaced the copy function with physical addressing that was previously used. Using virtual addressing is easier to control and thus more secure.

Many device drivers require information about the system environment or kernel settings. A new system call was added to request a copy of certain system information. The call is also used by a new information server that provides debug dumps of entire data structures.

Interprocess communication Interprocess communication (IPC) facilities are discussed in Section 3.2. MINIX' IPC is characterized by rendezvous message passing, which must be used with care to prevent blocking kernel tasks and deadlocks. The partial message ordering of MINIX 2.0.4 circumvents the problem, but does protect kernel tasks from being blocked. An important change to the system call handler solved this problem by verifying that the caller is waiting for a reply message.

MINIX' system call handler was updated in other ways as well. System call errors are now properly handled. Furthermore, nonblocking system calls were added to prevent a process from being blocked when the other side is not ready. Finally, a system call protection mechanism that checks whether a process is allowed to communicate with another process was added. A process now is only permitted to another process when that process' bit is enabled in its send mask entry in the process table. The result is a clean separation of policy and mechanism.

Dealing with asynchronous events MINIX 2.0.4 uses several different approaches to deal with asynchronous events in the kernel. A new notification construct obviated the need for this and has several important benefits for MINIX' kernel. The new construct can safely be used to handle all kinds of asynchronous system events, because it circumvents race conditions and does not block the caller when the destination is not ready to receive a message. This allowed to remove several exceptional cases that existed in MINIX 2.0.4 and provided an elegant solution for the problems relating to MINIX' shutdown sequence and the SYN_AL task. The latter, for example, could be removed in its entirety. Moreover, it simplifies and beautifies the kernel's source code, because all asynchronous events are now handled in a uniform way. This mechanism is treated in detail in Section 3.3.

A new shutdown sequence The shutdown sequence of MINIX was completely revised as discussed in Section 3.4. The most important benefit is a new stop sequence that notifies all system service of the upcoming shutdown, instead of abruptly shutting down as in MINIX 2.0.4. This is done by sending an alert with the new notification construct. The alert are sent according to the dependencies between different process types so that all processes get a fair chance to clean up. The FS server, for example, is notified before the device drivers so that it can still rely on them to shutdown. Only when all processes have exited—either gracefully or forcibly—MINIX is really shutdown.

A new information server MINIX 2.0.4 allows the user to make debug dumps of certain process table information by pressing `'F1'` or `'F2'`. This is done by the TTY task in the kernel. Since the TTY was moved to user-space and since directly touching important kernel data structures is not a good design a new approach was presented in Section 4.1.

The user-space TTY driver allows other process to register for notifications when function keys are pressed. This is much like the 'observer' design pattern. While certain function keys are reserved, the keys `'F1'`-`'F12'` and the combinations `'Shift-F1'`-`'Shift-F12'` can be observed. Notifications are sent using the new nonblocking send function to prevent the TTY driver from being blocked.

A new information server, IS, was created to handle the debugging dumps that were previously done withing the kernel, as well as for handling various new dumps. The IS server has registered the `'F1'`-`'F12'` keys a the TTY driver and blocks waiting until the user makes a request. When a function key is pressed, the IS servers requests a copy of the associated kernel data structure and dumps the information on the primary console.

Generic management of watchdog timers Device drivers in MINIX 2.0.4 heavily rely on watchdog timers that are managed by the CLOCK task. Unfortunately, this functionality is no longer accessible when the drivers are moved out of the kernel because the CLOCK task cannot directly call a watchdog function across address spaces. Therefore, the timer management functionality was extracted from the CLOCK task and made available as regular library functionality that is available to all processes. This is discussed in Section 4.2.

The generic timer management functionality has several benefits. First of all, it the makes it straightforward to maintain multiple watchdog timers with only a single synchronous alarm available. This allowed moving device drivers to user-space without compromising on timers. Furthermore, it puts the responsibility for running the watchdog functions of expired timers where it belongs, that is, at the process that actually uses the timers. Finally, the new approach resulted in simpler source code.

Dealing with unresponsive hardware The approaches for detecting timeouts that are used in MINIX 2.0.4 do not work for user-space device drivers. Therefore, two new approaches were devised as discussed in Section 4.3. A new timeout flag alarm was used to replace checking the elapsed time with the microseconds counter of the 8253A timer. Furthermore, a synchronous alarm was used replace watchdog timers that faked a hardware interrupt upon expiration.

MINIX' kernel was updated to use separate timer structures for each type of alarm and for each process so that different types of alarms can be combined safely. This is, for example, required by the RTL8139 driver.

Finally, Section 4.3 illustrates an advanced timer management scheme. This scheme is discussed in Section 4.2 and allows user-space processes to

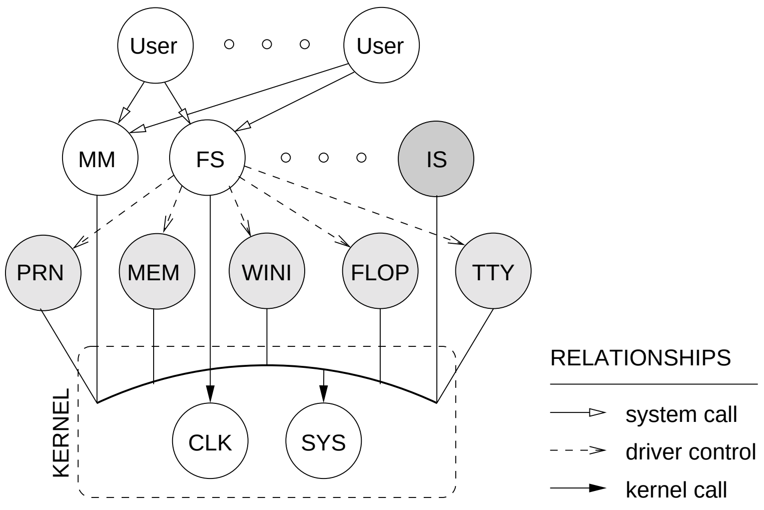

Figure 7.1: The outcome of this master's project: a multiserver operating system with a true microkernel. MINIX 3.0.0 has a true microkernel with user-space PRINTER, MEMORY, AT_WINI, FLOPPY, and TTY device drivers. The new IS server for debugging dumps is also shown.

have multiple outstanding watchdog timers with only a single synchronous alarm at their disposal. For example, the FLOPPY and TTY drivers require this scheme because they manage multiple devices.

User-space device drivers The main contribution of this work is the successful transformation of five device driver tasks that were an integral part of the MINIX' kernel into independent user-space programs. This result was made possible by a combination of the above contributions. Listed in the order that they were removed from the kernel, MINIX now has a user-space PRINTER, MEMORY, AT_WINI, FLOPPY, and TTY driver. This is illustrated in Figure 7.1.

The device drivers were removed by replacing their dependencies with alternatives that provide similar functionality in a user-space setting. The contributions that are discussed above made this possible. The removal of individual dependencies is not further discussed. Several improvements to MINIX that were triggered by the transformation into user-space drivers are discussed below, though.

Other contributions Process scheduling was greatly improved when the PRINTER driver was moved to user-space. The four fixed priorities that are based on the type of process were replaced by a generic multilevel scheduling algorithm where priorities can be set for individual processes. This provides better control over important system processes.

The transformation of the AT_WINI device driver revealed that the kernel of MINIX 2.0.4 dynamically maps controllers to disk drivers. Because this is a typical FS functionality and the original approach does not work for user-space device drivers the code was moved to the FS.

The device independent code was updated so that it could be used by the user-space device drivers. This not only concerned removing kernel dependencies, but the interface was updated as well to support MINIX' new shutdown sequence and to facilitate the management of watchdog timers. The latter, for example, was required by the FLOPPY driver.

The removal of the TTY driver from the kernel caused several problems relating to diagnostic output. The most important problem was that the kernel could no longer output diagnostic messages to the console. The solution that was implemented is to collect kernel messages in a circular buffer and notify the user-space TTY driver about them. When the TTY driver is ready it can request a copy of the kernel messages and output them to the primary console.

7.2 Retrospective

With the overview of the major contributions in Subsection 7.1 this master's thesis almost comes to end. This section briefly looks back to what was accomplished. Various kernel improvements and new applications made it possible to strongly reduce MINIX' kernel in size by transforming the most important device drivers into independent user-space device drivers.

The main contribution of this work is that MINIX 2.0.4 was fully revised to become a multiserver operating system with a true microkernel. In a stand-alone configuration without network support MINIX 3.0.0 can be compiled with user-space drivers only. All kernel-space drivers were removed, so that only the true microkernel remains. The project's outcome is illustrated in Figure 7.1.

MINIX' renewed structure brings many of the benefits that are examined in Chapter 1. First of all, the restructuring has greatly improved the MINIX' modularity, which is beneficial for many properties, including flexibility, maintainbility, robustness and security. Furthermore, a lot of complexity has been removed from MINIX' kernel and its size was tremendously reduced by 75%, which means a great simplification and makes it less susceptible to bugs. MINIX 3.0.0 thus is a big improvement over MINIX 2.0.4.

All in all this master's project has been very successful. MINIX' kernel has been transformed from a hybrid microkernel with device drivers to a true microkernel without device drivers. The goal that was set forth in the title of this master's thesis, 'Towards a True Microkernel Operating System', thus has been fulfilled. MINIX 3.0.0 is an important step forward, but, nevertheless, a lot of work remains to be done. The next subsection outlines possible directions for future research.

7.3 Future work

Performance analysis A preliminary investigation of the performance of user-space device drivers is part of the problem analysis. In Section 2.1 the time needed for typical request-response sequence is measured as illustrated in Figure 2.1. Unfortunately, the timing measurements do only provide an indication of the incurred overhead per context switch. The determine the actual performance penalty, however, the number of additional context switches and data copies that are required by user-space device drivers also must be taken into account.

Therefore, a more detailed performance analysis on modern machines must be performed for the user-space device drivers. In special, the number of extra context switches and data copies must be analyzed. Since the device drivers that were removed from the kernel are different in nature this analysis should be done for each class of drivers.

Performance optimizations Depending on the results of the detailed performance analysis, performance optimizations may be needed. It was not yet investigated whether a better performance can be obtained by applying different problem solving strategies. In general, all dependencies were replaced by a similar alternative that does the same job in user space, but no changes were made to the algorithms that are used.

Performance bottlenecks, for example, may be dealt with by combining multiple requests, by relocating tasks to the server where they are most used, or by redefining the communication protocol. Related research on SawMill Linux, for example, was aimed at finding an efficient multiserver protocol that minimizes IPC.

Other user-space device drivers In a stand-alone configuration without network support MINIX 3.0.0 can be compiled with only user-space device drivers. This setup is illustrated in Figure 7.1. To increase the usefulness of the system more user-space device drivers must become available. A user-space Ethernet driver probably has the highest priority in order to support normal networking functionality. The RTL8139 device driver seems a good candidate because network cards with Realtek RTL8139 chip sets are widely available for prices below 10 euros.

User-space Ethernet drivers form an interesting class of device drivers from a performance perspective. While Realtek RTL8139 based cards perform at 10/100 Mbps (Fast Ethernet), modern network interface cards perform at 1/10 Gbps (Gigabit Ethernet). The design and implementation of a user-space Ethernet driver that can manage such data transfer rate seems a challenging effort.

Dynamic control over system services Section 2.2 gives a comparison between static and dynamic control over system services. Although this project uses a static approach by including all system services in the boot image, the ability to dynamically start and stop system services provides many benefits. Therefore, an important area of future work is the design of a proper interface for dynamically controlling system services.

Among other things the design should cover the following aspects. Since system services usually have more privileges than ordinary user processes, the new interface should provide mechanisms for distributing rights. To support user-space device drivers the FS should be able to dynamically control the mapping between major devices and drivers.

Dynamic control also offers potential for automated system recovery. The system, for example, could transparently reload a malfunctioning server when its detects certain errors conditions. An interesting issue in this respect is how to deal with state information of system services.

System services as ordinary user processes System services in MINIX have a special status and are treated differently from ordinary user processes, for example, by the MM and FS. There are no obvious reasons for making this distinction, however. While system services typically have more privileges than user processes they are very similar in all other respects.

Making system services more like ordinary user processes would greatly simplify the system's architecture. The source code of the kernel, MM and FS can be simplified because exceptional cases for system services are no longer needed. Moreover, system administration becomes simpler because system services and user processes can be controlled in a unified way.

Restricting access to system resources Finally, an important area for future work is the design and implementation of mechanisms for restricting access to system resources. One issue is where and how to store the privileges of each process. As mentioned above, another interesting aspect is how to distribute the rights of each process. Currently a static approach is used, but a more intricate scheme is needed to support dynamic control over system services.

Once the mechanisms are in place appropriate policies must be defined. According to the principle of least authorization (POLA) all processes should be restricted as much as possible. This means that only those privileges should be granted that are strictly needed for the task at hand.

Bibliography

[1] Andrew S. Tanenbaum and Albert S. Woodhull. *Operating Systems Design and Implementation.* Prentice-Hall, Upper Saddle River, NJ 07458, USA, second edition, 1997. Includes CD-ROM.

[2] Jochen Liedtke, Uwe Dannowski, Kevin Elphinstone, Gerd Liefländer, Espen Skoglund, Volkmar Uhlig, Christian Ceelen, Andreas Haeberlen, and Marcus Völp. The L4Ka Vision, April 2001. System Architecture Group, University of Karlsruhe, Germany.

[3] M. Accetta, R. Baron, D. Golub, R. Rashid, A. Tevanian, and M. Young. Mach: A new kernel foundation for UNIX development. *Proceedings Summer USENIX*, July 1986.

[4] Dan Hildebrand. An architectural overview of QNX. In *Proceedings of the Usenix Workshop on Micro-Kernels and Other Kernel Architectures*, pages 113–126, Seattle, WA, USA, April 1992. Usenix Association.

[5] Brian Walters. VMware virtual platform. *Linux Journal*, 63, July 1999.

[6] Dawson R. Engler, M. Frans Kaashoek, and James O'Toole Jr. Exokernel: an operating system architecture for application-level resource management. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, volume 29, 1995.

[7] Simon Patience. Redirecting system calls in Mach 3.0, an alternative to the emulator. In USENIX, editor, *Proceedings of the USENIX Mach III Symposium, April 19–21, 1993, Santa Fe, New Mexico, USA*, pages 57–73, Berkeley, CA, USA, April 1993. USENIX.

[8] Abraham Silberschatz, Peter B. Galvin, and Greg Gagne. *Operating System Concepts.* John Wiley & Sons, Inc., sixth edition, 2001. Appendix B, The Mach system.

[9] Hermann Härtig, Michael Hohmuth, Jochen Liedtke, Sebastian Schönberg, and Jean Wolter. The performance of microkernel-based

systems. In *Proceedings of the 16th ACM Symposium on Operating System Principles*, St. Malo, France, October 1997.

[10] Birgit Pfitzmann and Christian Stüble. PERSEUS: A quick open-source path to secure signatures. 2nd Workshop on Microkernel-based Systems, Banff, Canada, October 2001.

[11] Alain Gefflaut, Trent Jaeger, Yoonho Park, Jochen Liedtke, Kevin Elphinstone, Volkmar Uhlig, Jonathon E. Tidswell, Luke Deller, and Lars Reuther. The SawMill multiserver approach. In *Proceedings of the 9th ACM SIGOPS European Workshop*, Kolding Denmark, September 2000.

[12] Jochen Liedtke. Towards real microkernels. *Communications of the ACM*, 39(9):70–77, September 1996.

[13] Jochen Liedtke. Improving IPC by kernel design. In *Proceedings of the 14th Symposium on Operating Systems Principles*, pages 175–188, Asheville, NC, USA, December 1993. ACM SIGOPS.

[14] Andread Haeberlen, Jochen Liedtke, Yoonho Park, Lars Reuther, and Volkmar Uhlig. Stub-code performance is becoming important. In *Proceedings of the 1st Workshop on Industrial Experiences with Systems Software*, San Diego, CA, USA, October 2000.

[15] Jochen Liedtke. On μ-Kernel Construction. In *Proceedings of the 15th ACM Symposium on Operating System Principles*, pages 237–250, Copper Mountain Resort, CO, USA, December 1995. ACM SIGOPS.

[16] Brian Bershad. The increasing irrelevance of IPC performance for micro-kernel-based operating systems. In *Proceedings of the Workshop on Micro-kernels and Other Kernel Architectures*, pages 205–212, Seattle, WA, USA, April 1992. USENIX Association.

[17] Henri E. Bal. Interprocess communication and synchronization based on message passing. In *Reader Parallel Programmeren, Najaar 2001.* Vrije Universiteit Amsterdam, 1995.

[18] Jonathan S. Shapiro. Vulnerabilities in synchronous IPC designs. In *Proceedings of the IEEE Symposium on Research in Security and Privacy*, pages 251–262, Oakland, CA, USA, May 2003. IEEE Computer Society Press.

[19] Daniel Julin, David Golub, Douglas Orr, Michael Jones, Richard Rashid, Richard Sanzi, Ro Forin, and Robert Baron. Mach: A foundation for open systems, September 1992.

[20] David B. Golub, Guy G. Sotomayor, and Freeman L. Rawson III. An architecture for device drivers executing as user-level tasks. In *Proceedings of the Usenix Mach Symposium*, pages 153–172, Berkeley, CA, USA, 1993. USENIX.

[21] David Golub and Richard P. Draves. Moving the Default Memory Manager out of the Mach Kernel. In *Proceedings of the Usenix Mach Symposium*, November 1991.

[22] QNX Software Systems Ltd. *QNX Neutrino RTOS architecture*, 2003. QNX website (http://www.qnx.com/) :: Developers Support :: System Architecture guide.

[23] Cl.-J. Hamann, H. Härtig, L. Reuther, M. Borriss, M. Hohmuth, R. Baumgartl, and S. Schonberg. Dresden Realtime Operating System, December 2001.

[24] Hermann Härtig, Michael Hohmuth, and Jean Wolter. Taming linux. In *Proceedings of PART'98*. TU Dresden, 1998.

[25] The L4Ka::Pistachio Microkernel (White Paper), May 2003. System Architecture Group, University of Karlsruhe, Germany.

Appendix A

Source tree organization

This appendix discusses MINIX' old and new source tree. Section A.1 provides an overview of all kernel files in MINIX 2.0.4 and shows that the majority of files belongs to device drivers. Section A gives an overview of the new source tree of MINIX 3.0.0.

A.1 Overview of all kernel files

This section gives an overview of the kernel of MINIX 2.0.4 as it can be obtained from *http://www.cs.vu.nl/pub/minix/*. The listing below shows all files in the directory *src/kernel/* together with their size and purpose. Although all files are compiled into the kernel, two groups were formed—microkernel files and device drivers files—to give a quick impression of the potential reduction of the kernel's size when device drivers are transformed into user-space servers.

In total, the kernel of MINIX 2.0.4 has 77 files and is about 878 KB in size. The microkernel comprises less files than the device drivers and, moreover, the average size per file is much smaller. Only 25% of the code (220 KB) belongs to the microkernel, whereas 75% of the code (658 KB) is taken by device drivers.

Microkernel files (25 files, 220 KB in total)

FILENAME	FILESIZE	PURPOSE
assert.h	653 B	/* header for assertions */
clock.c	2263 B	/* the CLOCK task */
const.h	5492 B	/* header with kernel constants */
exception.c	2090 B	/* exception handlers */
glo.h	3197 B	/* global variables */
i8259.c	5403 B	/* 8259 interrupt controller */
kernel.h	800 B	/* master header file */
klib386.s	16256 B	/* 80386 kernel utilities */

klib88.s	28978 B	/* 8088 kernel utilities */
klib.s	177 B	/* chooses 8088 or 80386 mode */
main.c	5123 B	/* main program of MINIX */
misc.c	6231 B	/* mem_init, env_parse, bad_... */
mpx386.s	16669 B	/* 80386 startup code + interrupt handlers */
mpx88.s	21450 B	/* 8088 startup code + interrupt handlers */
mpx.s	172 B	/* chooses between 8088 or 80386 mode */
proc.c	18928 B	/* process and message handling */
proc.h	4957 B	/* declaration of the process table */
protect.c	10804 B	/* protected mode (GDT, LDT, IDT) */
protect.h	5660 B	/* constants for protected mode */
proto.h	14769 B	/* kernel function prototypes */
sconst.h	847 B	/* constants for assembly code */
start.c	2846 B	/* C startup code for MINIX */
system.c	42107 B	/* the SYSTEM task */
table.c	5439 B	/* global data + tasktable */
type.h	3928 B	/* kernel types */

Device driver files (42 files, 658 KB in total)

FILENAME	FILESIZE	PURPOSE
3c503.c	6239 B	/* DP8390 task: 3COM Etherlink II */
3c503.h	2555 B	/* DP8390 task: 3COM Etherlink II */
aha1540.c	63465 B	/* AHA1540 task: Adaptec 154x SCSI */
at_wini.c	36213 B	/* AT_WINI task: IBM-AT winchester */
bios_wini.c	11917 B	/* BIOS_WINI task: ROM BIOS disk driver */
console.c	30324 B	/* TTY task: driver for 6845 video chip */
dmp.c	5942 B	/* TTY task: kernel dumping routines */
dosfile.c	12278 B	/* DOSFILE task: 'DOS file as disk' driver */
dp8390.c	48197 B	/* DP8390 task: device independent part */
dp8390.h	12043 B	/* DP8390 task: device independent part */
driver.c	10465 B	/* device independent driver interface */
driver.h	2043 B	/* device independent types and constants */
drvlib.c	6408 B	/* IBM device driver functions and utilities */
drvlib.h	816 B	/* IBM device driver definitions */
esdi_wini.c	21807 B	/* ESDI_WINI task: IBM PS/2 ESDI adapter */
fatfile.c	18868 B	/* FATFILE task: handles FAT files */
floppy.c	39973 B	/* FLOPPY task: NEC PD765 chip controller */
keyboard.c	15017 B	/* TTY task: PC and AT keyboard driver */
memory.c	9509 B	/* MEMORY task */
ne2000.c	7361 B	/* DP8390 task: Novell NE1000/ NE2000 */
ne2000.h	625 B	/* DP8390 task: Novell NE1000/ NE2000 */
pci_amd.h	547 B	/* PCI constants for AMD compatible */
pci_intel.h	1906 B	/* PCI constants for Intel PII compatible */
pci_sis.h	459 B	/* PCI constants for SIS compatible */
pci_table.c	8413 B	/* PCI tables with vendor and device ids */
pci_via.h	945 B	/* PCI constants for VIA compatible */
pci.c	28796 B	/* PCI support routines */

pci.h	2261 B	/* PCI constants and types */
printer.c	13006 B	/* PRINTER task: Centronics driver */
pty.c	12379 B	/* TTY task: Pseudo terminals */
rs232.c	28806 B	/* TTY task: RS232 serial driver */
rtl8029.c	2369 B	/* DP8390 task: Realtek 8029 driver */
rtl8139.c	59176 B	/* RTL8139 task: Realtek 8139 (requires PCI) */
rtl8139.h	19849 B	/* RTL8139 task: Realtek 8139 (requires PCI) */
sb16_dsp.c	18469 B	/* SB16_DSP task: Digital Sound Processor */
sb16_mixer.c	11951 B	/* SB16_MIX task: Mixer */
sb16.h	6153 B	/* SoundBlaster 16: shared header file */
tty.c	51055 B	/* TTY task: device independent part */
tty.h	4594 B	/* TTY task: device independent part */
wdeth.c	9825 B	/* DP8390 task: Western Digital WD80x3 */
wdeth.h	3294 B	/* DP8390 task: Western Digital WD80x3 */
xt_wini.c	27749 B	/* XT_WINI task: WD WX-2 controller */

A.2 Organization of the new source tree

The revision of MINIX resulted in the following source tree. Note that only directories where major revisions took place are listed.

Kernel

The files in this directory are part of MINIX' kernel. Compared to MINIX 2.0.4 the number of files is strongly reduced, because all device drivers have been removed from the kernel. The implementation of the system call handlers of the SYS and CLOCK task was moved to the directories *src/kernel/system/* and *src/kernel/clock/*, and is now compiled into two libraries that are linked with the kernel.

Device drivers

All user-space device drivers are contained in the directory *src/drivers/*. There are subdirectories for the PRINTER (Centronics compatible printers), MEMORY (*/dev/null*, */dev/mem*, */dev/kmem*, and */dev/ram* devices), FLOPPY (floppy disk drives), AT_WINI (AT Winchester hard disks), and TTY (terminal) device driver. These drivers were previously part of the kernel. The directory *drivers/libdriver/* contains shared code for block device drivers.

Servers

All user-space servers at contained in the directory *src/servers/*. There is a new information server (IS) which is discussed in Section 4.1. In short, it is responsible for the debugging dumps that were previously done in the TTY driver. The memory manager (MM), file system (FS) and Internet server (INET) were moved to this new location.

System libraries

All libraries are contained in the base directory *src/lib/*. The user-space implementation of the system library handled by the SYS task is contained in *src/lib/syslib/*. Numerous new system calls were added to support the user-space device drivers. Similarly, a new library was setup for system calls directed to the CLOCK task. In addition, two new libraries to support servers and device drivers were created. More information can be found in the respective header files: <*minix/extralib.h*> and <*timers.h*>. More details on the library for generic timer management can be found in Section 4.2. Appendix B discusses how to add a new system library.

Include files

Many existing header files were updated with new or changed constants and definitions. Especially <*minix/com.h*> was completely reorganized. This files now contains all definitions of process number and message types. Several new header files were created as well, for example, <*timers*> for the new generic timer management library.

SOURCE TREE	PROCESS	:	DESCRIPTION
/usr/src/			
+ ***kernel/***			
- *system/*	SYSTEM	:	system call handlers
- *clock/*	CLOCK	:	system call handlers
+ ***drivers/***			
- *printer/*	PRINTER	:	Centronics driver
- *memory/*	MEMORY	:	memory device driver
- *at_wini/*	AT_WINI	:	hard disk driver
- *floppy/*	FLOPPY	:	floppy disk driver
- *tty/*	TTY	:	Terminal driver
+ ***servers/***			
- *is/*	IS	:	information server
- *mm/*	MM	:	memory manager
- *fs/*	FS	:	file system
- *inet/*	INET	:	network server

Figure A.1: This figure shows the location of all system services in the source tree of MINIX 3.0.0. The most important change compared to MINIX 2.0.4 is that device drivers and servers have been moved to separate base directories.

Appendix B

How to apply changes ...

This appendix discusses how MINIX can be enhanced with new elements. The discussion includes the changes that were most frequently encountered during this project. Section B.1 discusses how new system services, such as user-space device drivers, can be added to the boot image. Section B.2 shows what is needed to create a new system call. Finally, Section B.3 treats updates to MINIX' system libraries.

B.1 Adding programs to the system image

All system services that are part of the system image are automatically started at boot time. The image of MINIX 2.0.4, for example, includes the MM and FS servers. The removal of device drivers from the kernel tasks yielded new user-space device drivers that were included in the system image as well. The steps below describe how to add a program, named X, to the system image.[1]

Changes to MINIX' source code. These changes concern MINIX source code. It is assumed that the new program to be included in the system image is available.

- If you want to be able to include and exclude the program from the system image, add a definition `ENABLE_X` in <*minix/config.h*>. It is required to use one of the values `0` and `1` to indicate whether to program is part of the system image.
- Define a process number for the new program in <*minix/com.h*>. This must be done so that is does not conflict with other servers and device drivers. Note that the MM and FS have fixed process numbers.

[1]Note that the number of programs in the system image is limited to 16. This has to do with amount of memory that is reserved for the boot monitor at the beginning of the kernel's data segment. See Subsection 2.2.1 for more details.

- Update the IMAGE_SIZE variable so that the kernel knows about the number of processes in the system image. If you created an ENABLE_X definition add + ENABLE_X, otherwise add + 1.
- Define a new send mask X_SENDMASK in *src/kernel/sendmask.h*. The kernel uses this bit mask to check if communication with other processes is permitted. Also update the send masks of existing processes that must communicate with the new program.
- The image table in *src/kernel/table.c* must be updated to include the new program. Among other things, this table includes the type of program, its send mask, and a name for the process table.

Changes to the tools. The *Makefile* in *src/tools* is used to build the system image. It must be updated to include the new program. This assumes that ENABLE_X in *src/minix/config.h* is set to 1 if the program is optional.

- Add the new server to the PROGRAMS variable. This ensures that is it included in the system image. It is important that the order of the programs must corresponds to the order using in the image table in *src/kernel/table.c*.[2]
- The programs make target must be updated to include a make rule for the new program. The exact order is not of importance here. Note that this step implies that a *Makefile* is available for the new program.

Changes to the file system. The file system is responsible for the mapping between major device numbers and device drivers. If the new program is a device driver it must be mapped onto a device to make it effective.

- The table with all device-driver mappings is found in *src/fs/dmap.c*. Update the dmap table to map a device onto the new device driver.

B.2 Adding system calls

This section discusses how to define new system calls to request kernel services from the CLOCK or SYS task. This is different from user system calls, such as `'alarm(2)'`, which are handled by either the MM or FS. The steps to add a new system call are rather simple.

- Define a new, unique SYS_CALL message type and all needed parameters in <*minix/com.h*>. In the rare case that none of the current message types can accommodate all parameters, a new message type may be defined in <*minix/type.h*>.

[2]The order of the programs in the system image must be known by the kernel so that it can look up the correct details for each program from the image table.

- Implement a handler function for the system call in the CLOCK or SYS task. This should be done in a separate file in the directory *src/kernel/clock* or *src/kernel/system*. See Section A.2 for more details.
- Update the table with system call-handler function mappings so that the new SYS_CALL message type is recognized and dispatches to the function that handles it. If no errors were made the call should work now.
- Because each system call requires building a request message, sending it to the kernel, and awaiting the response, it is convenient to define a new function in one of the system libraries to do this. See Section C.1 for more details on sytem call organization.

Numerous system calls were added to support user-space servers and device drivers. Appendix C provides an overview of the calls that are new since MINIX 2.0.4.

B.3 Adding system libraries

This section discusses how to add a new system library, *mylib*, to MINIX.

- Place the header file that defines the types and function prototypes for the library in *src/include/* or a subdirectory thereof. The precise directory depends on the kind of library it concerns.
- Create a new directory in *src/lib/*, for example, *src/lib/mylib/* for the library's implementation. By convention, use one file per function contained in the library.
- Place a new make file in your library's directory to build it. This is most easily done by copying a *Makefile* from one of the existing libraries and adapting it to your needs. The library's target name must start with *lib* and must be placed in the directory *src/lib*, for example, *src/lib/libmylib.a*
- Finally, update the master make file, *src/lib/Makefile*, to include the new library. You must add the library to the make targets all, install-i86, and install-i386, so that it is installed in the correct directory after issuing a '`make install`'.

Once the new library has been installed, it can be linked with an application or the system image. To link against the new *libmylib.a* library, for example, one must append '`-lmylib`' to the compile command.

Appendix C

MINIX' system calls

In general, system calls allow system processes to request kernel services, for example, to perform for privileged operations. Section 3.1 already discussed some system calls that were implemented to support the new user-space device drivers and servers. This appendix provides a complete overview of MINIX' system calls and shortly discusses their organization.

C.1 Organization of system call implementation

A system call means that a request is sent to a kernel where it is handled by one of the kernel tasks. The details of assembling a request message, sending it to the kernel, and awaiting the response are conveniently hidden in the system libraries. The header files of the libraries are <*minix/syslib.h*> and <*minix/clocklib.h*>. The implementation of the libraries can be found in *src/lib/syslib* and *src/lib/clocklib*, respectively.

The actual implementation of the system calls is part of the kernel. All calls are directed to either the CLOCK task or the SYS task; in what follows, TASK can be substituted by one them. Suppose that program makes a task_call() system call. By convention, this call is transformed into a request message with type TASK_CALL that is sent to the kernel task TASK.

The TASK handles the request in a function named do_call() and returns the result. The function prototypes of all the handler functions are declared in *src/kernel/task.h*. The actual implementation is contained in separate files in the directory *src/kernel/task/*. The files are compiled into a library *src/kernel/task/task.a* that is linked with the kernel.

The system call message types and their request and response parameters are defined in <*minix/com.h*>. Unfortunately, MINIX 2.0.4 does not follow a strict naming scheme. Therefore, numerous message types and parameters have been renamed in MINIX 3.0.0. System calls to the SYSTEM or CLOCK task start with SYS_ and CLK_, respectively. Furthermore, all parameters that belong to the same system call now share a common prefix.

C.2 Overview of system calls in MINIX 3.0.0

This section gives an overview of the system calls in MINIX 3.0.0. A concise overview is given in Figure C.1. The last two columns show the contribution of this master's project and indicate the future status of each system call.

System call	**Purpose**	**Status**	
PROCESS MANAGEMENT			
SYS_EXEC	Execute a process (initialize process)	-	T
SYS_EXIT	Exit system service (clean up process)	N	D
SYS_FORK	Fork a process (create new process)	-	T
SYS_KILL	Kill a process (send a signal)	-	T
SYS_NEWMAP	Install new or updated memory map	-	T
SYS_XIT	Exit a user process (clean up process)	-	T
SYS_SIGCTL	Signal handling (get and process it)	U	P
SYS_TRACE	Tracing (control process execution)	U	P
COPYING DATA			
SYS_COPY	General copying (virtual and physical)	-	D
SYS_PHYSCOPY	Physical copying (arbitrary memory)	-	D
SYS_VCOPY	General copying (vector with requests)	-	D
SYS_VIRCOPY	Virtual copying (local, remote, BIOS)	N	P
DEVICE I/O			
SYS_DEVIO	Read or write a single device register	N	T
SYS_SDEVIO	Input or output an entire data buffer	N	T
SYS_VDEVIO	Process a vector with multiple requests	N	T
SERVER CONTROL			
SYS_IOPENABLE	Set CPU's I/O privilege level bits	N	T
SYS_KMALLOC	Allocate memory for RAM disk	N	T
SYS_PHYS2SEG	Add segment descriptor in LDT	N	T
SYS_SVRCTL	System control (manipulate server)	U	T
SYSTEM CONTROL			
SYS_ABORT	Abort MINIX (shutdown the system)	U	P
SYS_GETINFO	Get system information (copy data)	N	P
SYS_IRQCTL	Interrupt control (toggle, set policy)	N	P
CLOCK FUNCTIONALITY			
CLK_GETUPTM	Get uptime (since MINIX was boot)	-	D
CLK_SETALARM	Set alarm (signal, message, flag)	U	P
CLK_TIMES	Get times (uptime and CPU usage)	U	P
CLK_TMSWITCH	Measure context switch overhead	N	D

Figure C.1: This figure provides an overview of the system calls in MINIX 3.0.0. The last two columns show the contribution of this master's project and future status of each call. The legenda is as follows: New or Updated since MINIX 2.0.4; Permanent, Temporary or Deprecated.

Some system calls have a temporary character and are likely to be modified or removed by future work. Therefore, an attempt is made to provide to future status of each call. A call is either permanent, temporary or deprecated. Permanent system calls are not expected to change in future versions of MINIX.

Temporary system calls are likely to change in the near future. For example, all process management system call will be unified in a new system call SYS_PROCTL—similar to what was done for SYS_SIGCTL. Furthermore, there are several system calls that concern allocating resources or obtaining additional privileges. These calls will be combined with the SYS_SVRCTL system call to form a unified interface for system control.

A number of MINIX 2.0.4 system calls has been deprecated in MINIX 3.0.0, but are still in place. This, for example, is true for the SYS_COPY system call that has been replaced by SYS_VIRCOPY. The calls could not yet be removed because they still have some temporary uses. The calls that are deprecated according to Figure C.1 will be removed.

Overview of all system calls in alphabetical order

A detailed overview of MINIX' system calls is provided below. For each system call the message type, the purpose, message type, request and/ or response parameters, and return value are specified. Additional remarks about the future status of the call also may be provided. Please note, however, that this project is ongoing work and that *all* system calls may be subject to change.

SYS_ABORT: Shutdown MINIX. This is used by MM, FS and TTY. Normal aborts usually are initiated by the user, for example, by means of the `'shutdown'` command or typing a `'Ctrl-Alt-Del'`. MINIX will also be taken down if a fatal error occurs in the MM or FS.

request parameters

ABRT_HOW: How to abort. One of the values defined in <*unistd.h*>.

- RBT_HALT: Halt MINIX and return to the boot monitor.
- RBT_REBOOT: Reboot MINIX.
- RBT_PANIC: A kernel panic occurred.
- RBT_MONITOR: Run the specified code at the boot monitor.
- RBT_RESET: Hard reset the system.

ABRT_MON_PROC: Process to get the boot monitor parameters from.

ABRT_MON_LEN: Length of the boot monitor parameters.

ABRT_MON_ADDR: Virtual address of the parameters.

return type

OK: The shutdown sequence was started.

SYS_COPY: A copy function to copy data using either physical or virtual addressing. Virtual address are in text, stack or data segment, or ABS to indicate a physical address.

request parameters

CP_SRC_SPACE: Source segment.

CP_SRC_BUFFER: Virtual source address

CP_SRC_PROC_NR: Process number of the source process.

CP_DST_SPACE: Destination segment.

CP_DST_BUFFER: Virtual destination address

CP_DST_PROC_NR: Process number of the destination process.

CP_NR_BYTES: Number of bytes to copy.

return type

OK: The copying was done.

EFAULT: Virtual to physical mapping failed.

remarks

This call is deprecated. It has been replaced by SYS_VIRCOPY.

SYS_DEVIO: Perform device I/O on behalf of a user-space device driver. The driver can request a single port to be read or written with this call. Also see SYS_SDEVIO and SYS_VDEVIO.

request parameters

DIO_REQUEST: Input or output.

- DIO_INPUT: Read a value from DIO_PORT.
- DIO_OUTPUT: Write DIO_VALUE to DIO_PORT.

DIO_TYPE: A flag indicating the type of values.

- DIO_BYTE: Byte type.
- DIO_WORD: Word type.
- DIO_LONG: Long type.

DIO_PORT: The port to be read or written.

DIO_VALUE: Value to write to the given port. For DIO_OUTPUT only.

response parameters

DIO_VALUE: Value that was read from the given port. For DIO_INPUT only.

return type

OK: The port I/O was successfully done.

EINVAL: An invalid DIO_REQUEST or DIO_TYPE was provided.

remarks

All device I/O calls will be unified in a single SYS_DEVIO call.

SYS_EXEC: A process has successfully executed a program. The FS has copied the binary image into memory and the MM requests the kernel patch up the process' registers for execution.

request parameters

PR_PROC_NR: Process that executed a program.

PR_TRACING: Flag to indicate whether tracing is enabled.

PR_STACK_PTR: New stack pointer.

PR_IP_PTR: New program counter.

PR_NAME_PTR: Pointer to name of program.

return type

OK: This call always succeeds.

remarks

This call will be combined with other process control calls. A new SYS_PROCTL will be created for this. Proper error handling must be added.

SYS_EXIT: A system process wants to exit. Clean up its process slot. Note that this call is different from the SYS_XIT call that is used by the MM to announce that a regular user process has exited.

request parameters

EXIT_STATUS: Zero on a normal exit. Non-zero if an error occurred.

return type

This call never returns.

remarks

This call will be removed when system services become ordinary user processes.

SYS_FORK: A process has forked. The MM has found an available process slot in its own process table and now requests the kernel to allocate the associated process slot in the kernel's process table.

request parameters

PR_PROC_NR: Child's process table slot.

PR_PPROC_NR: Parent, the process that forked.

return type

OK: This call always succeeds.

remarks

This call will be combined with other process control calls. A new SYS_PROCTL will be created for this. Proper error handling must be added.

SYS_GETINFO: Obtain a copy of all kinds of system information. This call supports user-space device drivers and servers that need certain system information. Furthermore it is used by the IS server to request entire data structures for debugging dumps. Note that a new message type, mess_7, was declared in <*minix/type.h*> to accommodate all needed parameters.

request parameters

I_REQUEST: The type of system information that is requested.

- GET_KENVIRON: Get the system environment as known by the kernel.
- GET_KADDRESSES: Get the physical addresses of kernel variables.
- GET_TASKTAB: Copy the tasktab table defined in *src/kernel/table.c*.
- GET_MEMCHUNKS: Copy the mem array with free memory chunks.

- GET_PROCNR: Retrieve the process number for a given process name.
- GET_PROCTAB: Copy the proc table defined in *src/kernel/proc.h*.
- GET_MONPARAMS: Get a copy of the boot monitor parameters.
- GET_KENV: Get a single parameter. Key provided by the caller.
- GET_SCHEDINFO: Retrieve the current scheduling queues.

I_PROC_NR: Process where the information should be copied to.[1]

I_VAL_PTR: Virtual address where the information should be copied to.

I_VAL_LEN: Maximum length that the caller can handle.

I_KEY_PTR: Virtual address of the key provided by the caller.

return type

OK: The information request succeeded.

EFAULT: An illegal memory address was detected.

EINVAL: Invalid request or process number, or key is too large.

ESRCH: The requested kernel environment string was not found.

E2BIG: Requested data exceeds the maximum provided by the caller.

SYS_IOPENABLE: Request the I/O Protection Level bits of the given process to be enabled. This gives user-space processes privileges to perform device I/O. This may endanger the system, so it only works if it is explicitly allowed by the current configuration. To make the call effective, the definitions ENABLE_USERPRIV and ENABLE_USERIOPL in <*minix/config.h*> must both be set to 1.

request parameters

PROC_NR: The process to give I/O Protection Level bits.

return type

OK: The I/O Protection Level bits were successfully set.

EPERM: The current configuration does not allow the call.

remarks

This call will be unified with the SYS_SVRCTL call.

SYS_IRQCTL: Interrupt control. This call allows user-space device drivers to enable or disable interrupts and to install a policy for the kernel's generic interrupt handler.

request parameters

IRQ_REQUEST: Interrupt control request to perform.

- IRQ_ENABLE: Enable IRQs for the given IRQ line.
- IRQ_DISABLE: Disable IRQs for the given IRQ line.
- IRQ_SETPOLICY: Set interrupt policy for the generic interrupt handler.

IRQ_VECTOR: IRQ line that must be controlled.

IRQ_POLICY: Bit map with flags indicating IRQ policy.

IRQ_PROC_NR: Policy: indicates process to be notified about hardware interrupts.

IRQ_PORT: IRQ Policy: indicates port to read from or write to.

IRQ_VIR_ADDR: Policy: virtual address at caller to store value read from port.

IRQ_MASK_VAL: Policy: Mask to strobe back value to port or value to write to port.

[1] Usually this is the calling process, but the MM may request system information to be copied on behalf of another process for backward compatability. When the MM detects the no longer supported SYSGETENV server control request, it is transformed into a SYS_GETINFO request that does the job.

return type

EINVAL: Invalid request, IRQ line or process number.

EFAULT: Invalid virtual address at caller.

EPERM: Only owner can remove its IRQ policy.

ENOSYS: Removal of IRQ policy is not yet supported.

EBUSY: Each IRQ vector can only have a single policy at this moment.

remarks

IRQ policies may be subject to change when more device driver are moved to user-space. Multiple IRQ policies should be available per IRQ line. The call should be simplified by copying IRQ policy structure from caller, instead of overloading the message with all values.

SYS_KILL: A system service sends wants to signal a process. All signals are forwarded to the kernel to prevent being blocked if the MM is not ready. The kernel notifies the MM about the pending signal for further processing.

request parameters

SIG_PROC_NR: Process to be signaled.

SIG_NUMBER: Signal number.

return type

OK: Always succeeds.

remarks

This call will be unified with the SYS_SIGCTL call.

SYS_KMALLOC: Request a (DMA) buffer to be allocated in one of the free memory chunks. This call is only used by the MEMORY driver to allocate a RAM disk before the MM grabs all memory.

request parameters

MEM_CHUNK_SIZE: Size of the requested buffer in bytes.

response parameters

MEM_CHUNK_BASE: The physical address of the start of the allocated buffer.

return type

OK: The buffer was successfully allocated.

ENOMEM: No memory chunk was big enough to hold the buffer.

remarks

This call will be unified with the SYS_SVRCTL call.

SYS_NEWMAP: A process gets a new memory map, either because it was just forked or because its map was updated. Fetch the memory map from MM.

request parameters

PR_PROC_NR: Install new map for this process.

PR_MEM_PTR: Pointer to memory map at MM.

return type

OK: New map was successfully installed.

EFAULT: Incorrect address for new memory map.

EINVAL: Invalid process number.

remarks

This call will be combined with other process control calls. A new SYS_PROCTL will be created for this.

SYS_PHYS2SEG: Add a segment descriptor to the LDT and return a selector and offset that can be used to reach a physical address. This is meant for device drivers doing memory I/O in the A0000 - DFFFF range. Currently the call is only used by the TTY driver to access video memory. For large segments, where 4K granularity is required instead of 1K, ENABLE_LOOSELDT in <*minix/config.h*> should be enabled.

request parameters

SEG_PHYS: Physical base address of segment.

SEG_SIZE: Size of segment.

response parameters

SEG_SELECT: Segment selector for LDT entry.

SEG_OFFSET: Offset within segment. Zero, unless 4K granularity is used.

return type

OK: Segment descriptor successfully added.

E2BIG: If size of segment is too large and ENABLE_LOOSELDT is not enabled.

remarks

This call will be unified with the SYS_SVRCTL call.

SYS_PHYSCOPY: Copy data from anywhere to anywhere in the memory. This copy call uses physical addressing. The SYS_VIRCOPY system call should be used instead whenever possible.

request parameters

CP_SRC_BUFFER: Physical source address.

CP_DST_BUFFER: Physical destination address.

CP_NR_BYTES: Number of bytes to copy.

return type

OK: The copying was done.

EFAULT: Either the source or destination address was zero.

remarks

This call is deprecated. It has been replaced by SYS_VIRCOPY.

SYS_SDEVIO: Perform device I/O on behalf of a user-space device driver. The driver can request a input or output of an entire buffer. Also see SYS_DEVIO and SYS_VDEVIO.

request parameters

DIO_REQUEST: Input or output.

- DIO_INPUT: Read a value from DIO_PORT.
- DIO_OUTPUT: Write DIO_VALUE to DIO_PORT.

DIO_TYPE: A flag indicating the type of values.

- DIO_BYTE: Byte type.
- DIO_WORD: Word type.
- DIO_LONG: Long type.

DIO_PORT: The port to be read or written.

DIO_VEC_PROC: Process where buffer resides.

DIO_VEC_ADDR: Virtual address of buffer.

DIO_VEC_SIZE: Number of elements to input or output.

response parameters

DIO_VALUE: Value that was read from the given port. For DIO_INPUT only.

return type

OK: The port I/O was successfully done.

EINVAL: Invalid process number, request, or granularity.

EFAULT: Invalid virtual address of buffer.

remarks

All device I/O calls will be unified in a single SYS_DEVIO call.

SYS_SIGCTL: Signal handling. When the kernel notifies the MM about pending kernel signals, the MM calls back to get the outstanding signals and process them.

request parameters

SIG_REQUEST:

- S_GETSIG: See if there are pending kernel signals.
- S_ENDSIG: Finish up after a KSIG-type signal.
- S_SENDSIG: POSIX-style signal handling.
- S_SIGRETURN: Return from POSIX-style signal.

SIG_PROC: Indicates the process that was signaled.

SIG_CTXT_PTR: Pointer to context structure for POSIX-style signal handling.

response parameters

SIG_PROC: Return next process with pending signals or NONE.

return type

OK: Signal handling action successfully performed.

EPERM: Only the MM is allowed to request the signal control operations.

EINVAL: Invalid SIG_REQUEST, SIG_PROC or SIG_CTXT_PTR provided.

EFAULT: Invalid context structure address, or could not copy signal frame.

SYS_SVRCTL: This system call allowes to dynamically load a server by giving it extra privileges. It currently is only used by the INET server.

request parameters

SVR_REQUEST: Server control operation that is requested.

- SYSSIGNON: Sign on as a new server.
- SYSSENDMASK: Only set a new send mask.

SVR_PROC_NR: Process number of the caller.

SVR_MM_PRIV: Process privileges as soon by the MM.

return type

OK: The calls succeeded.

EPERM: Permission was denied because the process is not running as super user.

EINVAL: Invalid process number or the request type was not supported.

remarks

This system call will be extended to provide both better support and security checks for servers or device drivers that must be dynamically loaded. This is part of future research.

SYS_TRACE: Observe and control processes. Handle the debugging commands supported by the ptrace() system call.

request parameters

CTL_REQUEST: The tracing request.

- T_STOP: Stop the process.
- T_GETINS: Return value from instruction space.
- T_GETDATA: Return value from data space.
- T_GETUSER: Return value from user process table.
- T_SETINS: Set value from instruction space.
- T_SETDATA: Set value from data space.
- T_SETUSER: Set value in user process table.
- T_RESUME: Resume execution.
- T_STEP: Set trace bit.

CTL_PROC_NR: The process number that is being traced.

CTL_ADDRESS: Virtual address in the traced process' space.

CTL_DATA: Data to be written.

response parameters

CTL_DATA: Data be returned.

return type

OK: Always succeeds.

EIO: Set or get value failed.

SYS_VCOPY: Copy multiple blocks of memory from one process to another. A request vector is fetched from the caller. Virtual addressing is used.

request parameters

VCP_VEC_SIZE: Number of elements in request vector.

VCP_VEC_ADDR: Virtual address of request vector at caller.

VCP_SRC_PROC: Source process.

VCP_DST_PROC: Destination process.

return type

OK: The copying was done.

EDOM: Request vector too large.

EFAULT: Virtual to physical mapping failed.

remarks

This call is will be unified with SYS_VIRCOPY.

SYS_VDEVIO: Perform a series of device I/O on behalf of a user process. The call accepts a pointer to an array of (port,value)-pairs that is to be handeld at once. Hardware interrupts are temporarily disabled to prevented the bactch of I/O calls to be interrupted. Also see SYS_DEVIO and SYS_SDEVIO.

request parameters

DIO_REQUEST: Input or output.

- DIO_INPUT: Read a value from DIO_PORT.
- DIO_OUTPUT: Write DIO_VALUE to DIO_PORT.

DIO_TYPE: A flag indicating the type of values.

- DIO_BYTE: Byte type.
- DIO_WORD: Word type.
- DIO_LONG: Long type.

DIO_VEC_SIZE: The number of ports to be handled.

DIO_VEC_ADDR: Virtual address of the (port,value)-pairs in the caller's space.

return type

OK: The port I/O was successfully done.

EINVAL: Invalid request, granularity, or vector size.

EFAULT: The address of the (port,value)-pairs is erroneous.

remarks

All device I/O calls will be unified in a single SYS_DEVIO call.

SYS_VIRCOPY: A copy function to copy data using virtual addressing. The virtual can be in three segments: LOCAL_SEG (text, stack, data segments), REMOTE_SEG (e.g., RAM disk, video memory), and the BIOS_SEG (BIOS I/O). This is the most common system call relating to copying.

request parameters

CP_SRC_SPACE: Source segment.

CP_SRC_BUFFER: Virtual source address

CP_SRC_PROC_NR: Process number of the source process.

CP_DST_SPACE: Destination segment.

CP_DST_BUFFER: Virtual destination address

CP_DST_PROC_NR: Process number of the destination process.

CP_NR_BYTES: Number of bytes to copy.

return type

OK: The copying was done.

EDOM: Invalid copy count (CP_NR_BYTES ¡ 0).

EFAULT: Virtual to physical mapping failed.

EINVAL: Incorrect segment type or process number.

EPERM: Only owner of REMOTE_SEG can copy to or from it.

SYS_XIT: A user process has exited. The MM sent a request to clean up the process table slot and to accumulate the child times at the parent process.

request parameters

PR_PROC_NR: Slot number of exiting process.

PR_PPROC_NR: Slot number of parent process.

return type

OK: The cleanup succeeded.

EINVAL: Incorrect process number.

remarks

This call will be combined with other process control calls. A new SYS_PROCTL will be created for this.

CLK_GETUPTM: Get the uptime since MINIX was boot.

response parameters

T_BOOT_TICKS: Number of ticks since MINIX boot.

return type

OK: Always succeeds.

remarks

This call is deprecated; CLK_TIMES provides the same functionality.

CLK_SETALARM : Set or reset an alarm. This system call provides a single interface to set different types of alarms.

request parameters

ALRM_TYPE: Action to be taken when the alarm goes off.

- CLK_SIGNALRM: send a SIG_ALRM signal.
- CLK_FLAGALRM: set a timeout flag to 1.
- CLK_SYNCALRM: send a SYN_ALARM notification.

ALRM_PROC_NR: Process that must be alerted when the alarm expires.

ALRM_EXP_TIME: Absolute or relative expiration time for this alarm.

ALRM_ABS_TIME: Zero if expire time is relative to the current uptime.

ALRM_FLAG_PTR: The virtual address of the timeout flag for CLK_FLAGALRM.

response parameters

ALRM_SEC_LEFT: The number of seconds left on the previous alarm is returned here.

return type

OK: The alarm was successfully set.

EINVAL: The alarm type or requesting process number was incorrect.

EFAULT: The address of the timeout flag was erroneous.

CLK_TIMES: Get all time information for a given process.

request parameters

T_PROC_NR: The process to get the time information for.

response parameters

T_USER_TIME: User time in ticks.

T_SYSTEM_TIME: System time in ticks.

T_CHILD_UTIME: Cumulative user time of children.

T_CHILD_STIME: Cumulative sys time of children

T_BOOT_TICKS: Number of ticks since MINIX boot.

return type

OK: Always succeeds.

CLK_TMSWITCH: Measure context switch overhead as described in Section 2.1. This system call is only meant to initiate the test sequence. The tests must be done within the kernel because microsecond precision timing is needed.

return type

OK: Always succeeds.