Internal Report 09-13 August 2009

Universiteit Leiden

Opleiding Informatica

An Implementation of

Intelligent Disk Block Replication in Minix

Kristian Rietveld

MASTER'S THESIS

Leiden Institute of Advanced Computer Science (LIACS)
Leiden University
Niels Bohrweg 1
2333 CA Leiden
The Netherlands

Abstract

With ever increasing speeds of central processing units and main memory, the speed of disks lags behind. Reading from disks such as hard disks, floppy disks and DVDs is orders of magnitude slower than the computer's main memory. Since the capacity of the main memory is limited and we cannot do without disks for large sets of data, there is a real need to optimize reading from disks as much as possible. One way to do this is by optimizing according to the usage of the computer system by the user.

We describe a way to optimize accessing disks by monitoring operations that are done on the disk thereby discovering patterns in these operations. By exploiting the fact that reading blocks that are stored on the disk in a contiguous sequence is faster than reading blocks that are scattered over the disk, we will store the blocks that make up these patterns in a contiguous sequence at another place on the disk. We will show that such replication of related disk blocks indeed leads to improvements in disk read performance in certain situations.

We have modified the Minix operating system kernel to be able to discover such patterns and support the replication of blocks on its native file system. A concise overview of the full Minix kernel and our modifications to it are given.

Voorwoord

Al sinds het begin van mijn studie Informatica ben ik uitgebreid geïnteresseerd geweest in besturingssystemen. Ook na het vak "Operating Systems" bleef de hebzucht bestaan om meer te leren over deze fundamenten van elk computersysteem. Onder begeleiding van André Deutz ben ik initieel begonnen met het bestuderen van de Linux broncode. Dat bleek te groot en complex te zijn om ook maar gedeeltelijk te doorgronden. Daarop zijn we verder gegaan met het compleet doorgronden van de Minix broncode. Van het begrijpen van een compleet besturingssysteem heb ik vele nieuwe inzichten in deze systemen gekregen.

Als eerste zou ik André Deutz willen bedanken voor zijn begeleiding tijdens dit project. Met hem heb ik vele interessante discussies gevoerd. Het idee om met Minix aan de slag te gaan en het daarop volgende project zijn van zijn hand. Daarnaast wil ik Harry Wijshoff bedanken voor het overwegen van deze scriptie als afstudeerscriptie.

Achter dit werk zit een proces van meerdere jaren. Zonder steun van vele andere mensen was dit werk waarschijnlijk nooit afgekomen. Ik zou de volgende mensen graag willen danken voor hun steun, motivatie, interesse en afleiding in de laatste jaren.

Thijs, Wouter en Pieter voor de vele, interessante, vaak technisch getinte discussies in de kroeg tijdens de weekenden.

Vian, Ron en anderen uit de FooBar, voor interesse en vooral mentale steun. Dankzij hun kreeg ik vaak weer extra motivatie om door te gaan met mijn studie en deze scriptie af te ronden.

Mijn huidige bazen, Mikael Hallendal en Richard Hult, voor het geven van een kans om voor hun te werken. Daarbij voor alle hulp en flexibiliteit om naast mijn werk mijn studie af te kunnen maken.

Mijn ouders, vanzelfsprekend, voor alle steun tijdens het studeren. Op hun kan ik altijd rekenen en terugvallen.

Mijn vriendin Ginny, zij heeft mij als geen ander weten te inspireren en motiveren om mijn studie en in het bijzonder deze scriptie tot een goed einde te brengen. Een groot gedeelte van deze scriptie is geschreven bij haar in Barcelona, weg uit het drukke en soms chaotische Nederland. Haar liefde, steun en interesse zijn ongeëvenaard. Zonder haar inspiratie had het maanden langer geduurd om dit werk tot een einde te brengen.

Kristian Rietveld
September, 2008

Contents

List of Figures

1 Introduction

In computer systems an operating system is at least as important as the Central Processing Unit (CPU). Of course, you are able to run simple programs on a computer system without making use of an operating system, but you will quickly become frustrated with writing tedious code for being able to use the computer's keyboard, showing output on the display, etc. One of the main tasks of the operating system is abstracting the hardware away from programmers, who want to get the next pressed key in the keyboard buffer with one library call instead of doing the Input/Output (I/O) operations with the keyboard themselves.

Microprocessors these days are quite ubiquitous and so are operating systems. Operating systems are being developed, or derived from an existing operating system, for various kinds of electronic equipment: DVD players, televisions, cell phones, video game consoles, etc. When designing new electronic devices with a CPU, or embedded systems, thinking about your operating system design is a very important. It determines what kind of tasks the system will be able to perform. When third-party developers are involved to write software for your platform, the operating system provides one of the Application Programming Interfaces (API) to them. The operating system is right in between the applications and the hardware and sometimes seen as part of the system. For system developers it makes a lot of sense to study the internals of operating systems.

We will closely study and comment on the source code of a small operating system, Minix [10], which is very popular among educational institutions. Minix is a full UNIX kernel in less than 50,000 lines of code (we have not included the network stack and most drivers in this count), which makes it very suitable for in-depth studies. In the first chapters we will outline some basics of operating systems and we will see what kinds of techniques Minix is using. This part is meant to be a brief and concise introduction to Minix, without getting carried away too much into details. Because of this, you almost certainly want to study Tanenbaum [8] if you want to have more detailed information. We also assume that the reader is already familiar with UNIX and some basic system calls such as `fork`, `exec`, `open` and `read`.

In the second part of the thesis we will design and implement an extension to the Minix operating system which will monitor disk accesses to discover patterns in frequently accessed disk blocks. We will replicate these disk blocks in groups at other places on the disk so they can be read all at once. This has as goal to improve the disk read performance and we will investigate if this is indeed the case. The second part of this thesis is also split up in two chapters. In chapter 4, we will outline the goal of the research in more detail and present our plan for achieving this. Chapter 5 will discuss the actual implementation of the ideas in the Minix kernel and measurements that have been done to verify whether or not the disk read performance has improved.

Part I
Detailed study of Minix

2 Basics of Operating Systems

At a first glance an operating system might seem like a complex piece of software. In fact, this is not true at all, the task of an operating system is very straightforward: run a given program on the given hardware. However, the techniques used for getting this done differ in complexity; from easy but not always efficient, to complex and more efficient.

In this chapter, we will introduce a number of basics which we will need in the further discussions in this thesis.

2.1 The Kernel

The term "operating system" is quite abstract. Usually people actually mean the kernel when talking about an "operating system". The kernel is the layer between the hardware and user-space software. It provides the programmer of user-space applications with simple interfaces to the hardware. Next to that, kernels also arrange scheduling of different processes to run, virtual memory, memory protection, etc.

Some people actually mean much more than just the kernel when talking about "operating systems". For example, when people are talking about the Windows operating system, they are talking about the kernel, plus the bootloader, C library, UI libraries, graphical shell, etc.

In our discussion we will focus on the kernel, which actually is the "real" operating system as pointed out above. Currently, we distinguish two types of kernels and a mixture of the two which is very popular.

2.1.1 Monolithic kernels

Most traditional UNIX systems, including Linux, have monolithic kernels. In a monolithic kernel all code, including device drivers, is linked together in a single binary which is loaded on startup. Of course, there are exceptions here and there; for example Linux has the possibility to load separately compiled modules into the kernel at runtime, but note that this module is actually "linked" into the kernel so in theory it is still a single binary.

Tanenbaum, the creator of Minix and evangelist of micro kernels, says that the monolithic kernel approach can be subtitled "The Big Mess" in his book [8, pp. 42-43], and claims that these systems have no structure. It is certainly true that "everything can call everything", but there certainly are exceptions to the rule that all monolithic kernels are unstructured. There are no problems creating monolithic systems with well-defined interfaces and rules for using those; when applied in a consistent manner it will be hard to come up with a system that earns the name "The Big Mess".

On system startup, the kernel binary is loaded into memory and started at a predefined function/address. Modern CPUs, with support for multitasking, have several protection levels. These protection levels define what kind of instructions a program can execute and which I/O ports, etc, it is allowed to access. Some programs are allowed to execute CPU instructions to setup memory protection, interrupt handler tables, and some, in a higher protection level, are not. It is clear that the kernel binary thus runs in the lowest protection level and

all user-space programs in one of the higher levels. The details differ per architecture and operating system.

User-space processes can, in general, not directly access the hardware. Therefore the kernel is involved with every communication with the hardware, and there has to be an interface to the kernel for the user process. In monolithic kernels the standard practise is to expose a small set of "functions" to user-space via system calls. For modern UNIX systems this interface is defined in the POSIX standard [12]. System calls usually work like a kind of trap. After setting up the arguments to the call, just like for a regular function call, a special interrupt is called. This interrupt traps into the kernel and because of this mode switch the protection level is now the lowest. The kernel then runs the requested system call after carefully checking the arguments to that call. When done, the context is switched back to the user process. The details of this mechanism again differ between the different architectures and operating systems.

2.1.2 Micro kernels

The most important property of the micro kernel is that the different "parts" or components of the kernel are running in separate processes. Every service a kernel provides is usually extracted in a separate part: think of a component handling file system access, a component accessing the disks and other hardware, a component doing networking, and so on. These processes do not necessarily have to be in kernel-space, in fact most are running in user-space or in a protection level in between user- and kernel-space if the architecture allows. The different parts usually communicate with each other by sending small messages.

Micro kernels are in general said to be more fault-tolerant, since the different parts of the kernel are all memory protected; a bug in one part of the kernel does not necessarily crash the complete operating system. Another advantage is that only the part of the kernel that really has to run in kernel-space runs in the lowest protection level and thus has access to the hardware, all other parts do not. Device drivers can get access to the hardware by sending messages to the "real" kernel, which checks whether the access is allowed and then executes the request. From this discussion it also follows that system calls executed by user-space processes will be transformed into messages to the correct subsystem and handled there subsequently.

2.1.3 A mix between micro and monolithic kernels

While micro kernels actually fix the memory protection problems of monolithic kernels by putting different components in separate processes with their own memory space, it does introduce another problem: performance. When passing messages between different processes and protection levels, costly context and mode switches will show up as an important performance problem [5, 2, 4].

Therefore commercial operating systems based on pure micro kernels rarely exist. Many commercial implementations of micro kernels ended up to be a kernel type in between the monolithic and micro kernels by moving the most important system servers, that usually run in user-mode, into kernel-space. Such operating systems are the Windows NT kernel [11] and the modified Mach kernel of Apple's OS X [9]. Research has been done to improve the performance of micro kernels by either moving these services into kernel-space [2, 4] and by improving the implementation and RPC mechanism of the kernel [5].

2.2 Interfacing with the hardware

2.2.1 Booting up

When a computer is turned on, the CPU starts executing instructions from a specified memory address. At this location one can usually find some kind of firmware (BIOS on x86 machines). This firmware executes some really low level initialization of the hardware. Also the different drives in the system are detected and each of those is probed, looking for a boot sector.

Once found, the control is passed to the boot sector, whose task is to load an operating system kernel (or sometimes even a second stage boot loader with more functionality) into memory and pass over control. The operating system kernel also does a good bunch of initialization of the hardware, setting up the CPU for multitasking and memory protection, initializing the PCI bus and hardware found on that bus, detecting the drives and looking for a root file system to mount, setting up the networking stack, etc, etc.

At the end the different services which are required are started and the user will be presented with a login prompt in some form.

2.2.2 Handling interrupts and exceptions

Once an operating system is running it will spend its time running the different outstanding processes. But how does it switch between processes? How does it handle user input? The answer here is interrupts.

Interrupts play a very important role in modern multitasking operating systems. Most pieces of hardware have the ability to send an "interrupt request" to the CPU. When the CPU has not disabled interrupts, it will receive an interrupt request. Interrupts are usually numbered and the operating system usually sets handlers for the different interrupt numbers; so it knows which interrupt number it received, which can be used to determine which piece of hardware sent the interrupt and how it should be handled.

By keeping in mind what an interrupt is, we can more precisely write down how user input is handled. For example, when you press a key on the keyboard, an interrupt request is sent to the CPU. The CPU then calls the handler for this interrupt that is set by the operating system. This handler knows it has been set up to handle keyboard input and will handle the interrupt request by reading the keyboard buffer.

A common piece of hardware is, for instance, an internal clock. This clock generates timer interrupts, at a specified frequency. It is guaranteed that the CPU is interrupted at this frequency. When interrupted the operating system usually does some left-over work from previous interrupt requests (an interrupt handler cannot "block" things for too long) and then picks another process to run next (depending on the scheduling algorithm used, more about this later on). It is exactly this what enpowers modern multitasking operating systems.

Exceptions are generated by the CPU and are usually handled in the same way as interrupts. Each exception is assigned a number and when an exception occurs a handler is found in a trap table and executed. An exception usually signals a problem found when executing the code, think of problems like division by zero, incorrect opcode, etc. The "page fault", another exception, deserves a special notion: this is what makes virtual memory implementations work. Recall that in many operating systems the memory space of a process is divided into pages with a fixed size. When the internal memory of the machine is full, some pages may be "swapped out" to the disk. When a CPU needs to access a memory page which is currently not in internal memory, it will throw a page fault exception. The operating sys-

tem will load the missing page into physical memory (usually from disk) so the process can continue running. Since Minix does not support paging, we will not discuss it any further.

2.2.3 Protection

Physical memories are often divided in pages and of course the different memory pages belong to several different processes. You do not want that process A can access a memory page from process B. This is why pages are usually "marked" with a kind of permissions you also see on files. When a process tries to access a page which it is not allowed to access, it will generate a page fault exception. The operating system decides what to do, usually the process in question is terminated.

From the above passage it is probably clear that this kind of memory protection needs hardware support. This could be in the form of tables containing this information (also saying which memory is reserved for the kernel and which parts are for user processes). The tables are placed in the main memory, in a specified format, and the address where the tables are located is loaded into the CPU.

Next to protecting memory accesses hardware also has support for protecting certain critical instructions. Modern CPUs have several protection levels, a program running in the lowest level can do everything, programs in higher levels are not allowed to execute certain instructions. As mentioned earlier the kernel has to run in the lowest level in order to be able to set up the interrupt tables, configure the memory protection unit, and so on. If user-space programs were allowed to execute those executions, writing a safe and stable operating system kernel would be very hard.

2.2.4 Context switches

A context switch occurs when we hand over control from one process to another. In general, this is a very costly operation [6, pp. 254-255] as we have to save the state, or context, from a process to the memory (often on the stack) and load the context of the next process to run. What the process's context exactly includes depends on the architecture, but often it includes the program counter, most CPU registers, details on its address spaces, the stack and when dealing with paging also translate tables, etc. The translate tables are often not saved, but just flushed. This means that when the context returns to this process again the translation tables will have to be rebuilt.

The name context switch is used since we are dealing with a change of context here. Other "switches" are also common, such as a "mode switch" when control is transferred to a code segment with a different privilege level and the "stack switch" where only is switched to another stack.

There exist CPUs that have hardware support for doing task switching. In x86 processors the Task-State Segment (TSS) and related logic is used for this. These CPUs are capable of switching processes completely by themselves. Usually this is referred to as "hardware task-switching". However, most modern operating system, including Linux [13] implement the task-switching logic in software and thus do all task switching by themselves. One of the main reasons for operating systems to do this themselves is that the hardware task-switching is "hard-coded" in the CPU and could differ from CPU version to CPU version. In addition to the fact that it is very hard to gracefully recover from a hardware task-switching failure if one of the components involved was in a bad state. Unsurprisingly, the software variant is

named "software task-switching". When doing software task-switching the TSS machinery is still involved, so without code documentation it is often very hard to determine how exactly task switches are handled.

2.3 Process management

2.3.1 Scheduling

Earlier in this chapter we explained that the timer interrupt is basically what makes multitasking work. This interrupt hands the control back to the operating system, that gets the opportunity to do some left-over work. The main question is then: which process to run next? For solving this problem kernels have a "scheduler", which is responsible for determining when to run which process.

There are many different scheduling algorithms, ranging from very easy (for example, run all processes in turn) to very complex. The kind of algorithm which is used has a significant influence on the total performance of the system. Different kinds of systems also require different kinds of scheduling algorithms in order to achieve a reasonable performance. Compare systems processing jobs in batches with desktop systems. For the former, you want the machine to spend as much time on the batch jobs as possible, in order to get the results as soon as possible. For the latter there a lot of different processes running at once; in this case you want to do a lot more context switches so the different processes all get a chance to run and the user is presented with a smoother user experience.

Not only the tasks the system will be used for are important for creating a good scheduling algorithm, also the nature of the processes running on the system play some importance. When you are copying files, you usually want to give more CPU time to that process than to a mail server waiting for a new connection to accept. If you experiment with different algorithms, you will notice differences in the behavior and performance of the system.

It will not come as a surprise that there are a lot of scheduling algorithms around already. We will shortly describe a couple of established scheduling algorithms, for a more complete description we refer to Tanenbaum [8].

An old and widely used algorithm is "round robin" scheduling and will probably look familiar to people who are known with "round robin DNS". The scheduler creates and maintains a list of all processes running in the system. Then it will run each process on that list in turn, for a fixed amount of time (usually known as a "quantum"). Important to note is that each process gets the same amount of CPU time to use, which is, as we discussed above, not always what we want.

So, usually we will want to give certain processes a higher priority and make sure they will be run earlier and/or get more CPU time. This is the basic idea of priority scheduling; the runnable process with the highest priority gets to run first. There are different ways to assign priorities to processes, this can happen on a static or dynamic way, and there are different factors which have to be taken into account. Writing an algorithm to calculate suitable priorities for processes is quite a task in itself.

When using priority scheduling, one should be very wary for avoiding "starvation". Starvation occurs when one process is running on the CPU indefinitely and the other processes do not get a chance to run (they are "starving"). A way to avoid this is for example to decrease the priority of that process after each timer interrupt. At some point the process will not have the highest priority any more and another process will be run.

Note that both algorithms that we described above are algorithms for so-called interactive systems. For batch systems, which process large jobs in batches, there is a separate group of suitable scheduling algorithms. Most modern supercomputers run special software to process outstanding submitted calculation jobs in batches. However, these algorithms fall out of the scope of this thesis.

2.3.2 Managing the process' memory

When you load an executable program into memory, you are actually not loading one large sequence of program code into memory. Next to the actual program code there is much more information in an executable, for example the definitions of the static variables, including their initial values. Often you can also find a symbol list, some general information about the executable (amount of memory required execute it, etc) and, when compiled with the appropriate options, debugging information.

In memory processes are structured in segments and in most common executables you will find text and data segments. In the text segment the sequence of program instructions is stored; in the data segment there is a couple of things, next to the initialized static variables there is space for the stack and the heap. The format of the executables on disk also has separation of data in segments, the general information about the executable file is stored in a "header".

From the above discussion you have probably noted that all the data the program code needs write access to is stored in the data segment. Most operating systems do not allow writing to the text segment.

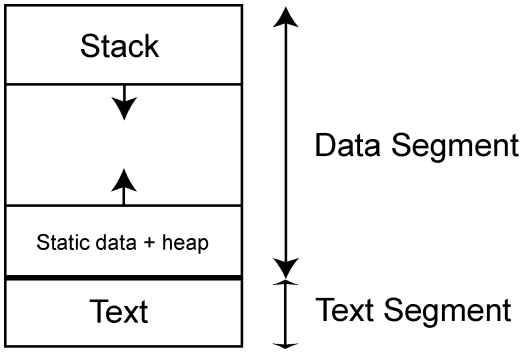

Figure 1: Simplified overview of a process' memory layout

The data segment has multiple functions, the remaining space (all space available for the data segment minus the room needed for the static variables) is used for both the stack and the heap. The heap is used for accommodating the dynamic memory allocations. In general the stack starts "growing" from one side of the data segment and the heap from the other side. This does indeed mean that in some non-common circumstances the stack and heap might collide with each other, causing a good amount of trouble.

2.3.3 System calls

As mentioned earlier the kernel is a layer between the hardware and (user-space) software. In a previous section we have discussed how the kernel interfaces with the hardware. Now for

the kernel to be a layer between hardware and software, there must be a way for processes to communicate with the kernel. If not then there is no way processes can send requests to the hardware. When a process wants to access some piece of hardware, for example read a block from a disk, it does this by sending a request to the kernel. Because data is stored on disk using a specified file system (see section 2.4.1), the request is usually much more abstract and of the form: *"I want to read these bytes from this file"*. The operating system takes care of seeing on which device that file is located, how to access that device, how to read the file system on that device, etc, etc.

The process can request something from the kernel by doing a system call (this can indeed be read as a "call to the system (kernel)"). Available system calls and their interfaces differ between operating systems. Almost all systems provide calls for accessing files, creating new processes and doing interprocess communication. Also system calls for doing networking are common.

Doing a system call is different from doing a regular function call. When you do a system call, you first call a kind of "stub" which is defined in the system's C library. This stub handles the operating system specific part of doing the system call, which involves putting the call's parameters at the right location (special format on the stack or in specific CPU registers), causing some kind of trap to notify the kernel that the process is requesting a system call and finally handing back the return value given by the kernel. For Linux the trap is to request interrupt 0x80, by requesting the interrupt the kernel will start executing at the correct interrupt handler and handle the system call. Please recall that the kernel and user-space processes are running in different protection levels and the mode switch to the highest privilege level is actually necessary, since only in that level code is allowed to directly access the hardware.

2.3.4 Signal handling

Like user-space processes can communicate/request things from the kernel using a system call, the kernel also needs a mechanism to notify a user-space process of certain events. UNIX systems use a scheme called "signals" for this. There is a list of defined signals, basically numbers coupled to a name, which can be sent to a given process. Common signals are for example signals for letting a program know that it is being interrupted or terminated, or the famous "segmentation violation". When a program is signalled with a segmentation violation this means that it attempted to write to or read from a memory location to which it does not have access. Most C programmers are very familiar with this signal, usually they made a mistake related to pointers. If you try to read from an address an uninitialized or messed up pointer is pointing at, there is a very big chance you will be confronted with a segmentation violation.

Some of the CPU exceptions, which we discussed in an earlier section, are directly mapped to signals. Signals like "illegal instruction" or "floating-point exception" are common.

Signals are not only there to allow the kernel to send notifications to user-space processes. User-space processes can also send signals to each other using a system call – this means that the kernel will actually deliver the signal. The involvement of the kernel is important here, because sending a signal basically means interruption of a process. When the kernel sends a signal, the current "context" of the process is saved (i.e. the program counter, contents of the different registers – basically the current state). After that a specified handler in the user-space process is run, if defined. Processes can set up a signal handler for each signal.

When that handler is done executing, the saved state will be restored. Sometimes a signal is so serious, in the case of a signal violation or immediate termination of program, that a process will be killed by the kernel.

2.3.5 Passing messages

Message passing is a well-known method for doing interprocess communication. Processes can use it to communicate with each other. Since in micro kernels the different components are separate processes, message passing is used for communication between those components. Most implementations have `send` and `receive` system calls, some also have a `sendrec` call, which sends a message and then waits for a message to return.

There are several ways to implement the actual sending of messages. One method is to use some kind of message queue. Using this, multiple messages can be queued for processing by another process. A second well established method is "Rendezvous" message passing. With this method there are no queues involved, but processes are limited to receiving a single message at once. If the message buffer is full, the sending process will be suspended until the message has been read and the buffer has been cleared. This can also be used as a synchronization primitive, since the sending process is suspended you can make guarantees that the sender and receiver will only continue executing after they have both reached the specified points in the programs.

2.4 Managing files

2.4.1 General file system structure

A file system consists of two parts: the actual files and an index plus some meta-information to be able to find those files back. The byte sequence on disk starts with the super block. Next to being able to identify the file system using the super block, it also contains important information about the file system's layout; it tells you how many blocks are available for storing the file index, where the file index is located, a pointer to the root directory of the file system, etc.

The space in the partition after the super block and the file system index is filled with the actual files. Just as the memory is divided into pages, disks are divided into blocks. Most files cover more than one block and it is the file systems index' task to keep track in which blocks each file is stored. File systems are not static, the sizes and content of files change over time. As files incrementally grow, more blocks are allocated to the file to store its contents. The blocks the file's content is stored in usually do not form a subsequent list but are scattered all over the disk.

2.4.2 Inodes and file descriptors

The structure of the file system index is specific to each file system. It is a very important part of the file system's design, badly designed indices will prove to be a performance penalty in extreme cases, for example when you have to deal with directories with a lot of entries.

Many UNIX systems work with index nodes ("inodes"). Each file in the file system (often directories and symbolic links are files too – with a special format) is associated with an inode. Every inode has a unique number and it contains the attributes of the file (file length, several timestamps indicating when the file was created, last accessed and last modified, access

permissions, etc) and a table with all the block numbers where the contents of the file are located.

Directories are stored as a file with (inode, file name) tuples. When one tries to access a file with a given name, the inode is looked up using the inode number belonging to the file. In that inode all further information needed to access the file can be found.

When you open a file using a system call, you usually get a number called a "file descriptor" as a return value. The kernel maintains a (per-process) table with information about each file descriptor. Next to the inode of the file that has been opened, the kernel also needs to maintain a field with the offset of the file descriptor – this is at which location the application currently is reading from/writing to the file.

2.4.3 Caching

Compared to reading from memory (which is already slow compared to reading from the CPU's registers), reading blocks from a disk is very slow. When you are accessing a file more than once, you really want to temporarily keep the file in main memory; this technique is called caching. The size of the disk or block cache depends on the size of the main memory and the available space next to the running processes. Often not the entire file is stored in main memory, just the blocks which are accessed most.

At some point your block cache will be full and you will have to remove some blocks from the cache before you can read in fresh blocks. There exist several algorithms for determining which blocks you can best remove from the block cache. Using a wrongly tuned block eviction algorithm will notably affect the performance of the system. Popular methods are FIFO, or first-in first-out, which means that the block which has been added to the cache most recently will be the first candidate for eviction from the cache; or LRU, least recently used, which removes the least recently used blocks from the cache first.

When a process is writing to a file, it is writing to the cached block in the main memory, because directly writing the changes back to the file takes far too much time. Note that when we are using an LRU scheme to find candidate blocks to remove from the cache, a file that is constantly being written to will not be removed from the cache any time soon. What happens with the changes of this file? What happens when the system crashes in between? A viable solution to this problem is to frequently (say, every 30 seconds) write all changed blocks back to the disk. Another solution is to modify the cache maintenance algorithm to remove blocks that have been modified and contain important information (for example inodes) much earlier.

3 Minix in detail

Armed with the preliminary knowledge acquired in the last chapter, we are now ready to have a detailed look at the Minix operating system. We will walk through the Minix kernel source code in a sensible order and briefly describe each component.

Note that not all of the code has been studied, we left the Internet server and TTY driver out of the discussions. The same holds for several other device drivers included with Minix; understanding the code of such drivers requires full understanding of the hardware the driver is controlling.

3.1 Origins

Minix was originally started as an operating system for educational purposes. Minix is a POSIX-compliant UNIX on the outside, but a modular micro kernel on the inside, in contrast to most other unices, which are monolithic kernels. The first version of Minix was released around 1987 [7] and a newer version of Minix, version 2.0, around 1996 [15]. Both releases were accompanied by the famous book, with a broad discussion on operating systems in general and a detailed study of the Minix code.

This thesis focusses on a more recent version of Minix, 3.x, released in 2005. While Minix 3 is still meant to be used for educational purposes, the scope has been broadened as it is now also being used for doing research. On the website [10] the goal has been set to make Minix 3 usable as a serious system on resource-limited computers; which is clearly achievable because of Minix' very small memory footprint. In addition Minix 3 is being pushed for applications which require high reliability and research is being done to make Minix highly reliable and secure. This is certainly doable because of Minix' micro kernel design, we will see later on why.

In the remaining sections of the chapter we will study the Minix 3.1.2a source code, this version is not much different from the version that comes with the 3rd edition of the book (3.1.0). A newer release, 3.1.3, has been available after starting this study and is significantly different in some areas.

3.2 The Minix kernel

The Minix kernel is very well-known for being a full, modern micro kernel. The different things a kernel has to do are split up between different components. Each component runs in the highest possible protection level (i.e. the least permissible one). We will call the components that are running in the lowest protection level, which are referred to as tasks, the kernel. The remaining components, running in user-space, are usually named device drivers and servers.

Some calls, for example accessing I/O ports, can only be done in the lowest protection level. If a task needs to execute such a call, it will have to send a message to the kernel. The kernel will check if the sender is actually eligible for executing that call and if it is it will execute the call and send a reply with the result back. Note that the set of kernel calls a component may execute differs per component.

3.2.1 Structure

In figure 2 it is clearly visible that Minix is structured into layers. The kernel-space part is of course running in the most permissible protection ring of the CPU, the user-space part in

the least permissible. We will briefly go over the layers from bottom to top.

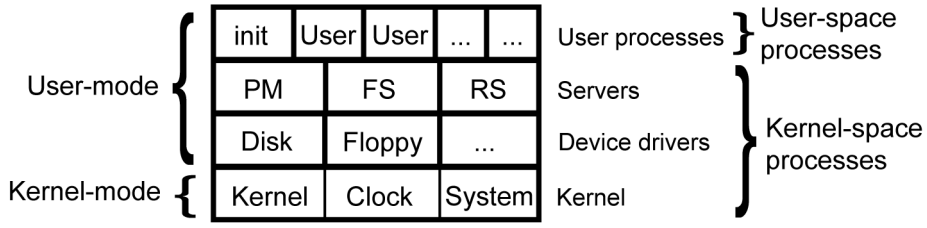

Figure 2: General structure of the Minix system

At the bottom we see the kernel and two of its companions. What is marked as the kernel here is only responsible for getting the system going during boot time, scheduling processes, handling interrupts and exceptions, and passing messages around. These will all be discussed later on in this chapter. Note that the clock and system tasks both run in the same memory space as the kernel, though they both run with their own call stack and are separately scheduled. You can see it as a single memory space with several processes/threads running in it.

In the level directly on top of the kernel, the device drivers are running. As argued earlier, in a proper micro kernel all of these drivers are running as separate processes in their own memory space. Compare the separate drivers being boxed by a thick stroke, as opposed to the kernel-space layer. All device drivers are allowed to make calls to the kernel for requesting data to be read from or written to an I/O port; this is achieved by setting the appropriate system call mask, see section 3.7.1 for more information about this.

Next, in the third level, the kernel's server processes are running. On every Minix 3.x system at least the process manager, file system and reincarnation server are running. Each of these will be discussed in this chapter. Optional are for example the network server, for connecting to intra- and internets; and the info server, which can be used to get information about the state and contents of the kernel's data structures. These server processes are also eligible to do a number of special kernel calls, especially for retrieving information about running processes and the boot image from the kernel.

In the upper level all of the user's processes are running. Init is a special process here, recall that init is the first process that is started on a UNIX system (and gets number 1 as process ID) and all other user processes are spawned from init. User processes do not have access to the special kernel calls and can only communicate with the kernel by means of doing system calls.

3.2.2 Message passing

Message passing is the blessed method for doing inter process communication between the different kernel processes. Minix has five means of sending messages:

1. `send`; send a message,
2. `receive`; receive a message,

3. `sendrec`; send and receive a message in a single call,

4. `notify`; send a notification,

5. `echo`; directly send the same message back to the same process.

Minix uses the "Rendezvous" scheme for passing messages, this means that if you send a message to a process which is not waiting for a message (by calling `receive`), the sender process will be blocked until the recipient calls `receive`. You can imagine this as a kind of handshake or, hence the term, rendezvous. The same holds for `receive`, if a process calls `receive` it will be blocked until a message is sent to it. By using this scheme, we do not have to take care of message buffers and queues.

Now we know that the `send`, `receive` and `sendrec` calls can block, we can understand the need for the `notify` call. A kernel process needs the possibility to notify another process of something, for example an event that has occurred, without being blocked waiting on the recipient. This is exactly what the `notify` call provides. Echo is also non-blocking, as it does not have to communicate with, and thus to wait for, other processes.

The calls listed above are defined in the small Minix system library, against which all processes are linked. They are implemented as small assembly routines that put the arguments to the call in registers and then trap to the kernel by executing the system call interrupt. At this point the kernel will immediately take over control (at the `sys_call` function in `src/kernel/proc.c`). Before this function will actually perform the requested operation, it will first go through a number of sanity checks:

1. Is the process allowed to make this trap? Its trap mask (a bit mask indicating which set of the `send`, `receive`, `sendrec`, etc calls a process may request) is checked to see if the requested operation is allowed. If the requested call is not one of `sendrec` or `receive` and the destination is a kernel task, the trap will also be denied. This is to avoid the situation where a process did a `send` call and will never call `receive`, causing a task to block. Therefore only `sendrec` and `receive` are allowed when communicating with tasks and thus it is guaranteed that task will always reply.

2. Next the given source/destination process number is checked for validity. Of course the process number has to be in the range of process numbers handed out by the kernel and the respective entry in the process table should not be empty.

3. For all calls except `notify` the message has been prepared by the process and has to be fetched from memory by the kernel. In this step the message pointer given by the process is checked to assure that the complete message is located in the process' data segment.

4. Just as there is a trap mask to check which traps a process is allowed to make, there is also a bit mask which indicates to which other processes the caller of the trap is allowed to send messages. If the destination has to be checked, it is verified that the destination process is present in the caller's mask with the allowable processes to send to.

5. At last, Minix uses a small algorithm to check whether this call will cause a deadlock. Starting at the source/destination of this call, we start walking down its dependency chain. Here, a process has a dependency on another process if it is currently in a

`send` or `receive` call waiting on the other process. If we find the current caller in this dependency chain, we know we will have a deadlock, since the caller will start waiting for the source/destination but the source/destination is already waiting for the caller. There is one exception: if the dependency chain exists of only the caller and the sender/recipient and they are directly sending/receiving to each other. In this case there is no deadlock.

If any of these checks fails, an error code will be returned to the calling process. Otherwise, the requested function is executed.

3.2.3 PIDs versus endpoints

In Minix several methods are used to identify processes. Like most other operating systems Minix assigns a unique number to all processes running in user-space referred to as the process' PID ("process ID"). Most UNIX systems often hand out PIDs until they reach 32768 (2^{15}), then they return again to 1 and look for a free PID from there. Because numbers in computers are bound by the number of bits used to store the number in we cannot continue to hand out numbers into infinity.

Minix only uses PIDs for the user-space processes and therefore only the process management server knows about them [3]. Internally the Minix kernel uses a different scheme. Operating systems typically have arrays with data blocks containing the process' information. Rather than a list, since the array allows us to get access to arbitrary items in constant time. The size of these tables differs amongst operating systems, often it is a constant that can be changed at compile time. Since this array will always be in memory, it cannot be very large because that would be a waste of memory. The default size of this process table is 100 slots in Minix. It is obvious that this puts a bound on the number of processes that can exist at the same time. Instead of PIDs these process slot numbers are used to identify a process in the Minix kernel. The process manager maintains a mapping between these slot numbers and their respective PIDs.

When using the messaging primitives discussed in the previous section, the slot numbers are indeed used to identify processes. We cannot use the PID here, because for one the kernel does not know about PIDs and secondly the different kernel tasks do not get a PID assigned. This does also mean that user-space processes cannot use these messaging primitives, they can only communicate with each other by using a different IPC mechanism.

An interesting observation here is that process slots get used by different processes over time. Once a process terminates execution the process slot it was filling will be freed. Later on a newly created process will be placed in this process slot. Say we send a message to the process in slot 58. Before this messages reaches its final destination, the process in slot 58 ceases execution. A new process is created and placed in process slot 58 that is empty. Once this new process will start to listen for messages, it will receive the message that we originally meant to send to the *old* process in that process slot [3] – with this system messages can thus potentially end up at the wrong processes.

For solving this Minix introduces "endpoints". The general idea is that every process slot also gets a generation number assigned. When the process table is initialized, every slot gets 0 as generation number. Every time a new process is put in a process slot, its generation number is increased. From the process slot number and its corresponding generation number we can generate a unique endpoint number. And from an endpoint number we can get the

process and generation number. Instead of the process slot number, the endpoint number is used for communication with the messaging primitives.

We can now easily see how this solves the problems in the scenario we saw before. If we now send this message again and the new process receives this, the kernel will check the endpoint number. The kernel will then note that the generation number has been changed since the time the message has been sent. It now knows that the process this message was destined for does not exist any more and can discard the message.

Because we still have to map the pairs of process and generation numbers on a finite set of integers, we cannot continue to increase the generation number infinitely. Therefore the generation number wraps around, back to 0, after reaching 10. Of course this does not give use guaranteed protection for the issue outlined above, but the possibility that 10 processes have occupied the process slot in the time between sending and receiving the message is very unlikely.

Note that this endpoint mechanism has only found its way into Minix recently. Versions of Minix before 3.1.2 did not contain this endpoint mechanism.

3.2.4 Scheduling

The scheduling algorithm is based on the round-robin algorithm, and extended to support prioritizing processes. By default there are 16 priority levels, with 0 being the highest and 15 the lowest. Each of these levels can be seen as a round-robin queue and has a front and a back.

Only processes which are runnable can be found in the several queues. When new processes are created (so also the processes in the boot image on boot), they are placed in a pre-defined queue. The kernel processes (clock and system tasks) are both in the highest priority queue. Servers and device drivers are initially spread over queues 1 to 3. Queue 7 is the initial queue for user processes. And, at last, queue 15 is reserved exclusively for the idle process. Because the idle process is always runnable, there is always a process that can be found in the queues.

As processes run, they can be moved to other queues, as their priority is increased or decreased by the scheduler. The user can affect the priority of a process by the UNIX `nice` command. A process can never be in the idle queue and thus queue 14 is the lowest queue a process can reach. The priority of both the kernel processes and the idle process will never be changed.

Processes are put on and removed from the run queues by several parts of the kernel. For example the messaging code will remove a process from the run queues if the process called `receive` and there is no message available, which will make it block (the process is not runnable any more). The kernel can achieve such tasks by the `enqueue` and `dequeue` functions, both defined in `src/kernel/proc.c`. The `dequeue` function will simply remove the given process from the run queues. The `enqueue` function will call the scheduler (explained below) to find out in which of the run queues a process has to be placed. Both of these functions will call `pick_proc` when they are done, since these functions change the state of the run queues, the pointer pointing to the process to run next has to be updated and this is exactly what the `pick_proc` function does. `pick_proc` will start searching for a process at the highest priority run queue and continue with the subsequent run queues. Once it finds a process it will set `next_ptr` to it and return.

The actual scheduling is done in the function `sched`. Given a process, this function will return which queue to use for insertion and whether to insert the process at the front or back

of that queue. The following things are checked:

- First it is checked whether the process has any time left in its current time quantum. If it has been fully consumed it is given a new quantum. If the previously run process was the same process, its priority will be decreased. This is to catch run-away processes that are stuck in an infinite loop for example, you do not want these processes to consume all available CPU time leading to starvation of other processes. Otherwise, the priority is increased since it is clear the process needs a bit more CPU time now.
- If the process is not one of the kernel processes or the idle process, the priority is updated accordingly. The new priority is checked to not exceed the bounds.
- The new priority of the process corresponds to the queue to place the process in. If the process has any time left in its quantum, it will be placed at the front of the queue, else at the back.

3.2.5 Interrupt and exception handling

Especially device drivers will want to be notified when a specific interrupt, in particular the interrupt where the device they are controlling is connected to, occurs. The kernel has a mechanism that allows for registering "IRQ hooks" for interrupts. Multiple IRQ hooks can be registered for the same interrupt, which makes sense since multiple devices can share a single interrupt. The kernel functions `put_irq_handler` and `rm_irq_handler` control adding and removing these hooks. Of course device drivers cannot call these kernel functions directly, so the system task exports this functionality to selected process layers. For all hooks registered via the system task the system task will call `put_irq_handler` with a pre-defined generic interrupt handler. This interrupt handler will translate all interrupts it receives to messages. Basically it adds a bit to the process' pending interrupt masks and then sends a notification, so the kernel will not be blocking in its interrupt handler. If the kernel blocks on a process to take action in its interrupt handler we have a big problem, because as long as an interrupt handler routine is active, none of the processes get to run.

During system start-up the kernel will setup interrupt handlers for all 16 hardware interrupts. These handlers are "registered with" the CPU, so the CPU knows which function to call when an interrupt occurs. They are very small routines written in assembly language, which do nothing more than calling `intr_handle`. This function will call all the registered IRQ hooks for this interrupt. Before calling the hook, the hook is marked as active. If the hook returns true the hook is unmarked again as active. If, after running all of the hooks, some hooks are still active the interrupt will not be unmasked/acknowledged; this means this interrupt will be ignored until it is explicitly unmasked again. When an interrupt is not immediately unmasked one of the IRQ hooks will unmask the interrupt later on. After running these hooks the kernel will switch the context to the next process to run.

Exceptions work likewise, just like for interrupts a handler will be registered for each exception. This handler is again a small routine in assembly language which will eventually call `exception` (in `src/kernel/exception.c`). If the exception occurred while running a process the kernel will send the appropriate POSIX signal to the process.

3.2.6 The clock task

In the previous chapter we discussed that the timer interrupt is what makes multitasking systems work. The clock task is the one that handles this interrupt. Upon initialization, the clock task sets up its timer interrupt handler and the timer hardware. It can do this directly since the clock task is running in kernel-space, as we saw in figure 2. Also the functions which provide the timer functionality are implemented in the clock task. Tasks can set up a timer by providing an expiration time (in clock ticks) and a "watchdog function". When the expiration time is reached the clock task will run the given watchdog function.

The interrupt handler does do a little bit of work, it increases the uptime counter, updates the number of clock ticks the running process has used until now and checks if one of the timers has expired. If indeed the running process is due for replacement or one of the timers expired, the interrupt handler will send a notification to the clock task. The clock task will receive the message and call the scheduler and watchdog functions if required.

The Minix source code clearly states (in `src/kernel/clock.c`) that the interrupt handler does a little bit of work, as described above, so that the clock task does not have to be called on every clock tick. This feels like an attempt to avoid a lot of context switches. In fact you could see the clock task as a kind of device driver for the on board timer hardware, that can be moved to user-space and controlled via messages. Tanenbaum seems to argue that it belongs in kernel-space, since it only interfaces with the kernel [8, pp. 113]. It is more likely that it has been done this way to avoid performance problems, theoretically sacrificing the micro kernel concept, that would be caused by context switching between the kernel and clock task frequently for handling timer interrupts, scheduling and watchdog functions. Since the clock task is in kernel-space anyway, it has been very closely knit with the kernel: observe that it is directly calling functions belonging to the process scheduler and timer handling.

3.2.7 The system task

There has to be a "gateway" between the user and kernel processes. The system task is this gateway; it handles all the kernel calls done by the server processes and device drivers, such as I/O with the hardware. When a process wants to do such a call, it sends a message with the number of the call requested and its parameters to the system task. The system task checks whether the number of the call is a valid number (and so the call actually exists) and whether the call number is marked in the process' call mask, which means that that process is allowed to do that call. If those tests succeed the system task will carry out the call and send the result back. This is the only type of message the system task handles and the only thing this task does.

Many different things are handled by the system task, since the servers and device drivers have very different needs. We will not list all the different calls separately here, but it does make sense to list all categories here with a small description to get more insight in what the kernel does and does not handle:

- Process management; calls like `fork`, `exec`, `exit`, for handling the kernel side of those calls. As we will see later on, doing process management is an interplay between the process management server and the kernel.
- Signal handling; `kill`, `sigsend`, `sigreturn`. Signal handling requires manipulating a process' stack, which can only be done by the kernel; we will explain this in further

detail later on.

- Device I/O; `irqctl` for enabling, disabling IRQs and setting up interrupt handlers; `devio`, handles the input and output of bytes, words to I/O ports.
- Memory management; setting up a new process map, manipulating segments; both also needed by the process manager for setting up new processes.
- Functions for translating addresses and copying data between the virtual and physical address spaces.
- Calls for getting uptime and process times and setting up an alarm handler.
- System control; `abort`, getting system information.

These different kernel calls are implemented in separate files in the `src/kernel/system/` directory. We will delay explaining the details of these calls until the sections where the respective kernel component using the call the most is further explained.

3.2.8 System call invocation

In chapter 2 we have seen what a system call is and how most modern operating systems implement system calls. The arguments to the call are set up (usually in the CPU's registers or on the stack) and a specific software interrupt is requested. This makes the machine switch to kernel-space and the kernel executes the call. When finished the kernel hands control back to the user-space process together with a return value for that call.

Just like the kernel calls, system calls in Minix are nothing more than messages. When a user-space process (this also includes the various servers) does a call like `read`, it actually calls a small function in the C library. This function will create a message with the correct call number and the arguments and sends this message to the correct server. For `read` this is of course the file server. The file server will receive this message, handle the call, and reply with a message containing the return value. Note that the user-process is obliged to do a `sendrec` call, so it will be blocked waiting for the return value.

From a first glance it looks like there is no "trap" involved like calling the software interrupt in other operating systems. Do not be fooled however, as we saw in section 3.2.2 sending a message involves trapping into the kernel using a software interrupt. In Minix the trap is actually abstracted away.

3.2.9 Locking

Since the kernel tasks are all running in the same memory space and thus share data, there is a need for doing locking. Different kernel tasks can and do manipulate the queues with runnable processes, it can happen that an interrupt occurs while a kernel task is manipulating those queues and another kernel task may end up changing the run queues too while the other task is in the middle of manipulating. It is clear that this will lead to unwished corruption of the queues.

To protect the queues and other shared data structures from concurrent manipulation, a lock will have to be obtained before continuing. When done changing the data, the lock will have to be released. Minix has the `lock` and `unlock` functions for this in kernel-space. For

commonly used functions which need to obtain a lock there are functions which will both obtain and call the wished function. For example there is a function called `lock_enqueue`, which will obtain the lock, call `enqueue` for putting a process in the run queues and then release the lock.

The `lock` and `unlock` functions are actually implemented by disabling and enabling interrupts respectively. By disabling interrupts you are guaranteed that the timer interrupt, or any other interrupt, will not be called and the currently running process, which will be changing the data structures, will not be pre-empted. That is, will not be interrupted and replaced by another process.

Since most of the kernel components in Minix are all running as separate processes and thus do not share their data, there is not much need for locking. Monolithic kernels, which have an enormous amount of data to be shared (keep in mind that such systems can have several kernel threads running in kernel-space, and sometimes even the kernel itself can be pre-empted) have many more locks and raise the need for more sophisticated locking. For monolithic kernels running on big multi-CPU machines, doing locking by enabling and disabling interrupts does not cut it.

3.3 Boot process

We have seen that when the BIOS has finalized doing the low-level initialization of the hardware, it starts looking at the various disks for a boot sector. When it finds a disk with a valid boot sector, it is loaded into memory and executed. Recall that a boot sector can only be 512 bytes in size. This boot sector then loads the Minix boot loader into memory and control is passed on. The bootloader is not limited by the 512 bytes disk limit and thus can be more feature full. Upon start-up, it presents the user a with a small shell, wherein certain options can be set and a kernel can be loaded.

After a valid kernel image has been chosen and the boot process has been initiated, the boot loader will try to load the given kernel image into memory. Once done, there is jumped to the entry point in the Minix kernel executable, written in assembly code. From here on the necessary tables which the CPU needs before it can enter the "protected mode" are set up. In the protected mode memory protection is enforced and multitasking is possible. Also the default interrupt and exception handlers are being set up.

When this is all done, the function `main` is called in `src/kernel/main.c`. This will first initialize the interrupt controller, something which does not necessarily have to be written in assembly code. It then proceeds with more higher level tasks, such as clearing the process table and filling it with the executables found in the boot image. Because it is impossible to load anything from disk without having the process server, file system server and disk driver in memory, all important servers and drivers are included in the boot image and loaded into memory by the boot loader. The loop which sets up the different processes will also call the earlier discussed `enqueue` function to put all processes in the run queues. When done, the `restart` routine is called, a small routine, also in assembly code, which will switch context to the next process to run.

From this point onwards the remaining tasks will not always be executed in the same order, because the scheduler now decides who is to run next. The file server will immediately call `receive` and will block waiting for the process manager. The process manager will first request some information from the kernel to set up its own copy of the process table. When done it will send another copy of the process table to the file server (which is indeed blocked

to wait on this information). When both of these servers finished synchronizing, they will start listening for calls from all parts of the system.

Soon, also the `init` process will get a chance to run. As with most UNIX-like operating systems `init` is the first process that is started and subsequently receives number 1 as PID. Once it is started, `init` will execute the `/etc/rc` file, which will set up the system clock, check the file systems and start some system services. Next, `init` will start consoles as specified in `/etc/ttytab`, restarting them when a session has been closed. The system is now fully up and running.

3.4 The process manager

An operating system is mostly useless if you cannot create new processes. For creating new processes and loading executables with program code we use the `fork` and `exec` system calls respectively. These calls are implemented by the process manager, another server running in user space. For fully completing these tasks the process manager requests the kernel, via kernel calls, to do some of the work that is required to be done in kernel-space. We will go into more detail on this shortly. Other than the fact that the process manager is also responsible for handling UNIX signals (it implements the related system calls), it also handles various other tasks such as the `ptrace` system call used for debugging, the system calls for retrieving and setting the system time and finally some miscellaneous system calls that deal with rebooting, allocating memory and setting process priorities. There is also a very rudimentary implementation of "swapping", which can be enabled/disabled during compile time. We will not discuss the `ptrace` system call in this thesis.

3.4.1 Initialization

The process manager has a lengthy initialization procedure, which warrants its own section. The following tasks are completed:

- The process table is initialized. The process manager keeps its own copy of the system's process table.
- Two sets of signals are built, one with all signals causing core dumps, which will be used later on to determine whether a core dump has to be made; and one with all signals that are to be ignored by default.
- Next the parameters passed in by the boot monitor (the boot loader) are requested from the kernel. This parameter list is then parsed, one of the parameters contains a list with memory areas that are available to Minix. Also a structure with some miscellaneous kernel information is requested.
- In the list of available memory areas we just requested, the memory in use by the kernel itself is contained. This is fixed by getting the memory map of the kernel process from the system task and "patching" the list of available memory by removing the memory area in use by the kernel from it.
- Now the initialization of the process manager's process table is continued. An image table of the boot image is obtained, again from the kernel. This table contains details of all processes loaded into memory by the kernel during boot time. These details include

the name of the process, priority value and pid. Also for each process its different signal sets (saying which signals to ignore, catch and translate to messages) are cleared and the memory map is obtained and patched into the list of available memory. At last the file manager is told about each process handled here for synchronizing its process table.

- Some of the details just set into the process table are overridden. The `init` process get its magical pid and is set to be the root of the process tree. The process manager is guarded against all signals by setting its signal map such that all signals are ignored. For the file server, tty and memory drivers all the signals will be forwarded as messages.
- We are done setting up the process table and the file server is messaged that no more new processes will follow: the synchronization step is complete.
- Finally, the hole list, which is used for finding a suitable spot of memory to place new processes, is initialized from the list of available memory areas. Information is printed on the screen about the amount of the total and available memory of the system.

3.4.2 Managing memory

Using the hole list set up during the initialization phase the process manager will manage the available memory for processes (in earlier versions of Minix the process manager was actually known as the MM or memory manager). The memory allocated to a process is described in a memory map. Setting up these maps happens at two places; during system setup where the processes contained in the boot image are being placed into memory (see section 3.3 for more information about how Minix boots) and when the process manager is requested to create a new process (the `fork` system call) or load in another executable (the `exec` system call).

We will start with discussing regular user-space processes and then see how the kernel tasks differ from this. In section 2.3.2 we have seen that processes have text and data segments and in the data segment the stack and heap both grow to the "middle" of the data segment. This model is exactly what Minix is using and is known as "common I&D". Here "I" stands for instructions and the instructions of a program are typically in the text segment. Minix uses one other type, called "separated I&D", where multiple processes which are running the same executable are sharing one and the same text segment in memory. So, in this case there is a single text segment in memory and multiple data segments for all the different processes. This is a nice memory saving, if we have 20 processes in memory running the same program we have only 1 text segment with this data in memory instead of 20.

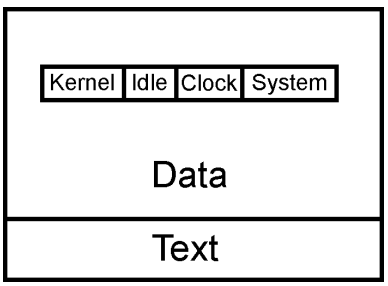

Figure 3: The kernel's memory map

The Minix kernel is a single executable image, so to start with we have a single text and data segment in kernel-space. However, the kernel consists of multiple processes, or threads, all running in this same data segment. Each thread needs its own stack and is scheduled separately. Because of this the data segment contains multiple stacks to accommodate this. This is depicted in figure 3.

On Minix start up, we have seen that the kernel sets up all processes contained in the boot image, this includes the process manager. All processes created later on, are all created by the process manager as we will see in the next section.

Memory maps, as explained above, are created and manipulated by the `fork` and `exec` system calls explained in the next section. When these calls need to allocate memory for a process, they use the `alloc_mem` call, with as argument the amount of space needed. How this memory is further divided into the different segments is up to the call setting up the memory map. Using the hole list, which has been created during the initialization phase, `alloc_mem` can find a hole which is big enough to allocate the requested amount of memory. The algorithm used for this is very simple, we iterate over this hole list until we have found a hole that is big enough – a "first fit" algorithm. After finding a hole, the hole list is updated and the address where the memory area begins is returned.

Allocated memory areas are freed using `free_mem`. This function does nothing more than merging this area back into the hole list.

3.4.3 Creating new processes

A modern operating system would be pretty useless without the `fork` and `exec` system calls. These system calls are used for creating new processes and loading in executables. We will discuss how these calls work internally in turn.

In general `fork` works by cloning the memory space and "forking off" a child process starting at the same location as where the parent forked. The return value of the `fork` call in both of the processes will indicate whether the process is the parent or the child. The following things happen in turn:

- The process table is checked to see whether there is still a free slot.
- We determine how much memory we need to allocate. Since both processes will continue to run the same program code, we can take advantage of the "separated I&D" method and we can suffice by cloning the data segment, since the text segment containing the (read-only) program code can be shared. The amount of memory needed is immediately allocated using the `alloc_mem` function.
- If the memory allocation succeeded, we copy the parent's data and stack to the child's memory using the `abscopy` kernel call. Note that as a user-space process, the process manager does not have direct access to the child's address space.
- Next, the process manager iterates over the process table to find an empty slot to set up this process. The slot is set up, fields are cleared, the parent pid is saved, the memory map is set up, we obtain a free pid and assign this to the child.
- Both the kernel and the file system server are told about this new process – this will cause both to create a new entry in their private process tables. The kernel is separately

informed about the process map that has been set up for this process using the `newmap` kernel call.

- At last the `fork` operation is done, we set a pending reply for the child process with 0 as return value which will cause the child to wake up. `fork` itself returns with the new PID of the child as return value, to be returned to the parent process.

Usually, the new process wants to load in another executable image next. The `exec` system call is used to accomplish this. This system call is slightly confusing, because some of the work is done in the system library. In the directory `src/lib/posix` the different implementations of the system call can be found, which will all finally call `_execve` in `_execve.c`. What happens here is that a small stack image will be created, containing the argument vector strings, the environment strings and appended to that a list of offsets to all strings in the argument vector and environment.

Then the user-space part of the `exec` system call is done and we switch to the kernel side of this system call. The following things are now done:

- Some checks are done to see if the passed in arguments are valid.
- The string containing the file name of the executable is fetched from the calling process' address space (only the pointer is passed in with the message). Also the stack created by the calling process is copied.
- Using the file name we check whether that file is executable. We read in the file header to see if this is a binary executable or a script. For a script the argument vector is changed to include the path name of the script to the executed and all arguments which are on the `#!` line of the script. The interpreter is fetched from the environment and this file is checked for executability.
- Now we have a path to a file with executable binary code, we check if there already is a process running the same executable so we can share the text segment.
- A new memory map is created, with a clean stack and data segment. The new memory is allocated, on success the kernel is told about the new map via the `newmap` kernel call.
- The stack image created by the code in the system library is fixed up by relocating the offsets to the strings relative to the address space of the process. This image is then copied to the stack of the process.
- The text (if it cannot be shared) and data segments are read in from the executable file into memory.
- We update the uid and gid, and tell the file server about these so its internal process table can be updated.
- Now we are almost done. The set with signals to catch (see section 3.4.4 for a discussion on signal handling) is cleared. Both the file system server and kernel are informed that the `exec` call has succeeded. No reply is sent to the process, it will continue running the new executable now.

At some point a process will cease execution, either forcefully by a signal or voluntarily by executing the `exit` system call. At all times the parent process is responsible to "clean up" the child using the `waitpid` system call. If this does not happen, the process slot will be kept occupied and the process enters the so-called "zombie" state.

Everything the `exit` call really does is freeing the process' resources in the various components. Pending alarm timers are killed, the file system server and kernel are told that the process is no longer runnable, all processes which are waiting to send or receive a message to/from this process will be alerted, the process' memory is freed, and if this process has any children, they will be disinherited and given to `init`.

Only whether or not to clear the process slot depends on the parent. If the parent is currently waiting for a child process to exit, then the process slot is immediately freed and the `waitpid` call the parent is currently blocking in will return. If there is no parent waiting, the child will be turned into a "zombie" and a `SIGCHLD` signal will be sent to the parent.

The semantics of the `waitpid` call are probably pretty clear already from the explanation above. `waitpid` will wait for the child with the given PID to exit. If the child is already in zombie state when this call is done, then the process manager will cleanup the process slot and the `waitpid` call will immediately return; if not the parent will be blocked until the child is finished. There also is a `wait` system call, which does the same as `waitpid`, expect that no PID is given and so the call will wait for any of the parent's children to exit. In case the parent has no children the system call will return immediately with an error code.

3.4.4 Signal handling

Before we dive into Minix' signal handling machinery it is very important to note that there are two different code paths: signals originating from the kernel and signals originating from user-space (eg. via the `kill` system call) are handled slightly differently.

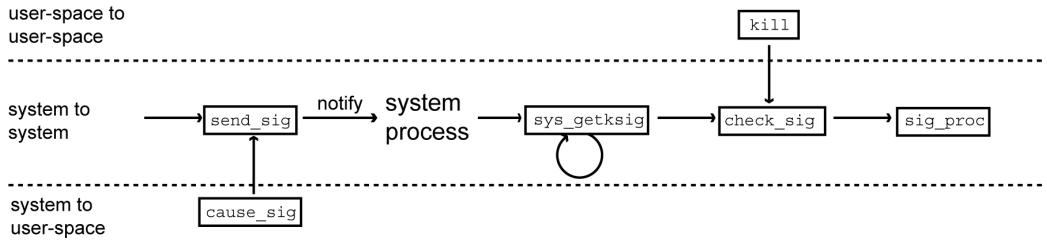

Figure 4: The different signal handling code paths

For sending signals to user-space, the kernel does not use the `kill` system call. Instead the signals are (sometimes grouped) sent to the process manager, which will take care of it further. Here we see where both code paths come together, because the `kill` system call is handled by the process manager signals originating from user-space processes immediately end up in the process manager.

There are two function calls in the kernel that are used for sending out the signals. `send_sig` is used to directly send a signal to a (system!) process using a notification message. This function call is not suitable to send signals to non-system processes, for this the call `cause_sig` is used. `cause_sig` sends the `SIGKSIG` signal to the process manager using `send_sig`.

Once the process manager receives the `SIGKSIG` signal, it knows that the kernel has pending signals for one or more processes. The process manager will now start calling the `sys_getksig` kernel call for retrieving pairs of process numbers and pending signal sets until the kernel is out of processes with pending signals. For each process number, the set of pending signals is iterated looking for bits that are set – this means this signal is currently pending and should be sent. Some signals are meant to be broadcasted to the whole process group or even all processes, for these signals the process number is changed here to reflect this.

Next the function `check_sig` is called with a process number and signal number. This function will search the process table for the process to send the signal to. Several checks are performed to see whether broadcasting to a process group is allowed for the given process and also if the owner of the process is allowed to send a signal to the other process, based on user id. If all passed, `sig_proc` is invoked to actually send the signal. Very important in `check_sig` is that the process manager will not reply with a message if the calling process has killed itself – if we did, the process manager would hang!

The `kill` system call is implemented by directly calling `check_sig` with the arguments found in the message.

The `sig_proc` function handles actually sending the signal to the process. There are several ways this can happen, based on the current state of the destination process. The following cases are handled by `sig_proc`:

- An empty process table slot or a zombie process, in such a case the function `panic` is triggered, causing the system to immediately shut down.
- A process is currently being traced for debugging. The process will be unpaused and stopped.
- The signal that is being sent is in the process' `sig_mask` set, in this case the signal is added to the `sigpending` set (signal should be blocked).
- When a process is currently swapped to disk, the signal is also added to the process' `sigpending` set and the process is put on the swap-in queue.
- If the signal is in the `catch` signal set, we will call the signal handler in the process. We will explain this mechanism in more detail later on.
- If the signal is in the `sig2mess` set, the `kill` *kernel* call is used to send this signal. The process manager, and all other system servers, cannot use the `kill` *system* call to do this, because the system call does a `sendrec` operation which may cause the server to hang. The `kill` *kernel* call requests the kernel to "handle" this signal. As we saw earlier, for system processes the signal is directly sent via `send_sig`, for all other processes the signal is routed to the process manager again via `cause_sig`, which is using `notify` for this (no chance for hangs!).
- Otherwise, a signal cannot or should not be catched and we create a core dump and terminate the process.

The different signal sets that are checked can be manipulated using the `sigaction` and `sigprocmask` system calls. Once all signals in the signal set retrieved using `sys_getksig` have been handled, the process manager calls `sys_endksig` for this process. This kernel call tells

the kernel that we finished handling all pending signals for this process. Usually the process is then put back on the run queues (if the kernel sends a signal to a process, it will also immediately dequeue that process).

Left to discuss is how exactly the signal handler in the process is called. Since the process manager is a user-space process, it cannot access the address space of the receiving process – it will call in the help of the system task to get this done. The system task provides a `sigsend` kernel call for this. This process starts in the `sig_proc` function in the process manager at the point where is checked whether the signal number is in the set of signals that should be catched (`mp_catch`). If this is the case, a `sigmsg` structure will be set up that will contain all data the `sigsend` kernel call needs to send the signal. The structure contains a mask of blocked signals, the signal number that is being raised, a pointer (relative to the receiving process' address space) to the signal handler that should be called, a pointer to the `sigreturn` system call (relative to the address space of the process that is to receive the signal) and the value of the stack pointer at this point.

Next space is made on the stack for the `sigcontext` and `sigframe` structures. These structures are used to save the current state of the process (values of registers, etc) and a stack frame for the signalled process for executing the `sigreturn` system call respectively. A call to `adjust` is used to check whether there is still enough room in the "gap" for growing the stack to accompany this data. If this fails, the process will be terminated immediately. According to the flags set using `sigaction` the `sigmask` and `catch` the signal sets are updated. Finally, the kernel call to `sigsend` can be done done.

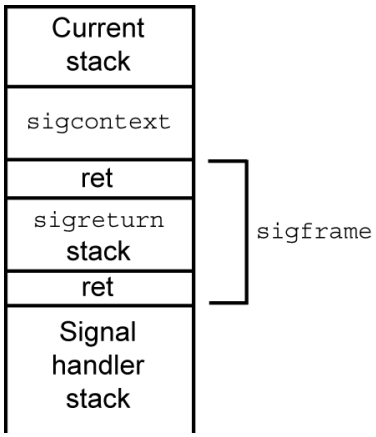

Figure 5: Overview of the stack during signal handler invocation

The `sigsend` call first fetches the `sigmsg` structure, which has been set up by the process manager, into kernel-space. With the information from that structure, it sets up the `sigcontext` and `sigframe` structures and copies those to the stack of the receiving process. If this succeeds the kernel call handler returns and back in the `sig_proc` function the signal number is cleared in the pending signals signal set and the process is unpaused, causing the process to end up on the run queue again. See figure 5 for the graphical representation of the stack.

The trick here is that `sigsend` has put two stack frames on the process' stack, with the

one on top being for the signal handler, and set the program counter to the signal handler, which will automatically cause the signal handler to be run the next time this process is being scheduled to be run by the kernel. If you are familiar with writing assembly programs for x86 machines, you will know that the address to jump back to after a procedure call is put on the stack – these are the return addresses we saw in the `sigframe` structure. We see that on the top of the stack the return address to jump back to is set to the location of the `sigreturn` call in the C library. When executing this call, the "second" stack frame we put on the stack will be used. This system call is handled by the process manager and from there the `sigreturn` *kernel call* is executed. The signal context is read from the stack of the receiving process, using the information in that structure all the registers are restored to the values they had before the signal was raised. Now the signal has been completely handled, the next time this process is scheduled to run it will continue running at exactly the same place where it was interrupted. The sharp-eyed reader will notice that the second return address we put on the stack is actually not used at all when handling the signal. This is indeed true, the return value is actually there for debugging purposes, so the stack traces shown by the debugger still make sense.

3.4.5 Swapping

Minix has a very simple and basic implementation of swapping. Many modern operating systems with an implementation of virtual memory do swapping on a by page-by-page basis. As Minix does not implement virtual memory, swapping in Minix is on basis of processes. Only full processes (i.e. the full text and data segment of a process) can be swapped.

Swapping can be enabled and disabled from user-space by using the `svrctl` system call. When enabled, a file is created on disk and a swap hole list (likewise to the hole list for free memory) is maintained. Once an attempt to allocate memory for a new process fails, the function `swap_out` is called. This function will try to find a process that can be swapped out, once it finds one it will immediately swap it to disk. Processes that are paused, waiting on something or suspended are candidates to be swapped.

When a process is receiving a message or a signal, it will be put on the swap-in queue. The process manager will periodically check whether there are processes on the swap-in queue, and if so these will be swapped in.

Swapping can of course only be turned off when it is possible to put all processes in the swap file back into memory. If this operation fails, swapping will not be turned off and an error code will be returned to the entity that requested to turn off swapping.

3.5 The file system server

One of the biggest pieces of Minix is the file system server. In this section we will discuss the most important parts of this server. Currently the Minix file system server only supports the Minix file system, version 1 and 2. The implementation of these file system drivers is contained in the file system server. In a lot of other operation systems you will see an abstraction layer between the file system server and the file system drivers.

The file system server can work with different disk drivers, and it uses a defined device driver interface to communicate with the disk driver processes. This interface is described in section 3.6.

Please note that the file system server which we are describing in this section is the one

in Minix version 3.1.2a. In Minix 3.1.3 the file system server has been massively changed to be more structured and flexible.

Before we start looking at how the file system server initializes, what it does in its main loop and how the familiar POSIX file system calls (`open`, `read`, etc) are implemented, we will first have a short look at some general and lower level topics. We will not discuss the implementation of the `select` system call, pipes and file locking in this thesis.

3.5.1 Inodes

Inode is short for "index node" and an inode is used for pointing at files and directories on file systems. Inodes are saved on the disk and loaded into memory when needed, and some additional information is added to inodes that are in memory. They all have a unique number and contain information about a file such as the file's mode (type and protection), owner of the file, times of the last access, modification, time of creation, and of course a list of blocks where the file's contents can be found on the disk. In memory, information such as the device the inode is on, reference count, whether the inode is clean or dirty (whether or not modified in memory), is added. For the full, commented definition see `src/servers/fs/inode.h`.

On disk, inodes are stored in so-called inode tables. These tables are created when the file system is created and have a fixed size. On the disk a large bitmap (with an entry for every inode in the disk) is stored, which says if a given inode is occupied or not. Two functions are provided to work with this bitmap, `alloc_bit` and `free_bit`, which allocate and free a bitmap entry respectively. This functionality has been abstracted in these two functions taking a bitmap, since there are more parts using bit maps.

File systems have several thousands of inodes and disk blocks are 4Kb, thus a typical inode bitmap covers several disk blocks. Minix remembers the number of the first block with free bitmap entries. This is a small optimization, allowing `alloc_bit` to start the search for a free bitmap entry from this block, skipping the occupied ones at the beginning. The implementation of `alloc_bit` is not very hard, it simply iterates over the bitmap in these blocks until it finds a free entry, which it will subsequently return and mark as occupied. `free_bit` is even easier, using the inode number it can immediately load the bitmap block containing this entry and mark the entry as free.

For retrieving existing inodes you usually want to use the function `get_inode`, which takes a device number and inode number as arguments. It will first search the in-memory inode table to see if there is an occupied slot containing the inode we are looking for. If so, then the reference count of this inode is increased and a pointer to the inode is returned. If not, and there is still an empty slot in the in-memory inode table, the inode will be read from disk using the `rw_inode` function. This last function nicely abstracts the tediousness of reading/writing inodes to/from disk blocks away.

Here it makes sense to explain the relation between files and directories. On most unices, directories are actually files, but marked as a directory. A directory is nothing more than a file of one or more disk blocks containing a list of file entries that are in this directory, together with the inode for each file. The root directory is special and its inode is saved in the super block (which we will explore in the next section).

3.5.2 The super block

It is obvious that there needs to be a block on the disk containing information like where the inode bitmap is located, where the inode table starts, how large the file system actually is, which inode contains the root directory, etc. Indeed such a block exists and it is called the super block. Just like with inodes the super block is loaded into memory too, and some extra information is added there. Though, in contrast to inodes, the super block is kept in memory as long as its file system is mounted. Inodes might have to be replaced with other ones when we run out of space in the in-memory inode table.

3.5.3 Accessing disk blocks

Accessing disk blocks is done using the `get_block` and `put_block` functions. When a piece of file system server code needs to access a disk block, it will call `get_block` with the block number. After the code has finished performing operations on this block, it has to release the block using `put_block`.

Note that these get and put functions again seem to hint at the use of a reference counting scheme, just as with accessing inodes. And indeed, just as a certain number of inodes are kept into memory, the same holds for blocks. Remember that hard disks are orders of magnitude slower than the system's main memory, so keeping blocks into memory and performing the operations on that copy and then write it back to disk is much faster than directly performing each operation on the block on the disk. As we have seen in chapter 2, this technique is called caching.

So, `get_block` actually checks whether the requested disk block is still in memory, before getting it from disk. If it is, the reference count is increased and the block is returned. Otherwise, the least recently used (LRU) block is taken from memory, flushed to disk if necessary and the requested block is read from disk into here. The `put_block` function, decreases the reference count on blocks and manages the LRU list. Next to the block to release, the type of the block is passed in. This type is used to decide where to put the block in the LRU list. Blocks that are not likely to be needed again soon are put at the front of the LRU list, so they will be the first to be re-used. Some blocks are very important and need to be written to disk as quickly as possible, such blocks are written to disk immediately in the `put_block` function. This happens with for example blocks containing inodes.

Minix has a second level cache to supplement the default block cache of (16 bit) Minix systems. This second level cache can be enabled during compile time and its size during boot time and is implemented as a RAM disk where blocks are stored once they have been released by the first level cache. The cache acts as a FIFO, newly released blocks are put at the start of the FIFO and old blocks will drop out at the other end. When the first level cache needs to read a block from disk, it will first ask the second level cache whether the block is cached there.

3.5.4 General model of operation

Now we have enough basic knowledge about the file system server, we can have a look at the general model of operation of it. Once the process is started, we first go through an initialization procedure. Recall that the file system server has its own copy of the process table and that the process manager will send a message about every process existing during boot time when it initializes. Using these messages the file server will fill up its own process

table, when the process manager tells us it's done, we will reply with an "OK" message to end this synchronization step.

From here on the file server is basically on its own and continues initialization by sanity checking a number of constants set in the header files. After that a few more file system specific things are set up:

- The buffer pool, the available blocks in memory to be used for caching disk blocks, is cleared.
- A table is built which maps device drivers to actual devices. Next, the major number of the device containing the root file system is enquired from the boot arguments and a mapping is made from the boot device driver (determined and compiled-in during compile time) to this device. We will briefly discuss Minix' device driver interface in section 3.6.
- Depending on the current settings of the boot arguments a RAM disk will be created and loaded with content from an image device.
- The super block of the root file system is read in. If this fails we will panic, since we cannot function without a valid root file system. Also the inode of the root directory is loaded. Note that in Minix a root file system is always supposed to be mounted. And this makes sense, because it is almost impossible to run without a root file system, all initialization code and supplementary drivers are on this root file system.
- Now that the root file system can be accessed, we set the root and working directory of every process (which are only the processes which are in the boot image at this point) to the root directory.

After this initialization phase, the file system server will enter its message loop. This message loop handles the `SIGKSTOP` signal, at which the file system will sync and exit, device driver notifications and the valid system calls. If needed, the server will try to read ahead a block, which means that it will read a given block into the cache before it is actually needed and likely to be needed soon.

3.5.5 Mapping between processes and file descriptors

All file accesses in UNIX happen using file descriptors. When you open a file, you get a file descriptor, which will act as a "handle" to that file. Every time you access that file, you need to pass in the handle so the operating system knows which file you actually want to access. Keep in mind that forked processes are exact copies of each other and share the same set of file descriptors. On `exec` file descriptors marked with the `NO_CLONE` flag (using the `fcntl` system call) are closed.

The relationships between inodes and file descriptors are stored in the "filp" table. An entry in this table contains information such as the inode, position in the file, how the file has been opened (read-only, read-write, etc), flags from `open` and `fcntl` and how many file descriptors share this entry. You can find the definition of this structure in `src/servers/fs/file.h`. Every process descriptor in the file server's process table has an array of pointers to "filp" entries called `fp_filp`, which is the file descriptor table.

3.5.6 Disk block management

In section 3.5.1 we saw that the numbers of the disk blocks containing the file contents are stored in inodes. Minix does actually not store block numbers in the inodes, but zone numbers. A zone is an entity that can contain one or more subsequent disk blocks. For example, it is possible to have a file system with zones of 4K and disk blocks of 1K, in this case each zone will contain 4 blocks. One of the reasons this technique is used is that all blocks in a single zone belong to the same file, and if these are all on the same cylinder we can possibly improve the performance when the file is being read in sequentially [8, pp. 554]. Another reason is that by pointing to zones in the inodes, we can create larger file systems without increasing the size of the pointers in the inodes.

Of course we cannot list all of the zones that belong to a file in a single inode, because inodes need to have a fixed size. A Minix inode has 10 fields available for storing zone numbers, and you can imagine that this can never be enough to store all zones belonging to a very big file. This is why we can allocate separate zones which contain lists of zones belonging to this file again, and the inode points to such zones. These zones are called indirect blocks. Of the 10 entries available, 7 contain "direct" zone numbers, the other three can point to indirect blocks. Currently, one of these three is used to point to an indirect block and one to a double indirect block. One entry is left unused.

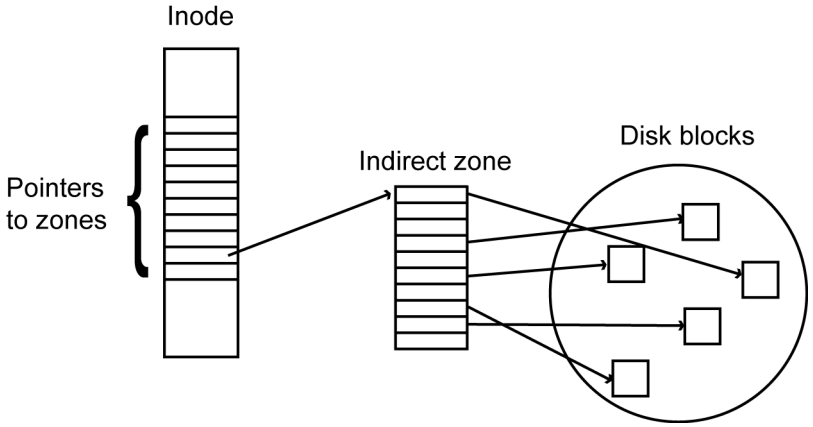

Figure 6: Overview of inodes and indirect zones

When even indirect blocks are not enough, Minix also supports double indirect blocks. This zone contains a list of zones, which are again indirect blocks. The entries of those zone lists are zones of the actual file.

3.5.7 Examples of operations

Basically all of the system calls implemented by the file server are nothing more than manipulators of the various data structures outlined above. In this thesis we cannot fully explain the implementation of each system call in detail. Below, we will discuss three of the more important ones: creating links and mounting file systems (both are forms of inode manipulation); and reading and writing to files (these are more dependent on the disk block management code). Studying the implementation of the other system calls is left as an exercise for the reader.

Creating links

Using the `link` system call you can create a *hard* link in Minix. Minix does not support creating symbolic links. When you create a hard link to a file, it will point to the same file and share the various attributes of the object and the link count is increased. You can remove links with `unlink`, this will basically decrement the link count. When the link count becomes zero, the file is deleted. Often the `rm` command is the same as `unlink`. Note that it is not possible to create hard links to directories, as mandated by the POSIX standard [1].

The system call is implemented in the function `do_link` in `src/servers/fs/link.c`. The first thing that happens there is that we try to retrieve the inode for the file we want to link to. If this succeeds, we check if the file has the maximum number of links already (we do not want the `nlinks` field in the inode to wrap-around) and whether it is not a directory. Though, apparently Minix allows the super user to create hard links to directories.

Next we check the path of the new link to create. We check if the last directory of the path exists, because this is the directory wherein we have to create a new entry. Obviously, the name of the link we are trying to create may not exist in the directory already. Lastly, we check whether the directory where we have to create the entry for the link and the file to link to are on the same device.

Now, finally, we are ready to link. For this we call the function `search_dir` with the inode of the current directory, the file name of the link we want to create, the inode of the file we want to link to, and the `ENTER` operation. The `search_dir` function can handle several different operations, so we need to specify what we want to get done. In summary, when asking for the `ENTER` operation, this function will create an entry for the given file name in the given directory with the given inode number. This involves reading in disk blocks containing the directory entries to see whether the given name is already there. If not, it will be added to a free spot in a current disk block, or a new block will be added if the current ones are all full.

So, we will end up with another directory entry, pointing to the same inode. If the `search_dir` operation succeeds, we will increment the `nlinks` field in the inode and mark the inode as dirty.

Mounting file systems

Other file systems can be "added" to the root file system by mounting. Given a device containing a file system and a directory, the file system can be "hooked up" to this directory. This is also an interesting operation inode-wise and worth to briefly discuss.

What the Minix implementation of the `mount` system call does first, is check if the file system on the given block device has not been mounted yet (of course after checking if the given file is really a valid block device). If not, we read the super block from the file system and verify it is a Minix file system.

Secondly, the given file name is checked whether it exists and is not a special file. That inode may also not be busy; the reference count should be zero, indicating that no other parts have this inode in use. Next, we get the root inode of the new file system and check whether the file types of the root inode and the inode where the file system will be mounted are the same – for now both will have to be a directory. After all these tests have succeed, we can finally mount the file system: this involves setting the `I_MOUNT` bit on the `i_mount` field of the inode where this file system will be mounted, indicating that a file system is mounted on this inode. In the super block we read into memory, we set the `s_imount` field to the inode where

this file system has been mounted. All code in the file system server that is traversing inodes has a special case for inodes with the `I_MOUNT` bit set. When we come along such an inode, we will iterate over the list of (mounted) super blocks, to find a super block with `s_imount` equal to the inode we are currently at. Once we found this super block, we continue traversing at that super block's root inode.

Reads and writes

Without question the `read` and `write` system calls are probably the two most used system calls in UNIX systems. We will describe how both work in this section. Both function by taking a file descriptor and a buffer. Using the file descriptor and the process number, we find the inode of the open file and the offset into the file. The operations will take place exactly here, there is no inode manipulation involved, only reading from and writing to disk blocks. The calls share a function called `read_write` in `src/servers/fs/read.c`.

This function is large and slightly confusing, since it handles both reads and writes for both block and character devices (and also pipes, but we will ignore that in our discussion). What happens is, roughly:

1. Check the given file descriptor and get the corresponding "filp" entry.
2. Convert the given memory address of the buffer from the process' virtual address space to a physical memory address and check whether it fully fits in the process' data segment.
3. Split up the transfer on block boundaries and call `rw_chunk` for each block. `rw_chunk` can only handle a single block and will read or write from disk by requesting the correct block from the disk cache, do the transfer to that block in memory and release its reference on the block.
4. Update the file size (if writing), update the file offset in the "filp" entry. If the transfer was successful we will update the timestamps in the inode.
5. We are done, return the appropriate return value to the user.

Let us now have a look at this process in more detail.

As said, we start by looking up the "filp" entry for the given file descriptor. If this entry does not exist, we were given an invalid file descriptor and return. Otherwise, the entry does exist and we check whether this file descriptor is indeed opened for reading and/or writing; if not we will also bail out. The number of bytes to read is also checked, when this is less than zero we return an error code. If this number is equal to zero, this is actually a valid call (but only when the given file descriptor is valid), but not really useful. We solve this by returning zero immediately (which means: zero bytes read successfully).

Some other checks are up next. Using the `umap` kernel call we convert the given address of the buffer (which is an address relative to the calling process' address space) to a physical memory address, that is: the exact location of this buffer in the physical memory. This operation also checks whether the buffer fits into the data segment, on failure an error will be returned.

Before we can properly split up a transfer on block boundaries, we need to know the block size. This block size can vary between the different kinds of devices we can operate on. If we are operating on a device directly, we have opened a character or block device by opening

its entry in the `/dev` directory. Recall that these devices are special files and we can figure out their types by reading the inode's `mode` field. For character and block devices we use the first zone field to check whether this is actually a device and obtain the block size from it. Otherwise we are just accessing a mounted file system and we obtain the block size from the super block.

We now know everything we need in order to initiate the transfer. For character devices, we just call the `dev_io` function here, this function is part of the device driver interface which we will discuss later on. For all other cases, most notably block devices and (block based) file systems, the problem is a little harder. When we are about to write to a file system, there is a possibility that the file size will change and we have to do some extra checks for this here: we check whether the file will not grow beyond the maximum allowable size; if the `O_APPEND` flag is set we need to append on each write and thus we have to set the position to the end of the file; and finally if we are about to write past the end of the file, a hole is about to be created between the end of the file and the new position, this data is still unwritten and there is left-over data from old files (possibly from other users!), because of security we zero-out this zone starting from the current end of the file.

The transfer is now started. We enter a loop that will continue until all bytes have been written or read. The loop's body does some arithmetic and bookkeeping so `rw_chunk` is only called with chunks that only span a single block. We will look into `rw_chunk` later in this section. We break out of the loop in the case of "EOF" or an I/O error.

After the loop, if we have written past of the current end of the file, we update the file size to be the current position of the file, whether an error occurred in the middle or not. Remember that even if an error occurs in the middle of the transfer loop, we still have written some new data to the file. In all cases the position field of the "filp" entry is updated.

If all operations were successful, meaning that there was no disk error and the user-space copying went okay, we will update the time stamps on the inodes and return a value indicating success. Otherwise, an error will be returned.

The purpose of `rw_chunk` is simple: read or write a part of a block, or a full block. The first thing that `rw_chunk` does, is determining on which block on which device to operate. When reading or writing directly to/from a block device, we just divide the position by the block size to get the block number, and the device information is stored in the inode. For file system operations we use the `read_map` function to get the block number corresponding to the given inode and position in the file.

Next, we have to obtain a buffer of that block in memory to operate on. Here we distinguish the following cases:

- We are reading from a non-existing block on the file system. Get a free buffer and zero it out.
- We are writing to a non-existing block on the file system. In this case we need to create a new block and hook it up to the inode, we use the function `new_block` for this. This function will take care of allocating a new block in the current zone if there is one left, or otherwise allocating a new zone and updating the inode and possibly indirect blocks. We will not go into detail about this zone bookkeeping code.
- Otherwise, when reading, we will just read in the required block. When the block is already in the buffer cache, a pointer to that block will be returned.

- If we are only going to partially overwrite a block, then we have to fetch the current block from the disk first. When we will overwrite a full block, we can write this new data to disk without having the load in the current data first.

If we do not have a valid buffer by now something is really wrong and we will trigger a kernel panic. Before continuing, another sanity check is done: if we are about to write to a new block past the end of the file on a file system, we clear this block with zeroes first for security reasons.

With a pointer to a valid buffer, the remainder of `rw_chunk` is very simple. We use a kernel call to copy the buffer chunk from the calling process' memory to this buffer or vice versa for reading. After this we are done and will release the block, with the block marked dirty if have just written to it.

Minix uses a technique called "read ahead" when reading from block devices, we left this out of the discussion above. Read ahead is triggered from two places in the file system server code: from `read_write` when reading from a file and when reading a block needed by `rw_chunk`. What we do in both cases, is read a little more blocks from the disk than necessary, with the assumption that a request to read these blocks from the disk will follow later. We can then satisfy that request simply by returning the buffer from the cache. Since Minix allocates blocks from zones in files, the full zone is read in when one of those blocks is requested.

This concludes our discussion of the Minix file system server. While we have not discussed the complete source code of the file system server in detail, the full picture of how the file system server operates should be clear now. It should not be hard to gain knowledge of the remaining parts of the file system server.

3.6 Device driver interface

Device drivers form a layer of abstraction on top of different pieces of hardware which have the same function. For example different kinds of disks have the same function: storing and retrieving blocks. But controlling a floppy drive is different than controlling a hard disk. A layer of device drivers can abstract this away by providing a generic interface for storing blocks and hiding the details of controlling the actual hardware. In this section we will have a look at how device drivers are implemented and work in Minix.

3.6.1 Overview of the interface

It will not come as a surprise that the communication between device drivers and the rest of the kernel takes place using messages. As we have seen in section 3.2.1 the kernel call mask allows device drivers to execute special kernel calls to be able to access I/O ports. The device drivers in Minix are all separate processes, and they share a generic message format. Included are commands like:

- Opening and closing a device.
- Reading and writing to/from a device.
- Scattered reads and writes, for transferring vectors of blocks.

- "`IOCTL`", for sending control commands.
- Cancelling operations.

There are some more messages that a device driver usually handles, like signals and pings from the reincarnation server (discussed in the next section). Drivers for network devices have a different set of messages to control them.

From the types listed above, it is obvious how the file server can use these to open devices and transfer blocks. Since all device drivers share the same interface and are separate processes, implementing the main message loop handling the same messages in each of the drivers would be quite tedious. To overcome this Minix provides a small library, which implements a skeleton of a device driver. This can be found in `src/minix/src/drivers/libdriver/`. New drivers can link against this to get their core functionality implemented. The most interesting part of this is the implementation of the message main loop. Each device driver initializes this library by providing a structure with pointers to the functions implementing the different "handlers" for opening a device, reading a block, etc. The driver library will invoke these functions when a corresponding request message is received.

3.6.2 Driver mappings

In a UNIX system devices are usually identified by their major and minor numbers. We need a way to map these major and minor numbers to a device driver. This mapping takes place in the file system server. Basically the file system server maintains a table where it maps a major number to a process number where that driver is running. When the file server starts up, a mapping is creating from the given major number of the device containing the root file system to the (hard coded) process number of the device driver. This device driver is selected and compiled into the boot image during compile time.

In the next section we will see that the reincarnation server can be used to start up new system servers using the user-space "service" command. This includes loading in new device drivers. When a new device driver is being started up, the reincarnation server will call the `mapdriver` function. This function will send a `devctl` message to the file system server, which will create a new mapping from a given major number to a given process number so this newly loaded device driver will be used. Analog to this, drivers can also be unmapped when they are unloaded.

3.7 Improving reliability

In the introduction we saw that Minix 3 has a small focus shift to reliability. We will have a short look at some specific attempts from Minix to improve reliability in this section.

3.7.1 Message passing and system call masks

Of course message passing has been with Minix since the beginning, but it is one of the more important features that improve the reliability. It makes it possible for the different kernel components to have their own memory space, so the entire kernel does not crash if one of the tasks corrupts the memory. Also the system is more secure because of the fact that a viral kernel task cannot "infect" the other ones. Secondly, the kernel controls the message passing between the different tasks and so it gets to decide what is allowed and what not. On Minix,

you can only do a kernel call with a `sendrec` operation. This is because when a task would only call `send` and never `receive`, the kernel would block at a certain moment, waiting for that task to call `receive` – an unwished situation.

Processes also have a so called "trap mask" that indicates which kernel traps a process is allowed to request. Another per process mask, the "call mask", is used to set which kernel calls (calls handled by the system task) a process is allowed to execute. For user-space processes this mask is always zero. For the different kernel tasks the mask can be set in such a way that the task can only request the kernel things that it really needs.

3.7.2 The reincarnation server

The reincarnation server is new in Minix 3 and is responsible for launching kernel services and drivers which are not contained in the boot image. After launching it will also keep an eye on those services by sending "ping" messages every now and then. When a services does not respond any more, the reincarnation server will restart the service in an attempt to resolve the problem. Eventually, if a service keeps dropping out, the reincarnation server will stop restarting the service. Since buggy drivers are separate (user-space!) processes, they are not able to bring down the system. With the reincarnation server drivers which crash every now and then will be automatically restarted and continue to function.

Most of these services are launched by init from `/etc/rc`. There is a small user-space program called "service" which is able to send requests to start or stop a given service to the reincarnation server.

Note that services included in the boot image (the process manager, file server) are not controlled by the reincarnation server. Right now these servers are considered "special" and when one of those processes crashes a kernel panic is inevitable. There are several problems why these processes cannot be automatically covered. One obvious one is that both the file server and process manager have very specific initialization code that must run in the right order. Both also store data crucial to the system, it will be very hard or close to impossible to recover this data after a crash of a component. Another issue is that these "core" services run with a fixed and pre-specified PID. The reincarnation server will have a need for a special kernel call to set PIDs back on these servers. They will also have to be loaded in the memory area that the kernel allocated at start up for placing these servers in. At last but not least the more practical issue, when the file system server or the main disk driver goes down there is no way to reload their binaries from disk.

3.7.3 The data store server

Some drivers have a bit of state information which they need after they have crashed in order to be able to fully restore the functionality they were providing. The data store server has been written for this purpose. It is running as a separate process and other services can store and retrieve data by sending messages to this server. Data is stored as an array of (key, data) tuples and keys have to be unique as they are used to access the data.

The implementation of this data store server does not seem to be fully complete in this version of the Minix source code. There are data fields and functions stubbed out which will provide more functionality when implemented. One of these features is storing a "secret" in addition to the key and data; a process accessing the key would probably have to provide the correct secret for the key in order to gain access. Right now all information seems to be public

for all processes. Another feature is to make it possible for a process to subscribe to a certain key. When done, the process will receive notification messages when a key has changed value.

3.8 Conclusions

In the preceding sections we have had a detailed look at the most important parts of Minix. We will end this chapter by listing some deficiencies found by reading the Minix source code.

3.8.1 A real micro kernel?

Throughout the book [8, pp. 42-43] Minix is said to be a true micro kernel, fixing the "mess" typically found in monolithic UNIX kernels like Linux, by design. And Minix indeed appears to be one, with the process manager, file server and other components running as separate processes in user-space. So, for the most part, yes, Minix is a real micro kernel, however there are some interesting workarounds to be found in the source code.

To start with the clock task. In fact, the clock task is a driver for the timer chipset found on most computers, providing an abstract interface for getting the current time and setting up timers. But, the clock task is not running in user-space as all other device drivers, it is actually running as a kernel "thread". Apparently, it has been placed in kernel-space because it only interfaces with the kernel [8, pp. 113]. A same argument could then be made for moving block device drivers into the file system server, since that is the only component interfacing with them. As argued in section 3.2.6, it is probably more likely that this has been done for performance reasons.

We did another interesting observation in the file server code. The process manager uses the `read` and `write` system calls to load in and save segments of a process. Because apparently first copying the memory chunks from the process to the process manager and then from the process manager to the file server takes too long, a little workaround has been added. Instead, they put the process number and segment in the upper 10 bits of the file descriptor. Using this, the file system server will access the calling process' address space to do the transfer. Tricks like this tend (and they do) make the implementation of basic system calls more messy, and can form potential security breaches too, completely undermining two of the basic arguments for using micro kernels.

Minix does not provide calls to fetch information about processes to the user-space processes. This poses a problem for the implementation of the UNIX `ps` command. In Minix this is "fixed" by exposing the memory area of the kernel to user-space via the `/dev/kmem` device that is only readable by a certain group. The `ps` has been marked with the set group ID bit, when the binary is run the group ID of the process will be set to make it eligible from reading the kernel's memory via the `/dev/kmem` device. `ps` then works by acquiring the data is needs directly from the kernel's data structures, the definitions of these data structures have all been copied into the source code of the `ps` binary. Next to being a disaster to maintain, this is a very bad thing to do security-wise.

3.8.2 Drawbacks of the micro kernel approach

Having a true micro kernel still appears to be less than ideal. Communicating with other kernel components by means of message passing gives a considerable overhead compared to doing a direct function call. However, if the message passing overhead only leads to a slight decrease in performance, it is fully justifiable by claiming that it improves reliability and

security. In Minix the overhead seems to be fairly high, which has led to the introduction of certain workarounds to improve the performance, as we have seen in the previous section.

Micro kernels seem to all suffer from the overhead introduced. Most commercials attempts to create (performant) micro kernels have turned into a new concept called "hybrid kernels" as we saw in section 2.1.3. Here the advantages of both micro- and monolithic kernels have been combined.

3.8.3 Areas which can be improved

Bringing real micro kernels to a comparable level of performance still needs research and experimenting. The same holds for creating a kernel without workarounds, which could potentially turn Minix to be a bigger "mess" than Linux. Of course, Minix has been kept small to be able to use it as a teaching tool. But, because of that, it also is a very viable testbed for doing easy experiments with micro kernels. Here, we list a few areas which would be potentially interesting to experiment with.

- Message passing is very slow, and blocking. For example, if the process manager has to load a segment of a process from disk, it messages the file system server and blocks until it gets a reply. If the file server needs to fetch a block from the disk, it will message the device driver and block until it receives the block. Now both the process, and file manager, two very critical components of the system, are blocked waiting on the disk.

  The blocking here occurs because of the usage of a Rendezvous approach to messaging. Replacing this with non-blocking message queues would be interesting, and probably increase the performance. Of course, it will also increase the complexity of the kernel.

- The reincarnation server is a very nice addition to Minix 3, which is able to restart crashed system servers. However, the servers (and device drivers) which are loaded from the boot image are not managed by the reincarnation server. One of the reasons seems to be that we need to have a way to reload the executable of the server when it has to be restarted. This could be solved by keeping a ram disk in memory, from which the kernel can reload the executables. Note, that when reloading the file and process manager (if even possible), there need to be a way to resynchronize the process table after system start-up.

- In the version of Minix we studied, the file server is really more than just a file server. It also manages device mappings, and includes an implementation of a file system on disk. Splitting this in, for example, a device mapping server and a file system driver would be interesting. Work to re-engineer the file server is already underway, as seen on the Minix website [10].

- Minix does not have support for paging and virtual memory. Adding this functionality would increase the complexity of Minix considerably, which is not wished. Though, to be able to run more complex software, a simple implementation of paging would not hurt. Currently, work is underway to add support for shared libraries and paging to Minix.

- Portability is about non-existent in current Minix. Minix currently contains code to run on x86-based systems and the 68000 architecture. The platform specific code for the

different architectures is currently separated in the source code using `#ifdef` macros. To ease porting to other platforms (if Minix wants to be a player on the embedded market, a port to the ARM architecture is a must), it would be helpful to have a nice abstraction layer for the different architectures. There would be separate directories with implementations of this abstraction layer for each supported architecture. According to the release notes of the latest Minix release [14], a split between architecture dependent and independent code has recently been created.

3.8.4 An option for resource-limited computers?

In section 3.1 we have discussed the new goal the Minix community is aiming at: becoming usable as a serious system on resource-limited computers. We do not think this will be realized in the near future because of the numerous performance and architectural problems. The file system server seems to be the main culprit and problems are in the process of being addressed. Minix will also have a need for a proper implementation of virtual memory and paging, otherwise it will be close to impossible to use the little physical memory in embedded systems efficiently and run complex applications.

Part II
Implementing disk block replication in Minix

4 Intelligent disk block replication

In this and the next chapter of this thesis we will use Minix as a "playground" for doing some prototyping. We will attempt to discover disk block access patterns in sequences of disk operations. From these patterns we can then distill groups of blocks that are often operated on around the same time. We can then replicate these blocks at another place on the disk, all next to each other, so they can be read from the disk all at once with a single read operation. When doing a read for a single block, some other blocks that were not explicitly requested have now also been prefetched into the disk block cache, with the expectation that they will be needed soon. This way we expect the disk I/O performance to increase, since if these prefetched blocks are indeed used later on we can satisfy those requests using the cache and we do not have to wait for a costly disk access.

The sequence of disk operations represents the usage of a computer system by a user. Our method for extracting usage patterns from this sequence is driven by finding pairs of blocks that are often accessed together. This is a first approach at finding such patterns and can be improved on in the future. We will use sequences of disk operations that have been generated with artificial patterns inserted as a means of verifying that the algorithm works.

We will write code, essentially in the form of extensions and additions to Minix, that will log all disk operations and save these to disk in a human readable format. This code will also be able to discover the access patterns and replicate selected disk blocks at another location of the file system. The file system server will be modified to support the logging of the disk operations, to able to replicate blocks on the disk and to read the disk blocks that have been replicated.

In addition to the Minix code, we will first write a couple of scripts to test the concept of the algorithm that we will use to discover disk access patterns. It is much easier to write such a script to test the algorithm and verify that it will improve things before implementing this in kernel-space.

We silently assume here that scattered disk reads are slower than contiguous disk reads. In order to verify this claim we will write a small test program to measure disk reads of both contiguous lists of blocks and blocks that are scattered over the disk. We do need to verify this, because if there is not much difference between contiguous and scattered disk reads, our efforts to group disk blocks will be of no use.

For testing and doing the measurements we will be using a floppy disk drive. Floppy disk drives are orders of magnitudes slower than hard disk drives and provide us with visual and audible feedback of the operations. Especially the speed difference will help us in determining if our changes improve the situation.

During the development of the scripts to test the concept of the algorithm, we have found that it is not possible to create an exact copy of the Minix caching code outside of the Minix kernel. That means that the results we will get from these experiments should not be considered final test results. The test results can only serve as an indication whether or not replication will have an impact, which is again an indication whether it is a good idea to go

ahead with the implementation of the idea in the kernel.

In this chapter we will look at some other measures that Minix currently takes to improve the disk I/O performance. Following that we will study the algorithm that we have designed and how we plan to do measurements so we can actually verify whether or not this algorithm improves the performance of disk I/O. Using a prototype implementation of the algorithm we will do preliminary measurements. In the next chapter we will explore the implementation of these ideas in Minix in detail and have a look at the results of the measurements.

4.1 Current performance measures

You can imagine that reading disk blocks into the cache block by block is unacceptably slow and operating systems are employing several methods to improve this. In this section we will have a look at two techniques Minix is using to attempt to increase disk throughput.

4.1.1 Zones

We have already seen in section 3.5.6 that Minix is using "zones" and not blocks to store data in. A zone consists of one or more consecutive blocks. The number of blocks per zone depends on the device and file system. When this number is bigger than one, files will be laid out on the disk in groups of disk blocks instead of fully scattered. This will result in performance improvements if the file is being read sequentially [8, pp. 554].

On the floppy disks we will use, each zone will only consist out of a single block. Each block and thus also each zone has a size of 4Kb. Because each zone only consists out of a single block, the concept of zones will not have any impact on the disk read performance.

4.1.2 Read ahead

The basic idea behind read ahead is: when we have to access the disk anyway, why not read more than we need? This happens on the assumption that the extra blocks that have been read will be used at some later stage. If this is the case, the blocks will already be in the cache saving a costly disk access. This technique is used a lot in operating systems. From the fundamental assumption you can imagine that there are a lot of different algorithms around for doing read ahead.

Minix has a not very sophisticated form of read ahead. When a block is read Minix will try to read between 32 and 64 extra blocks that immediately follow the disk block that was originally requested. This way, we hope to find more blocks that belong to the same file that is currently being read so that subsequent read requests further on in the file can be satisfied out of the cache. Of course this will work when zones are being used and the file has been written to the disk contiguously. In this case, when reading blocks that immediately follow, more of the same zone is being read and thus more of the same file. It is clear that this method will not help much with files whose blocks are all scattered over the disk. The Minix implementation of read ahead can be found in `src/servers/fs/read.c` in the function `rahead`. That this implementation of read ahead is suboptimal is also noted in a C comment in that source file. A better solution is suggested that would involve looking at available zone pointers and indirect blocks and read those disk blocks. This way we are sure that the extra blocks we read are actually part of the same file.

4.2 Our idea

What we will try to achieve in this thesis is basically a different take on read ahead. Where Minix is trying to read more of the same file in advance, we will focus to read blocks that are "related" in the sense that these blocks are often accessed together. By looking at sequences of disk operations we will try to determine which blocks are often accessed together or in a short time period. These blocks will then be selected for replication at reserved, contiguous, space on the disk.

When a disk block is requested, we already end up in Minix' read ahead function. There, we will check whether the requested block is one of the blocks that has been replicated. If this is the case, we will read this block and as many of the "related" blocks from the disk space where these blocks have been replicated. Otherwise, if the requested block was not replicated, we will proceed with the default way of doing read ahead in Minix.

By prefetching blocks that have been shown in the past to be correlated, it is very likely that these blocks will be used later on compared with just prefetching blocks from the current file position hoping for the best.

4.2.1 Contiguous versus scattered disk reads

We should note that we rely on the assumption that reading blocks contiguously is faster than reading a sequence of blocks that is scattered over the disk. To back up this assumption we have written a small test program that does raw disk reads. Using command line arguments the program can read a contiguous sequence of blocks or a scattered sequence. The source code of this test program can be found in appendix A. We ran this program on a test machine (an Intel Pentium II 350Mhz, with a default floppy drive) running Minix. We made it read several different amounts of disk blocks directly from the floppy device (`/dev/fd0`), both contiguously and scattered. Reading directly from the floppy device implies that we are circumventing the cache. In both cases we did 5 test runs, with a 4 second pause in between test runs. The Minix floppy disk driver turns off the floppy disk motor 3 seconds after the last disk operation has been completed. By having a 4 second pause in between test runs, we are assured that each timing will contain the initial overhead of starting the motor and positioning the read head.

The time the program needed to complete the requested operation was measured with the UNIX system command `time(1)`. This command reports the time elapsed since the invocation of the program, as acquired from the system clock with a granularity of tenths of seconds. This includes all time spent by the system (including all overhead from the operating system) to execute the program. Because we are testing on a single-user system and there are no other users that can influence the time that our processes need to complete, the `time` command is a good enough means to time our test program.

In figure 7 we observe that reading scattered blocks takes between a factor 2.2 to 3 more time than reading the same amount of contiguous blocks. A realistic number of blocks to replicate and that could be prefetched at once from a floppy disk is about 16 to 64 blocks. Also around this number, the difference in time to read contiguous or scattered blocks is factor of 2.2 to 3.

We have drawn trend lines for both data series in the graph to determine the initial overhead. For the contiguous case this uncovers a linear function between the number of blocks to read and the time that this operation takes to complete. If we interpolate this to

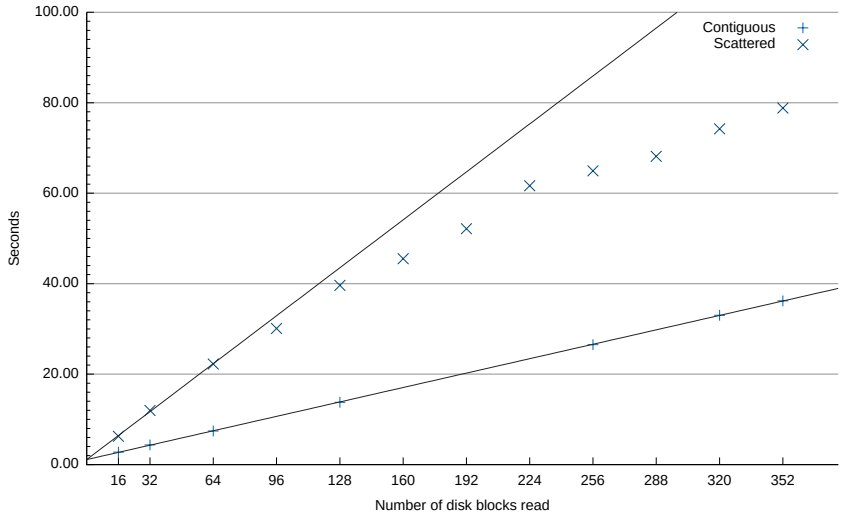

Figure 7: Seconds needed to read various amounts of blocks from a floppy.

the y-axis, we find an initial overhead of 1.1 seconds. The figures are different for the scattered case. The first few data points indicate a linear function, but after that the slope becomes flatter. We think this is due to the more blocks are being read, the larger the chance is for series of randomly picked block numbers to be on the same track. Such series require less track changes and thus less seek time. When we take the first few data points of the scattered graph and draw a trend line, we again see an interception at the y-axis around 1.1 seconds.

These results generally indicate a factor 2 to 3 speed up between contiguous and scattered reads. We can thus reasonably expect that our algorithm should have a visible impact on performance if it is functioning correctly.

4.2.2 Logging disk accesses

For discovering patterns in a sequence of disk accesses, we first need to be able to actually get these sequences of disk operations out of the system. We will create a new Minix server that will be able to log disk accesses that occur in the system. We name this server the "Optimization Server", or "os" in short. Next, some kind of hooks in the core workings of the Minix file system server are required to be able to get the information we need. To accomplish this, we will patch the Minix file system server to send a message to the optimization server about every block that is operated on. The optimization server receives these messages and writes the information they contain to a log file on the disk. Later on, the optimization server will also be responsible for finding correlated blocks and selecting blocks for replication.

If we would log all events on all devices, including all block operations that happen on the disk where the log file is located, we are stuck in a cycle: every time the optimization server writes a new log file entry, a new hard disk access will occur resulting in a new message to the optimization server. Since we have decided that we will do all measurements using the floppy disk drive, we have coded the file system server hooks in such a way that only the operations done on the floppy disk (this device has a major number of 2) are logged.

The part the file system server has in logging the disk operations is constrained to sending

information to the optimization server. All information the file system server has to send is readily available in the system and does not require complex routines to retrieve this. Therefore we believe the impact of the logging mechanism to the run-time performance of the file system server is for the most part dependent on the `send` routine. Because of the synchronous nature of the `send` primitive in Minix, this might put the file system server into a blocking state more often, waiting for the message to be sent. This can have a negative effect on its run-time performance.

We do log all disk accesses to the floppy disk drive and do not distinguish by which user these were done. For large, multi-user systems it might make sense to collect the data on a by user basis, so the patterns discovered are unique to a certain user. In this thesis we will focus on patterns based on log data from the complete system and we will only be doing disk operations with a single user. Another argument for doing things this way is that there Minix is in fact a full multi-user operating system, it is being used as a single user system for scientific research most of the time anyway.

There are two points in the file system server where disk operations can be sent off for logging. One point is to log disk operations where we actually send a request for a block to the disk driver. The other place is inside the `read` and `write` system calls and send the log messages from there. Recall from our discussion in section 3.5.7 that the `read` and `write` system calls have to operate on a block that is in the block cache,; i.e. read into memory. When a block is not yet in the cache it will be fetched from the disk and the former method of logging would send a log message to the optimization server. For writing, the former method of logging will only send off a log message if the block is finally written back from the cache to the disk.

Here we see the fundamental difference between both methods. Since the former method will only log those operations that are done on the disk driver, it will only log requests that are not satisfied by the cache. But, since it logs every access, it also logs accesses to the super block, inode tables and other file system metadata. The latter method logs all operations done by the user on a file. This excludes access to the file system metadata, since a user does not have access to this data via those system calls. It is clear that the former method will "miss" a lot of these disk accesses, since those have been satisfied by the block cache.

We have chosen to implement the latter method into the Minix kernel. Because this will give us detailed data of every operation the user has done without having been influenced by the block cache, it will allow for us to discover more detailed data access patterns than when using the former. This also guarantees us to give the same logging data when several test runs are done with the same test program. The former method does not guarantee this to us, since the state of the cache plays part in what will be requested from the disk.

Also, for our floppy disk cache, not taking the file system metadata blocks into account should not have a huge impact. Since the file system on a floppy is very small, the amount of file system metadata blocks is also small. As file system metadata we count the super block, inode map, zone map and the table with inodes. On a floppy disk this amounts to 5 disk blocks, or 1.4% of the disk. Most of these blocks are needed to read directory indexes, thus they will probably all end up in the block disk cache very soon. The gain that could be obtained by replicating these blocks is negligible.

4.2.3 Finding related blocks

In a list of disk operations that occurred subsequent in time, acquired as described in the previous section, we can try to discover blocks that are in some ways "related" to each other. We have devised an algorithm, that we will further on refer to as the "pair discovery algorithm", to achieve this that we will describe in this section and test in a later section.

The algorithm uses the concept of block "neighborhoods"; every block in the sequence has a neighborhood whose size is expressed in the number of blocks between the boundary of the neighborhood and the center – basically the block the neighborhood belongs too. A neighborhood with a size of three will consist of seven blocks; the center block and three blocks before and after the center block. By defining neighborhoods this way, we have symmetric neighborhoods. This means that, looking at figure 8, when block 207 is in the neighborhood of block 39, then 39 is also in the neighborhood of block 207. This does not have to be true for other definitions of neighborhoods. Our algorithm could work with different kinds of neighborhoods after a couple of subtle changes. An example is to not look at blocks around a block, but only blocks that follow after the center block. These other types of neighborhoods do probably perform differently and will not be discussed in this thesis, but could be subject of further research.

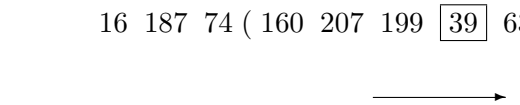

Figure 8: Sample sequence demonstrating neighborhoods

Behind the definition of neighborhoods lies the main idea of this algorithm: if in the different neighborhoods of the same block (we assume that this same block appears multiple time in the disk operation sequence) certain blocks appear frequently, they seem to be related to each other. These blocks are then candidates for replication.

Keeping this in mind, selecting blocks for replication is not very hard, it boils down to finding block pairs that are related to each other and then selecting those pairs that occur most often. In a nutshell, what the algorithm has to do is iterate over the sequence of disk operations and catalog the pairs of disk blocks that are found in all visited neighborhoods. Furthermore, we maintain a count of how often we have seen the pairs in the sequence. We end up with a data structure that contains all of this data. In this data we can easily find pairs of blocks that occur often; simply sort the pairs by decreasing frequency of occurrence. We can then select pairs of blocks for replication, starting with the pairs with the highest frequency. The amount of pairs that can be selected depends on the amount of blocks that are available to store replicated blocks. Pairs that have only been seen once can of course barely contribute to an increase in I/O performance. Therefore we will only replicate blocks that have been seen together for at least a given amount of times – we call this amount the threshold.

Conceptually the algorithm described above is simple, but we are left with one problem. When we are iterating over the sequence of disk blocks that are being operated on, when do we stop and decide to replicate the blocks with the highest frequency? There are several possibilities, we could replicate after an explicit command has been given by the user; or after a

certain time interval or after processing a certain amount of disk blocks (possibly for example as many blocks as fit in the cache). During our tests we will have to repeatedly mount and unmount the disk we are testing to assure that none of the blocks will stay in the disk cache, affecting our measurements. Because of this, we will be replicating the blocks when the disk is being unmounted. When a test program is being run on the same disk multiple times, we should be able to see a trend towards a shorter execution time of the test program. The other suggested ways of solving this problem could be evaluated in further research.

We have written a proof of concept implementation of this algorithm in Perl, which can be found in appendix C. In the remainder of this section, we will go over this algorithm in detail. First, as we have also seen in the brief explanation of the algorithm above, we have to create a data structure that contains all block pairs that are found in the neighborhoods when iterating over the list of disk operations and their frequency of occurrence. To simplify the code a little, we first read the complete sequence of disk operations from the standard input into an array called `blocks`. We can now easily iterate over this array and do the required counting, which is exactly what is done next.

What we want to know is the frequency of occurrence of all block pairs we have found in the neighborhoods. To easily and effectively keep track of this, we need quite a smart data structure. In the Perl code, this is done in the array called `accesses`. This array is indexed by disk block number. Each of the subscripts of this array contains a hash table. The keys (its indices) of the hash table are block numbers and the values coupled to those keys are frequency counts. The overall structure of this data structure is shown in figure 9. What we can store now is the frequency of occurrence of a certain block (the index into the array) with another block (the index to the hash table).

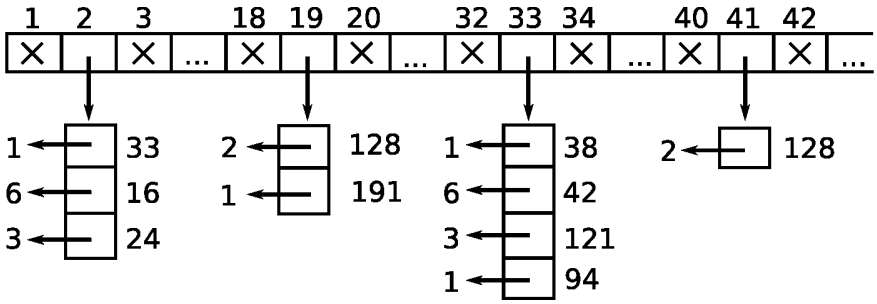

Figure 9: Overview of the "accesses" data structure. Note that not for each subscript a hash table has been drawn.

This is exactly the point where we have to be careful. As we said we use the index of one block for the array and the index of the other for the hash table. We have to create a consistent rule that says which block number we will use as index into the array and which block number for the hash table, otherwise the frequency counts we end up with will be incorrect. If we have seen block B in the neighborhood of block A, it will be registered in the hash table under the array index of block A. Vice versa; if we have seen block A in the neighborhood of block B, it will be registered under the array index of block B. The total frequency of occurrence is both of these counts added together. In order to deal with this correctly right from the start, we have decided to always use the smallest block number of the pair as index into the array.

Because of the symmetric nature of our definition of neighborhood, this is not a problem. We do lose the information that says in whose neighborhood we have seen a certain block. Since we are only looking for pairs right now, we do not consider this to be a problem.

Now that we have defined how we will store the data extracted from the sequence of disk blocks, we are ready to look at the code that will actually fill this structure. We simply iterate over all the blocks in the `blocks` array. For each of these blocks we have to look at its neighborhood. We do this by calculating the `start` and `end` boundaries of the neighborhood. After calculating these, we check if these do not overrun the bounds of the `blocks` array. Then we simply iterate from `start` to `end` over the `blocks` array and increase the count that the two blocks have been seen together. As we have argued above, we use the smaller block number as an index into the `accesses` array and the other block number as a key for the corresponding hash table.

After we have iterated over all blocks, we are done with this "initialization" phase. Our `accesses` data structure now contains occurrence counts of all block pairs that have been seen together. The next step is to "sort" all these block pairs by decreasing frequency of occurrence. We do this by using another data structure that is indexed by frequency and for each of these frequencies points into the `accesses` data structure. The idea here is that for a giving frequency of occurrence, we can find corresponding subscripts into the `accesses` array that contain block pairs with the given frequency. This data structure is called `count` and is again a hash table. Its subscripts contain an array with block numbers, which are the indices into the `accesses` data structure.

Filling the `count` data structure requires a simple iteration over `accesses`. For each block pair, we register the index into `accesses` as an entry under the frequency of the block pair in the `count` table.

Finally, we can select the block numbers that we want to replicate. We have seen that we want to iterate over the block pairs that we have found in decreasing frequency of occurrence. The `count` hash table makes this very easy for us, we simply get all of its keys and sort these in decreasing order. We then iterate over the entries in the `count` table; for each subscript of `count` we iterate over the block numbers we find in there. These block numbers are used as subscript to access the `accesses` array, in the resulting hash table we will search for entries that match the current count number. The matching entry number and the number that served as index into `accesses` form a block pair and are selected for replication. We continue this until the number of wished replication candidates has been reached or until the current frequency we are looking at is below the given threshold. We use a special way of bookkeeping here to make sure that each block is only added to the candidate list once. A single block can of course have multiple pairings with other blocks.

The maximum number of blocks that we can replicate is bound by the space that is left on the disk. We also have a lower bound, if we cannot replicate at least 15% of the disk space that is currently in use there is not much point in doing replication because the number of replicated blocks will be too small to achieve gains by reading these blocks contiguously. However, how much exactly of the available disk space we will use for replicating blocks is not yet clear and hard to say. We will be doing experiments with different numbers.

4.3 How to do measurements?

We will be doing measurements with 800Kb of random data placed on a floppy disk. This data is spread over 10 files, with an equal length, that we will write out incrementally adding random amounts of new blocks each time. By not writing out these files contiguously, this means that the different blocks of the files are scattered over the disk. Each of these 10 files is 80Kb large and thus consists of 20 Minix disk blocks, given the block size of 4Kb that Minix uses on floppy disks. In total we have 200 blocks available to operate on.

On these files we will be performing sequences of disk operations. The blocks we will operate on are chosen randomly, but we will also insert artificial patterns. The reason for inserting patterns is to be able to see more activity at certain places on the disk. Our algorithm should be able to find such places with more activity if it functions correctly and thus they function as a means to verify that the algorithm does what we expect it to do. The patterns we will use are artificial in the sense that we want them to reflect sequences of disk operations as done by real software. We will write and use a "pattern generator" that can generate these sequences of disk operations and insert several patterns with a given density. The pattern density is the percentage of the resulting sequence of disk operations that is generated from a pattern instead of randomly. The higher this pattern density, the more patterns can be found in the generated sequence of operations. On these sequences, which are exactly like the sequence in figure 8, we will apply the algorithm written in Perl as explained in the previous section. We do this to verify that the algorithm is indeed able to distill blocks that make up the pattern from the sequences of random disk operations.

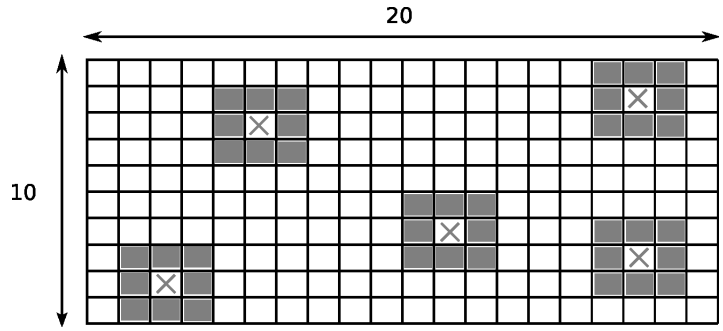

Figure 10: Area with 5 points chosen and blocks in their radii in gray. Here a radius of 1 is used.

The measurements will be carried out using three different patterns. The first pattern is simple, basically the same sequence of subsequent disk blocks is inserted every now and then. This sequence are the numbers from 0 to 9. In practice this means that each time this pattern occurs we access the first half of the first of the 10 files. Such a sequence basically reflects a program reading the same file that has been placed contiguously on the disk a few times during operation.

Secondly, we have a pattern that is a little more complex. Imagine a 2-dimensional area, such as the one in figure 10. Before we start generating the block sequence, we will pick 5 points in this area, determined by x and y coordinates. Each time when a pattern has to be inserted, the algorithm chooses one of these 5 points. From a given radius around this point, we will select 5 points that will each be mapped onto one of the 200 available block numbers

with a simple formula and then put at the end of the sequence. We will end up with a list of random blocks where points based around these 5 selected points will repeatedly occur. This reflects a program doing random binary access on a large file, an example of such a program can be a program manipulating a database or table on disk.

Finally, the last and third pattern is basically a variation on the second one. Instead of using a 2-dimensional representation of an area to select blocks from we will generate 5 lists each containing 10 random blocks beforehand. The algorithm will end up working in the same way, instead of selecting one of the 5 points, we will just select one of these 5 lists and then select 5 blocks from that. Compared to the second pattern, the blocks will appear to be more randomly distributed with this version of the algorithm.

Before we will implement the algorithm in the optimization server and do tests with Minix on the actual hardware we will be testing our idea with the proof of concept implemented in Perl. We will be feeding this algorithm with random sequences of disk operations, as generated with the pattern generator algorithm explained above. Since our algorithm is deterministic, we only have to run the tests once; different runs of the test program should not yield different results. We will be counting the number of disk operations that are needed with and without replicated blocks. For this to work correctly, we need to know exactly when a block has to be read from disk. In the Minix file system server this is determined by the state of the LRU (least recently used) list, as we have seen in section 3.5.3. The implementation of read-ahead also plays a role here, since the blocks we are interested in might have been read ahead. This means that we will have to simulate the block eviction policy of the Minix cache and the read-ahead algorithm in our concept implementation. If our algorithm works as expected and has a positive impact, we should be able to see a decrease in disk operations required. And later on, with the implementation in the kernel, this means we should be able to see a decrease in time.

Once we have verified that our algorithm has the effect we expected, we will implement it in our optimization server on Minix. With this implementation we will be able to do the actual timings on real hardware. These timings will be done on both the sequences as generated by the pattern generator but we will also do some timings with lists of blocks that are completely random, without knowingly inserted patterns.

For both the concept and real implementation, we have to decide how large the Minix cache should be. By default, on 32-bit Intel architectures, the Minix cache has a size of 1200 blocks. When a 4Kb block size is assumed, the cache size is about 4800Kb. Our floppy disk of 1440Kb will fit into this cache multiple times, provided that the blocks from other disks have all been evicted from the cache. We thus might have to shrink the cache in order to do proper measurements. We will be experimenting with different cache sizes in our proof of concept implementation.

4.4 Results of measurements with the proof of concept implementation

The proof of concept implementation actually consists out of two different things. The first part is the algorithm that will generate a list of replication candidates, as described in section 4.2.3. This is what will actually be implemented in the "optimization server". Additional replicated blocks will then be read by the Minix file system server when applicable. Whether or not replicated blocks should be read into the cache fully depends on the Minix caching code, just like with read-ahead. In order to get figures of disk reads required with and without

using replicated blocks, we have to write simulation code that can simulate the workings of the Minix cache as we have argued in the previous section.

We have written such a simulator in Perl, as can be found in appendix D. The simulator has a few command line options to be able to simulate different scenarios that can affect the total number of disk reads required. The most important are the setting of the cache size in blocks and whether replication is enabled or not. Once running, the simulator will iterate over a list of blocks and maintain the state of the cache and the LRU chain. As we have seen in section 3.5.3 Minix uses an LRU chain to keep track of blocks that can be evicted from the cache once they have not been used for a while. A correct simulation of the LRU chain is key to getting proper disk read counts.

During development of the simulator, we have also tried to implement the read-ahead code of Minix to be able to test how the read-ahead and replication code would perform when used simultaneously. When a block that is being read is not replicated, you could then fall back to doing read-ahead instead. It appeared to be impossible to precisely implement the same behavior of the read-ahead routines outside of the Minix kernel. The main problem is that the number of blocks to prefetch is determined at run-time and depends on several things such as the contents of the cache, the operation that is currently being done (read, write or seek) and the length of the file that is being operated on. The maximum number of blocks that is being read ahead will never exceed the free space that is left in the cache, and the actual number of blocks that will be read depends on the disk device driver. For the floppy disk, the disk driver will refuse to read ahead blocks that are not on the same track as the requested block. Finally, the Minix read-ahead code will never request more blocks than the remaining length of the file, from the position that is currently being read. Thus, without also keeping track of detailed file system information it is not possible to exactly simulate the behavior of the read-ahead algorithm and the number of blocks that would be prefetched would almost always be guessed wrong.

The normal disk cache algorithm itself can also not be exactly simulated. In Minix, as we have seen in section 3.5.3, a block is only put on the LRU chain once `put_block` is called. The call of `put_block` fully depends on the operation that is being done again; something we do not keep track of. Instead, we just discard the block immediately, which is what happens most often. We also do not make a distinction between blocks that will be put on the front or tail of the cache, mainly because we do not keep track of metadata blocks (these will never be replicated). Furthermore, the disk write counts are not of much use, since we cannot keep track of whether a block is dirty or not – and only dirty blocks actually get written back to disk. Since each write requires the block to be in the cache (and thus to be read) we are mostly concerned with read operations anyway.

With this test setup, many different variables can be set. We will briefly list all of these:

- **Pattern density**. The pattern density as used for generating the testing sequence. We have varied this from 15% to 65% with increments of 5% and skipping 60%.
- **Neighborhood size**. This is a parameter of the pair discovery algorithm and controls the size of the neighborhoods as seen in figure 8. Tests have been done with the following values: 3, 5, 7 and 9.
- **Candidates**. Another parameter of the pair discovery algorithm that controls how many blocks may be selected for replication. The threshold value, as explained in

section 4.2.3, is for now fixed on 5. Here, we have been using the values 8, 16, 32, 48 and 64.

- **Cache size**. The size of the cache in Minix disk blocks as used by the cache simulator. Cache sizes we have been simulating with are 100, 125, 150, 175 and 200. Keep in mind that all of our test data (all 10 files) can fit in a cache with a size of 200 blocks.

All testing was done with a sequence listing of 2000 blocks and with of course both replication enabled and disabled. Read-ahead has not been taken into account as explained above, since the implementation of this outside of the Minix kernel was very troublesome.

At first, we have been running the block sequences as generated by the pattern generator through the pair discovery algorithm. If the pair discovery algorithm fails to find corresponding blocks (i.e. those blocks that have been inserted on purpose during pattern generation), then the replicated blocks will have a very small chance of improving the performance. For all of the three patterns described above, the pair discovery algorithm indeed succeeded to find correct correlations that made sense. All of these appeared high on the candidate list, so the blocks that are going to be replicated are relevant.

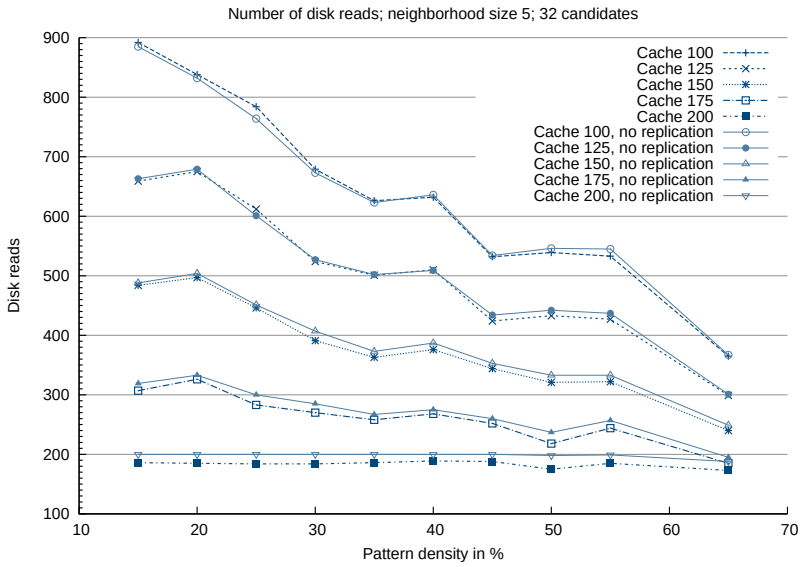

Figure 11: Results of testing with the "contiguous blocks" pattern with different pattern densities

From these observations one would say that an improvement in performance should be noted. However, the first simulation runs were slightly disappointing. With varied densities, neighborhood and cache sizes there was only an very slight decrease in disk reads between with and without replication, as can be seen in figure 11. We also observed that the size of the neighborhood appears to have little influence on the number of disk reads. These first tests were done with the "contiguous blocks" pattern. In the next test runs, we decided to vary the number of replication candidates – the amount of blocks that may be selected for replication. This also did not appear to have much influence; in some cases selecting more candidates even had a negative effect.

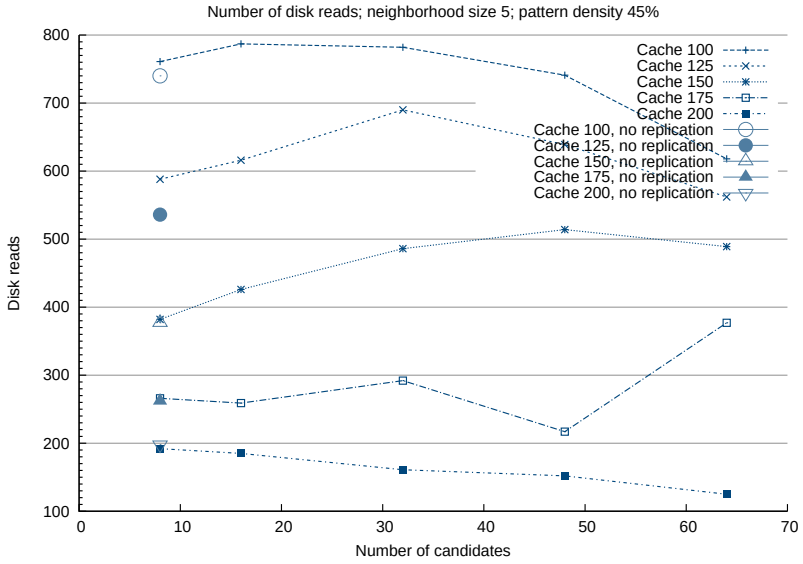

Figure 12: Results of testing with the "common blocks" pattern with different numbers of candidates

We did the same set of tests with the second pattern. This turned up even more disappointing results; all results with replication enabled were worse compared to those with replication disabled. The graphs also showed very unexpected behavior. We can see in figure 12 that the graphs for the different cache sizes are all very different. Compared with the numbers for replication disabled, the results with replication enabled are on average worse. Since we had verified that relevant blocks were selected, we felt that caching thrashing might be going on. Cache thrashing occurs when the cache is almost full and we have to evict some blocks from the cache in order to make space for some other ones. However, the blocks that have just been evicted are requested very soon afterwards, causing these blocks to be fetched from the disk again (very costly!). In our case we suspect that by reading in replicated blocks we had to evict blocks from the cache that were apparently more relevant than the replicated blocks. Since these blocks had to be read from the disk once again, this gives a decrease in performance.

When revisiting the implementation of the cache simulator, we actually found two issues in the simulator's LRU chain handling. We re-ran all tests after fixing these issues and completely different results came out. When replication is enabled, we actually clearly see positive changes in performance for all cases. Compare figure 11 with figure 13 and you can observe that the decrease of disk reads in the latter is bigger. Since the neighborhood size and number of candidates did not seem to have much influence, we have only done these tests with a neighborhood size of 5 and 32 replicated blocks. The tests have been done for all three patterns.

A few different modes of operation have been identified that can be used when a replicated block is found. These all have a different effect on the test results:

1. Read only this replicated block that has been requested.

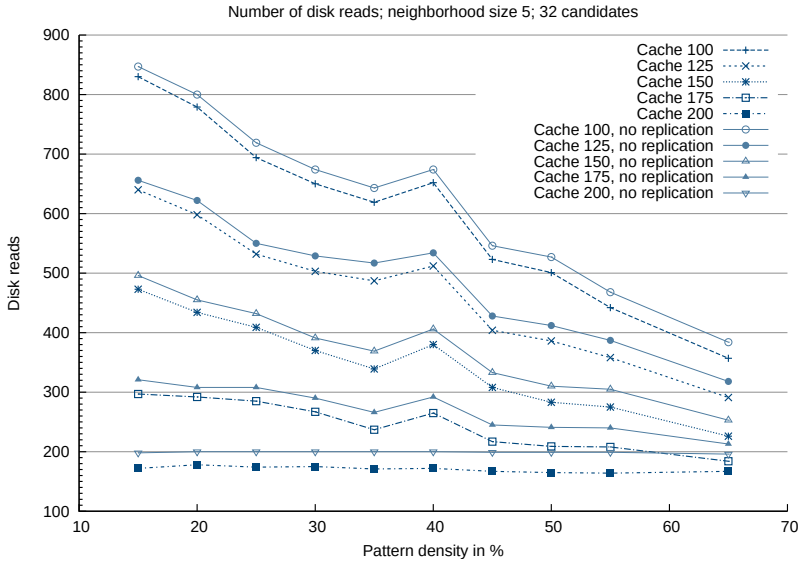

Figure 13: Results of testing with the "contiguous blocks" pattern with different pattern densities using the fixed cache simulator

2. Read all blocks that have been replicated on the disk into the cache. The number of blocks that will be read equals to the number of replication candidates the algorithm was allowed to select.

3. Together with the replicated block that has been requested, also read additional replicated blocks that have been placed before and after this block. We do this until we get to a block that is already in the cache.

If we really want to exploit the fact that contiguous reads are faster than scattered reads, option 1 is obviously not the way to go. The two other options both read multiple blocks in one go. The main difference between option 2 and 3 is that cache thrashing can occur when using option 2. There is a real danger that we will replace very relevant blocks in the cache, that were not replicated, with replicated blocks. Also, we might read blocks from the disk that were already in the cache, possibly wasting time. But because we have seen a factor 3 speed-up between contiguous and scattered reads this might be well worth the cost.

Another option that can be set on the cache simulator is whether or not to enable the write-back of blocks that are in the cache but have not been used for some time (in other words the blocks that have been least recently used). This is another measure to avoid cache thrashing. When we are reading blocks into the cache, we have to write back the contents of (dirty) blocks if the cache is full in order to get an empty buffer. With write-back enabled this will indeed happen. What can occur now is that again relevant blocks are already evicted from the cache and possibly written back to the disk in order to load a replicated block that might or might not be used. This can be disabled and then replicated blocks are only loaded if there are empty buffers in the cache. Of course, once the cache has been filled up no extra replicated blocks in addition to the requested one will be read. Now the replicated blocks basically only serve as a way to quickly fill an empty cache but play no role afterwards. In figure 14 we

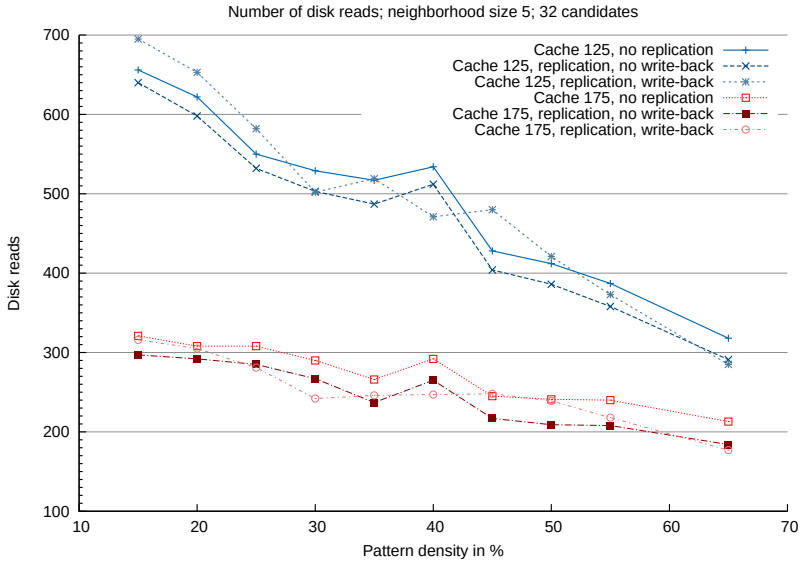

Figure 14: Comparison between running the tests with and without write-back enabled

can see that the results with no write-back are better on average; this means that we can probably better not write-back blocks from the cache to make space for new replicated blocks.

One other issue is that the replicated blocks are written on the disk in a non-specified order. Depending on which of the three modes of operation described above is used this can be troublesome. Especially with larger disks mode 2 will not be feasible, as we probably have hundreds of replicated blocks. We want to use mode 3 in such a case and make sure that the blocks that are placed before and after the requested block are really heavily related – the order of the replicated blocks on the disk should be optimal. Right now, any replicated block might be there.

Still, mode 2 could be used for a form of amortized analysis. Once a replicated block is not in the cache, we will read all replicated blocks from the disk, this is a worst-case operation. We expect many of the next requests to be satisfied from the cache. This way the worst-case operation only occurs once in a while and using amortized analysis we can determine the average-case performance which gives us a means to compare mode 2 with the performance of the other modes.

In future research the algorithm could be further improved to find out the optimal ordering of the replicated blocks on the disk. Such an optimal ordering would imply that groups of blocks that are often accessed together are written to the disk next to each other; this is basically a form of clustering. We can find such groups by creating a graph with all blocks that have been accessed, for each pair of blocks our algorithm has found we create an edge in the graph with as weight the amount of times this pair has been seen. In the graph we probably want to discard edges with a weight below a certain threshold. This way we expect to get several graphs that are disconnected from each other and each of these graphs then forms a group that can be optimally ordered.

How exactly to retrieve the best order to save the blocks of the group to disk is not

immediately clear. We would have to generate and analyze a few of these graphs in order to find out. There might be certain structures in the graphs that we are not aware of right now and there might be interesting differences between graphs that resulted out of tests with different patterns. A few elementary rules can already be given. It is clear that two blocks connected by an edge with very high weight are very related with each other and should thus be ordered immediately next to each other. The edges with a high weight can serve as a starting point to order blocks that should be immediately next to each other. For further ordering we could start with the "cores" of each graph: the node with the most high weighted edges. Immediately around this block the two blocks connected with edges with the highest weights, and so fort. Once more: the best way to do this should be determined by analyzing a couple of these graphs.

The goal is to come up with an ordering of the replicated blocks that will help when doing contiguous reads of replicated blocks – blocks that are very related to each other are now always next to each other and will be picked up together.

We have proceeded with writing an implementation of this algorithm and file system server code for replicating the selected blocks in the Minix kernel. The next chapter of this thesis will deal with the details of this implementation, as well as measurements and test results. The test data we have collected in this chapter is superfluous, when testing on hardware we need constrain the variables of the experiment. In the graphs we have seen a last sharp decrease when going from a pattern density of 40% to 45%, after that the decline stagnates. We also consider that using a larger pattern density than 45% is not good enough of a test for the algorithm that has been devised. We will therefore be working with a pattern density of 45%.

The differences between the different neighborhood sizes do not seem to be that large, we have decided to use a neighborhood size of 5, which is the average of what we have tested with. In the several candidate counts we have tested with we have found mixed results. Generally a candidate count of 32 appears to be on average. For a floppy drive containing 360 blocks, we consider a number of replicated blocks of 32 (which is 10% of the floppy's capacity) to be a reasonable number. Using these settings for the algorithm's parameters, we will be doing measurements with several different cache sizes and patterns.

We will be using the same patterns as with the simulator when testing on hardware. To start with, we will not be writing back blocks and when a replicated block is found we will use mode 3 as argued above. At first we will disable Minix' read-ahead functionality, later on we will also test with it enabled. If the first results using these parameters are not satisfying, we will evaluate the situation and change the parameters as necessary. In addition we will also do a test run with a fully random data set using the best parameter set that we found in running tests with actual patterns embedded.

5 An implementation of disk block replication

A working implementation of the algorithm explored in the previous chapter has been written for the Minix kernel. This includes the required changes to the file system server to be able to store and read replicated blocks on the disk. In this chapter we will go through the code of this implementation just like we did with the Minix code in chapter 3. After that we will have a look at how we have done the performance measurements on real hardware and analyze the results. We will conclude this chapter with a couple of recommendations for future research.

5.1 General overview of the implementation

As we have already seen in the previous chapter, the implementation of disk block replication in Minix consists of two parts. One part is the "optimization server" that does the logging of disk blocks and runs the algorithm to select a given number of candidates from the disk blocks that have been seen so far. The other part are patches to the file system server to support the optimization server. These patches include sending messages to the optimization server for each block that has been read from the disk and actually replicating a list of blocks at another place on the disk. Of course the file system server has also to be made aware of the fact that there could be replicated blocks on the disk that it can take advantage of. In this section we will first look at the newly written optimization server. After that, we will explore the modifications that have been done to the file system server. We conclude by explaining how the file system server and the optimization server communicate with each other.

5.1.1 The optimization server

The optimization server is a pretty small server. We will start by looking at the main file of its implementation, which is `src/servers/os/main.c`. This file is responsible for doing all communication with the other parts of the system. In the `main` function, we see the main message loop that we have also seen in most other Minix servers. The message loop supports all messages that the optimization server can receive; these messages are all defined in `src/include/minix/com.h` as usual. The purpose of all the messages will become clear in the remainder of this section.

Before the message loop is entered, the server has to be initialized. As with most other Minix servers this happens in the function `init_server`. After installing a signal handler for catching UNIX messages, we send a message (`OS_UP`) to the file system server to notify that we are online and ready to receive notifications of blocks that have been operated on. When logging is enabled by defining `ENABLE_LOGGING` at compile time, we call `log_open` that will open a log file at a fixed location (currently `/root/os.log`).

When the server receives a terminate signal, it will exit. The exit is handled through `exit_server`. This function will first notify the file server that the optimization server is going down through the `OS_DOWN` message. When logging is enabled, the log file will be closed. Finally, we exit.

The file system server has been patched to send an `OS_BLOCK` message each time the file system server has operated on a block. We will discuss the file system server part of this in the next section. In the optimization server the `do_block` function is called each time an `OS_BLOCK` message is received. This function is quite simple, if logging is enabled we will call `log_write` to write a log entry for this disk access to the log file. If the operation that was

done was a read operation, we will call `process_block` that will put the block number on the "to do" list for the pattern discovery algorithm.

The implementation of the pattern discovery algorithm can be found in `src/servers/os/discover.c`. At the top of this file we see the parameters that can be set to control the algorithm, after reading section 4.4 these should be familiar. The remaining part of the algorithm is a kind of "black box" and fully abstracted away. It is only publicly accessible through two functions: `process_block`, that puts a given block number on the queue; and `select_candidates`, that will cause the algorithm to select candidate blocks for replication and return these. The optimization server only calls these two functions to communicate with the algorithm. It has been done this way to make it very easy to "plug in" other implementations of such pattern discovery algorithms. As long as other implementations have these two functions defined in exactly the same way, it is trivial to use it to replace this pattern discovery algorithm.

Let us look at the remainder of `main.c`. The remainder deals with the other two messages the file system server can send: `OS_GET_CANDIDATES` and `OS_GOT_CANDIDATES`. Not surprisingly, these messages deal with getting the array with selected blocks for replication to the file system server. The first message is handled by `do_get_candidates`. This function will call the `select_candidates` function in the algorithm implementation, that will select blocks for replication and return a memory allocated array containing the block numbers of the selection. We then send an `OS_CANDIDATES` message to the file system server containing a pointer to this array and the number of elements in that array. The file system server can then copy this array to its own address space using the data copy calls implemented by the system task. Once the file system server has copied this array, it will send an `OS_GOT_CANDIDATES` message, which will cause the optimization server to free this array.

We now return to the most interesting part of the optimization server, which is of course the pattern discovery algorithm. The concept algorithm written in Perl, as discussed in the previous chapter, has been re-implemented in C. The basic idea remained the same and therefore it is useful to first read section 4.2.3 before continuing with this brief description of the C implementation. To simplify the explanation of this source code, we will split it into two parts and discuss both separately. We have seen that the discovery algorithm is controlled through two functions, which are exactly the two points where we can easily split up the code: one part handling incoming blocks to process and doing the frequency counting; another part selecting the candidates from the data structure containing the pairs and their frequency of occurrence.

In the Perl concept implementation we had the "luxury" that we knew the entire sequence of disk operations in advance. In the Minix kernel itself, this is of course not the case, blocks come in one by one via the `process_block` function. We could have created a dynamically growing array to store all these blocks and process this entire sequence once the file system server asks for a list of replication candidates. This may require a lot of costly re-allocations (and move operations) of the array. Also, we would have to cut off this array at some point to avoid it to infinitely grow. And finally, keeping the full array in memory is wasteful because it is possible to process the disk operation data as it becomes available. Therefore, the C implementation uses a kind of sliding window, which does not require us the keep the entire sequence in memory. Neighborhoods are analyzed as block numbers come in.

The sliding window has a fixed size, defined as `WINDOW_SIZE` at the top of `discover.c` to be twice the size of the neighborhood plus one. We will denote the neighborhood size as *nsize*

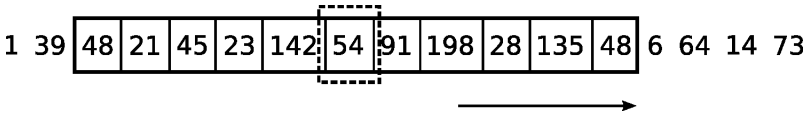

Figure 15: A sample sliding window, only the numbers inside the rectangle are in memory

further on. As we can see in figure 15 this is exactly enough to fit the entire neighborhood and the center block in memory. The `process_block` function adds block numbers to the window and advances the position. Once the window is full, we drop the first subscript of the window, move all other subscripts one position to the left and add the new block at the now free position at the right. After a block has been added, we call `analyze_window`.

In order to be able to explore the neighborhood for the first block, we need to have at least $nsize + 1$ blocks available in the window. The first block of course does not have any blocks to its left and its neighborhood contains the $nsize$ blocks directly to its right. Before the window has $nsize + 1$ blocks, `analyze_blocks` will bail out. Once these blocks are available, it will call `walk_window_for_position` that will walk through the neighborhood for the given block adding any relevant block pairs through `add_pair`. At some point the index that we pass in to `analyze_blocks` will stay at upper bound of the window array. Then, `analyze_blocks` will analyze the neighborhood for the center block in the window array each time `process_block` has been called. Usually, we never really know when we have reached the "end" of the sequence of block operations. If we indeed hit the end of the sequence, we can choose to explore the neighborhood for the $nsize$ last blocks in the window array. This can be done by calling `analyze_window` with the done parameter set to one. We could say that the end of the sequence is reached when the disk is unmounted, so we would call `analyze_window` this way from `selection_candidates`. For now, we will not be doing this. Since our neighborhood size is only 5 by default we will only miss the information of 5 operations, this has very little impact on the results anyway.

The data structures that we use to store the data of the disk block pairs we have found in the disk operation sequence is more or less the same as in the Perl implementation. We have a linked list called `accesses` that is built out of `BlockInfo` structures. Like the `accesses` hash table in the Perl implementation, it is indexed by block number. Each `BlockInfo` structure has an index and another linked list containing neighbors that have been found and how often (the frequency count). This second linked list is called `neighbors` and built out of `NeighborInfo` structures. The `add_pair` function takes care of creating these data structures and if the pair already exists in this data structure it will increase the count of occurrence.

At some point the file system server will call `select_candidates`. We will then transform the `accesses` data structure into a data structure that is indexed by decreasing count. Basically, we will create a new linked list, `count`, and move all `NeighborInfo` structures from `accesses` to `count`. This `count` list is created in `create_count_array`. We iterate over the `accesses` list and for each element over its list of neighbors. If a neighbor (each neighbor has a `NeighborInfo` structure) has a count that is higher than the defined threshold, it will be added to the `count` data structure using `count_info_add_neighbor`. This function will insert the `NeighborInfo` structure just like is done in `accesses`, but instead of indexing on block number, it is indexed on frequency of occurrence. In addition to that, the resulting `count` array is kept sorted on the frequency of occurrence.

Once done we have a data structure that is exactly in the form we need to select candidates for replication. This is now easily done by iterating over the count array. For each `CountInfo` structure, we will walk over all `NeighborInfo` structures in the blocks list. As long as we have not reached `MAX_CANDIDATES`, the maximum number of blocks we may select, we call `add_candidate` to add the block pairs to the allocated array that we will return to the file system server. Before actually inserting the given block number in the array, `add_candidate` will first check if the block is already there. If this is the case there is nothing we have to do and just return. When we have reached the maximum number of candidates to add or the end of the `count` data structure, we will free all memory that we have in use and return the array to the file system server.

This concludes our discussion of the "optimization server". One may ask why the optimization server has been developed as a stand-alone server and has not been fully integrated with the file system server code. This is actually a valid question, since it is very closely tied with the file system server and a lot of communication has to be done. The reasons are two fold, we did not want to "pollute" the file system code and separate out as many as we could. And because the optimization server is stand-alone, when it would crash due to a bug in, for example, the pair discovery algorithm, only the optimization server would go down and not the file system server (which usually ends up in the entire system going down). Another reason is that the file system code will do nothing related to disk block replication if the optimization server is not running. So by turning the optimization server on and off it is trivial to turn the replication mechanism on and off.

It is worth noting that unlike the other servers in Minix the optimization server is not compiled into the kernel. After running `make` you have to run `make install` to install the optimization server on the local file system; by default it is installed in `/sbin`. Once installed, the optimization server can be launched by the command `service up /sbin/os`. The server can by stopping by looking up its pid using `ps` and then run `service down <pid>`.

5.1.2 Modifications to the file system server

In the introduction we have seen that we have multiple patches to the Minix file system server. We will go over the different patches that have been produced one by one.

Disable read-ahead

We have a need to disable the read-ahead functionality in Minix. This need started in the very early stages of development where we did not want the read-ahead code to influence the blocks that were going to be logged. Since we started logging the blocks directly in the `read` and `write` system calls, this is no longer a problem. However, we still want to refrain the read-ahead code from polluting the cache with irrelevant blocks, because this might have an impact on the performance of our replication mechanism.

The code that disables read-ahead can be conditionally compiled in the file system server by using the `DISABLE_READHEAD` define; adding `-DDISABLE_READAHEAD` to the `CFLAGS` in the `Makefile` will enable this patch. When enabled, the patch disables calling `read_ahead` in the main loop of the file server, in `main.c`. Furthermore, in `read.c` the `rahead` function is patched by removing all code that calculates how many blocks to read-ahead and sets up the buffers before initiating the vector read. Instead, we will only do a regular `get_block` call for

the block that has been explicitly requested.

Floppy cache limit

In section 4.3, we have argued that we might have to shrink the cache in order to properly do the measurements. We have devised a patch that can limit the amount of disk blocks from the floppy disk drive in the cache. Since all blocks in the disk cache are related to a certain device, it is trivial to keep a count of how many blocks a certain device has in the cache. We have chosen for this approach instead of shrinking the entire cache because we only want to limit the cache size for the device where we are replicating blocks. If we would shrink the entire cache, the small cache would also contain blocks from the hard disk in addition to floppy disk drive blocks. Then, the amount of floppy disk blocks in the cache is never fixed and depends on the state of the system. This also means that the results of the measurements depend on this.

The floppy disk drive cache limit patch can again be compiled in conditionally. This is done using the `ENABLE_FDCACHE_LIMIT` define. The limit of floppy drive blocks that may be in the disk cache is defined in `const.h`. In `cache.c` we have a static variable `fdcache_size` that will maintain a count of how many floppy disk blocks are currently in the cache. In the same file we also find the most important part of this patch. The `get_block` function has to take care of finding an empty or least recently used block buffer. Here, we special case this search code when a floppy drive block is requested. When the maximum floppy disk cache size has been reached we will force to look for a non-empty buffer that currently contains a floppy drive block and re-use that. Otherwise, we use the default Minix code that just takes the front of the LRU chain (regardless of whether that block is being used or not).

There are some other places in the code where block buffers are being validated (by setting the `b_dev` field to a device) and invalidated (by setting `b_dev` to nil). If the buffer is being validated for the floppy drive or being invalidated and was the floppy drive, we have to increment or decrement the `fdcache_size` variable respectively. These checks are found in `cache.c` and `read.c`.

Optimization server support

All other patching in the file system server is all related to support the optimization server. Once again, this support can be conditionally compiled in with the `ENABLE_OS` define. We can split this patch in three parts: sending disk operation logging information to the optimization server, writing replicated blocks to the disk and read replicated blocks from the disk.

As said before the file system server will only do work related to replicating disk blocks when it knows that the optimization server is active. We also need to know the endpoint of the optimization server in order to send messages to it. The file system server is informed of the state of the optimization server by the `OS_UP` and `OS_DOWN` messages that were also discussed in the previous section. In the file system server's main loop, defined in `main.c`, we handle these two messages by setting the `os_pid` variable to the endpoint or -1. All other code in the file system server uses this variable to check if the optimization server is up and running.

Disk blocks are logged directly from the `read` and `write` system call implementations. Both of these functions use `rw_chunk` for reading/writing the disk blocks. In `rw_chunk` we will notify the optimization server of the disk operation by calling `notify_os`. This function will

check whether the optimization server is running and if the disk operation is being done on the floppy drive. Otherwise, it directly returns. Recall from section 4.2.2 that we will get cycles if we also allow for hard disk operations to be logged. Next, it will properly fill the message structure of the `OS_BLOCK` message. The message numbers and the location of its member fields are all defined in `include/minix/com.h`. When done, it will try to send the message.

Since the optimization server might be blocking on the file system server at the moment of sending the message, this operation can always fail. Because of this, a simple queuing mechanism has been included to make sure none of the messages gets lost. After a failure (or if the queue is not empty, we want the messages to arrive in the correct order) we use the function `notify_os_queue_message` to queue the message. This will automatically maintain the queue, which is a simple linked list. We have another function, `notify_os_send_top`, that will send the message that is at the front of the list. This function is called after every main loop iteration, so if the queue fills up, the file system server will sporadically send messages from it after each operation it has finished. Only when the submission of the message was successful it will be removed from the queue.

When the file system is unmounted, we want to replicate disk blocks. The unmount code is in `mount.c`, in the function `unmount` we call `replicate_blocks` before the file system is synched and the floppy drive blocks in the disk cache are invalidated. Most other replication logic is in the new file `servers/fs/replicate.c`. The replication process consists of three steps: check if there is enough space on the device, reserve this space and copy the disk blocks and finally write a block to the file system that contains information on which blocks have been replicated. We will discuss each step in turn.

To simplify the implementation we have decided to replicate the blocks at the end of the file system. In the last block of the file system we will store the information block, which is a list of block numbers that have been replicated. The blocks preceding the information blocks are the blocks where the replicated blocks will be stored.

Figure 16: General layout of the file system including replicated blocks

Checking space is handled by `check_space`, which is mostly concerned with checking and manipulating the file system' zone map. Each data block in the file system (this excludes the metadata blocks) has an entry in the zone bitmap, telling whether or not this block is currently in use. What we have to check is if there are enough blocks unoccupied at the end of the file system to store the information block and all blocks to be replicated. The first thing the `check_space` function does is access the zone bitmap and check if this is indeed the case. If there are enough free blocks, it will change the zone bitmap and mark these blocks to be written as occupied and return successfully. In case the last block of the file system is already occupied, we will read this block and check if this is the information block. Just like the super block of a file system, the information block has a magic number at a pre-specified location that we can use to check if this is indeed a valid information block. We will return unsuccessfully (and the replication will not continue) if the last block is not an information

block. If this last block is indeed an information block and enough blocks preceding it are still free, we will go ahead with replication and update the zone bitmap.

After updating the zone bitmap, we are ready to copy the replicated blocks to their second home. This is handled by the `copy_blocks` function that will simply use `get_block` to get a block buffer, change the block number in the cache buffer structure to the replicated position and call `rw_block` that will write the block to the replicated position on the disk (`rw_block` will have to be declared public instead of static in `cache.c`). When done, we change the block number back to the original block and release it using `put_block`. It should be clear that once we know the block number of the last block of the file system, the locations of the replicated blocks can all be calculated.

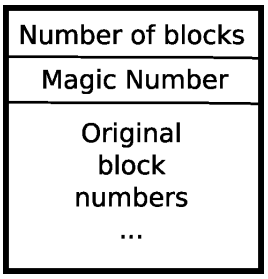

Figure 17: General layout of the info block

Finally, we can write the information block to the disk. This information block is actually very simple. It starts with a number indicating how many disk blocks have been replicated followed by the magic number (which has been chosen to be 0x00914470 for now). The original block numbers of the disk blocks that have been replicated follow in the same order as they have been written to disk by `copy_blocks`. `write_info_block` takes care of getting a buffer to the last block in the file system and writing this information to disk. When done the file system contains all disk blocks and further information we need to take advantage of the replicated blocks.

Now only a single part remains: we want to be able to detect if a file system has replicated blocks and if so, take advantage of those. A logical position to check if a file system has replicated blocks is when a file system is being mounted. In the function `do_mount` we will call `has_replicated_blocks`. The return value is saved in the `s_has_replicated` field in the super block (we have extended the super block structure in `super.h` with fields telling whether the file system has replicated blocks and fields to store the list of replicated blocks). The implementation of `has_replicated_blocks` can again be found in the file `replicate.c` and is fairly trivial. We first check whether the file system is on a floppy drive. If it is, we check if the last block of the file system is occupied and if this is an information block using `check_info_block`. The block is read from the disk and the array of replicated blocks is copied from the buffer to the `s_replicated_blocks` and `s_replicated_count` fields of the super block.

We now have all information in memory that we need to determine if a requested block from a file system is replicated or not. This information is used in `rw_chunk`. As we have seen the `read` system call reads all of its blocks through this function. If the operation is indeed a read operation, we will check the super block to see whether the file system has replicated blocks. When this is the case, we call `rreplicated`, otherwise we fall back to the default which

is calling `rahead` (remember that our patch disables read-ahead inside the `rahead` function, so we do not have to deal with it here).

Reading the requested block and any additional replicated disk blocks takes place in `rreplicated`. The implementation of this function is very similar to that of `rahead`, but instead of reading more blocks following the requested blocks, we read a couple of replicated blocks. It is checked whether the requested block (`baseblock`) is replicated by calling `is_block_replicated` (this function simply iterates over `s_replicated_blocks` to see if it is in there). If this is not the case, we again fall back to calling `rahead`. Otherwise we continue here and prefetch cache buffers that we will try to get filled with data using `rw_scattered`. For picking which blocks to read we use "mode 2" as explained in section 4.4. We iterate over the array of replicated blocks before and after the requested block, until we find a block that is already in the cache. When the floppy disk cache limit patch is enabled, we will also bail out once we have exceeded the amount of cache buffers available for the floppy drive – the write back of blocks or eviction of other blocks from the cache is thus not done.

We do prefetch cache buffers for the blocks we want to read using `get_block` and specify the *original* block number, not the block number where the block has been replicated. We want the block to be known in the system under the original block number, the replicated block does not exist for the other parts of the file system server. However, after getting the cache buffer, we will replace the block number in `b_blocknr` with the block number of the replicated block, so the data is actually being read from that position on the disk. After we are done getting cache buffers, we will call `rw_scattered`. This function will do a vector read of all requested blocks. The disk driver that is being used (for us the floppy driver) has the final call which of the requested blocks are being read. This means that there is a good chance that not all of the blocks will be read.

When `rw_scattered` is done, we once again iterate over the list of requested blocks and put the original block numbers back in the `b_blocknr` fields. To conclude, we call `get_block` to get the requested block – if this block was not read by `rw_scattered`, we are guaranteed that `get_block` will read it from the disk so a valid disk buffer is returned.

The main part of this code are the two loops that determine which blocks to read from the disk in addition to the requested blocks. A few changes can be made here to slightly change the behavior. We can choose to prematurely evict blocks from the cache (enable write-back) by removing the two if statements in the for-loops that check if the maximum floppy disk cache size has been read. If we want to force the file system code to read all replicated blocks we have requested instead of having the disk driver decide when to stop reading a little more changes are in order. Basically, we then will not use `rw_scattered` to read the blocks for us, but call `get_block` for each block ourselves. In the loops we first check if the block is already in the cache, if not we do get an empty buffer returned. We now release this buffer using `put_block` and do another non-prefetch `get_block` call that will read the data from the disk. Furthermore, this code is easily modified to read all replicated blocks from the disk ("mode 3") instead of limiting this by blocks that are already in the cache by changing `break` to `continue` in the if statement in the for-loops under the "Oops, block already in the cache" comment.

Although the optimization server patch to the file system server is not large in size, it is a complicated piece of code and not perfect. Let us iterate over some issues known as of writing:

- The size of the queue containing log messages pending to be sent to the optimization

server is not limited. When the optimization server goes down abnormally (due to the segmentation fault or similar), the file system server does not know the optimization server is offline and will continue to queue messages (because the `send` calls will continue to fail). This will eventually lead to the file system server to go out of memory and crash.

- The variables to the pattern discovery algorithm are compiled into the module, it might be nice to instead make these configurable through for example a configuration file in `/etc`.
- The size of the data structures in the optimization server are not limited and the server can therefore also run out of memory.
- All code is more or less floppy drive specific for now. The information block containing the block numbers of all replicated blocks can only be a single disk block in length. And the bitmap manipulation code can only handle zone bitmaps of one block in length. The number of replication candidates is thus bounded by these issues can may be too small for proper application to file systems on hard drives. Also, the zone bitmap manipulation code does not yet support shrinking the number of replicated blocks in the bitmap.
- When Minix `fsck` is run on a file system that has replicated blocks it will note that the blocks containing replicated blocks are in use from the state of the zone bitmap. However, it will not find any inodes that reference these blocks and therefore mark these blocks as being occupied but unused and ask to fix this. We can probably only properly fix this by patching `fsck` to also take the information block into account.

5.1.3 Communication between both servers

We will shortly summarize all communication that takes place between the file system server and optimization server in this section. In total there are six messages (all defined in `include/minix/com.h`:

- `OS_UP`; sent to the file system server by the optimization server to notify that it has been started and ready to receive logging messages.
- `OS_DOWN`; sent to the file system server by the optimization server to notify that it has been stopped and the file system server should refrain from sending any more logging messages.
- `OS_BLOCK`; a logging message that the file system server sends to the optimization server. In the message a couple of fields should be set: `OS_OPERATION` whether we are reading (`OS_OPERATION_READ`) or writing (`OS_OPERATION_WRITE`); `OS_DEV_NR` to the device number where the event occurred; `OS_BLOCK_NR` to the block number that is being operated on; `OS_PID` and `OS_UID` to the pid and uid of the process that is requesting this operation; `OS_INODE` to the inode of the file that is being operated on.
- `OS_GET_CANDIDATES`; when the file system wants to get an array with a list of replication candidates, it sends this message to the optimization server.

- `OS_CANDIDATES`; returned by the optimization server after the `OS_GET_CANDIDATES` message has been received. The following fields should be set in the message: `OS_CANDIDATES_POINTER` a pointer, relative to the address space of the optimization server, to the array of replication candidates; `OS_CANDIDATES_COUNT` to the length of that array.
- `OS_GOT_CANDIDATES`; sent by the file system server after receiving the `OS_CANDIDATES` message to tell the optimization server that the array with replication candidates has been copied and that memory can thus be freed.

5.1.4 Problems during implementation

During the process of writing the Minix kernel code, we have encountered some problems and eventually found a way to solve these. We have outlined these in this section so that future readers that will work with this code are aware of these issues.

The rendezvous nature of the Minix messaging system can lead to cycles in message chains. The deadlock mechanism that is present in the kernel does prevent the system from locking up. We have experienced problems early on to get all blocks the file system server operated on logged, since the optimization server may already be stuck in the `write` system call for writing to the log file when the file system server may be attempting to send a new log message to the optimization server. This results in a deadlock and without precautions (i.e. checking the return value of the call to `send`) the log message would be dropped. In the file system server this is solved by queuing the messages in case the call to `send` resulted in failure. These cycles in message dependency chains make it very hard to add another server to the kernel that interacts with core kernel components.

We have seen that the pattern discovery algorithm builds quite a large data structure. In the early stages the optimization server was very quickly crashing, later on also causing the file system server to crash. We have determined that is this caused by the servers running out of memory. In the makefiles of the servers the `install` command is called, that sets a maximum stack size on the binary. By increasing this stack size, the problems went away.

5.2 Performance measurements

With all code changes to the Minix kernel as described in the previous section we have done performance measurements to discover whether our goal of improving the disk read performance has been achieved. In this section we will first describe the conditions on which the testing has taken place. After that, we will present and analyze the results.

5.2.1 Testing conditions

All testing has again been done on a Intel Pentium II 350Mhz with a default floppy drive. As argued in section 4.4, we used the following parameters (please refer to section 4.3 for an explanation of these parameters):

- Neighborhood size of 5.
- Number of blocks that can be selected for replication: 32. The threshold for selection is a frequency count of 5.

- Floppy disk cache size limits (in blocks): 100, 125, 150, 175 and 200. Note that the second level cache we saw in section 3.5.3 is also disabled.
- Read-ahead is disabled.
- No more additional replicated blocks will be read if the floppy disk cache has been reached (write-back disabled).
- Pattern density of 45%. From the measurements with the concept implementation, the increase in performance seems to be relatively the biggest around 45%.
- We test with both the "contiguous blocks" pattern and the "common area" pattern. The sequences consist of 1200 blocks.
- The floppy disk contains 10 files of 20 blocks scattered over the disk as elaborated in section 4.3. The same floppy image has been used for all testing.

A test run consists of reading all the blocks in the generated sequence from the floppy disk. The test run is being executed by running a shell script that contains a list of `dd` commands that will read blocks from the disk. Test runs are timed using the `time(1)` command. An important difference here is that the default block size for `dd` is 0.5K, where the Minix block size is 4K. Because blocks always have to be read from the disk in full by the Minix kernel, all 4K of the block is read even if 0.5K is requested. In order to create a script to do the test run, we convert a given list of (Minix) blocks to read to a series of `dd` commands. `dd` can be instructed to read a given block by using its `skip` parameter, what we have to do is convert the given Minix block (in the range from 0 to 200) to one of the 10 files and one of the 160 `dd` blocks in the file. This is easily done by applying the formulas `file = block / 20` and `offset = (block % 8) * 8`.

Timing the full run-time of the shell script implies that the test results will contain the time that is required to launch all `dd` processes. Because we will launch equal amounts of `dd` instances for each test, this does not affect our ability to compare the resulting timings with each other. Furthermore, we do not exactly know what the effect is on the timings if we have the optimization server running. Recall that when the optimization server is running, the file system server will send a message for each (floppy drive) block that has been operated. The communication overhead introduced by this could be costly. In the measurements we have done we have not found any clear time difference. Also due to the stateless nature of the floppy drives (it cannot remember on which track the head is positioned) it is hard to make exact claims about this at this point.

For each cache size and pattern, we have done 2 tests with a non-replicated floppy image and 2 tests with the same floppy image but then the selected replicated blocks in place. After these 4 tests the original floppy image has been copied back to the floppy disk to start the new series of tests with a clean state. The mean of the two test results has been used in the graphs. We have to add that we did not properly account for the initial overhead as determined in section 4.2.1. Therefore all results can be off by plus or minus 1.1 seconds.

5.2.2 Test results

The tests have been run using the parameters as determined in the previous section, resulting in figures 18 and 19. Not surprisingly, the time needed to complete the test runs decreases

as the cache sizes grow. This is because more blocks can stay in the cache. We have also seen the same downward trend with the concept implementation. Interesting is that we see a improvement in time in all but the small cache sizes. We think there is a very good chance that for the small cache sizes we read a couple of replicated blocks in the beginning to fill up the cache. But because the cache is small, these blocks will be evicted from the cache to make space for relevant blocks before they have been used. Remember that since write-back was disabled during the testing, our replication code could not have caused cache thrashing.

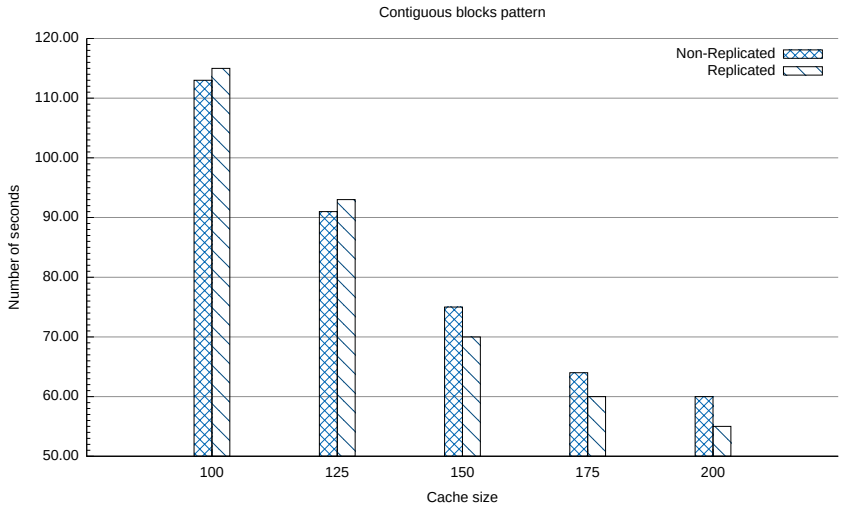

Figure 18: Results of measurements using the "contiguous blocks" pattern.

The speed-ups that we see for the cache sizes that are greater than 125 blocks are in the same league: about 5 to 7 percent. This is just slightly under what we expected from the simulations using the concept implementation. For the cases where we see a increase in time needed to complete the test run, the results are not as consistent. The tests with the contiguous pattern only took about a single second longer for the two smallest cache sizes, a difference of about a single percent. The common blocks pattern was clearly slower for the 100 blocks cache, but slightly faster with 125 blocks. We do not have a good explanation for this at this point.

In order to get some kind of idea why we do not see speed-ups with smaller cache sizes, we have done some additional test runs with different settings. We enabled the write-back of disk blocks, so that blocks may be evicted from the cache to make place for additional replicated blocks. For a cache size of 125 blocks this gave a very small speed-up for the contiguous blocks pattern compared to the original result – it is now on par with the time needed to read all blocks from a non-replicated file system. This slight speed-up was also observed for the common blocks pattern, bringing the speed-up at the same level as with the bigger cache sizes. A quick test with write-back enabled and a cache size of 175 blocks with the contiguous blocks pattern yielded the same results on a replicated file system as with write-back disabled; so the speed-up was still there. We can conclude that write-back does seem to have a slightly positive influence on the results here.

We have seen a way to force the file system server to read all replicated blocks we want to read instead of relying on the judgement of the disk driver. Using the common blocks

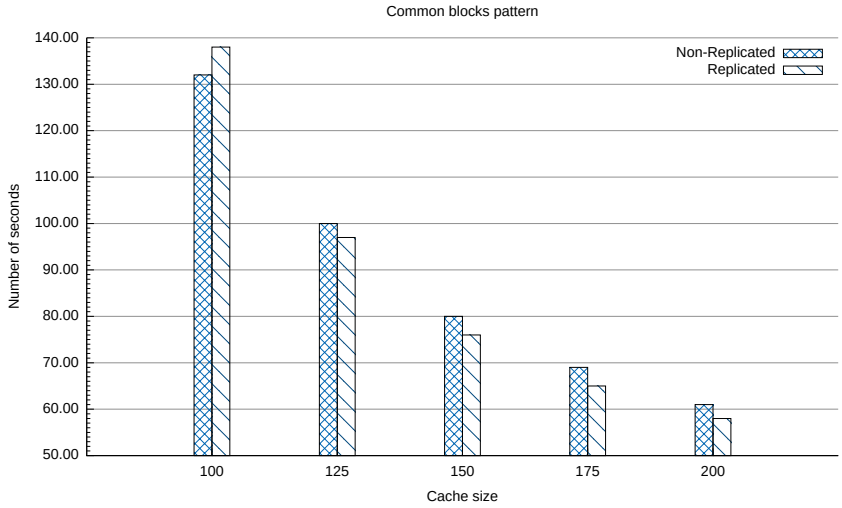

Figure 19: Results of measurements using the "common blocks" pattern.

pattern and the cache size of 125 blocks (we picked the common blocks pattern here because we wanted to investigate if we could even get a bigger gain in performance) we observed a big increase of time. Where the original results with a replicated file system were better than reading from a non-replicated file system, the results when "overriding" the disk driver show a increase in time of about 31%. Based on this result we decided that the disk driver is indeed in the better position to make a judgement when to stop reading blocks.

Another change that we have discussed is to read all replicated blocks that are on the disk instead of looking for additional replicated blocks to read left and right from the requested block until we find blocks that are already in the cache (mode 3 versus mode 2). We tested the effect of this change using the contiguous pattern and cache sizes of 125 and 175 blocks. Write-back was enabled during these tests, otherwise the change would very quickly not have effect because we could not evict blocks from the full cache. Since the disk driver can still limit the amount of blocks that will be read, it is very probable that it will never read all of the replicated blocks. For a cache size of 125 blocks, it performed slightly better than reading from a non-replicated file system. However, the decrease in time is not as big as we have seen with the "default" configuration at the beginning of this section. The tests done with a cache size of 175 resulted in slightly worse timings, but still more or less the same as the original timings.

5.2.3 Other tests

We are aware that the current algorithm is by no means perfect, but still we have done some other tests under different circumstances to see how the current algorithm would hold up.

In the previous section we have done timings with the disk driver overridden and the code modified to always read all available blocks separately. The former lead to much worse performance, the latter did perform slightly better in some cases. However, we noted that it is very probable that it will never read all of the replicated blocks. To see what the effect would be if we forced the system to read all of the replicated blocks (by overriding the disk

driver) we did a few test runs with both changes in place. So and we force `rreplicated` to read all replicated blocks and we override the disk driver to not bail out earlier. We only did a test with a cache size of 125 blocks and the contiguous blocks pattern. With replication enabled the system needed 139 seconds to complete the operations versus around 89 seconds without replication enabled. The system continued to read all blocks over and over again as expected and this took a lot of time. We suspect (this has also become clear by reading the source code of the floppy driver) that jumping from track to track on a floppy disk is a very costly operation and this is exactly the criterion the floppy disk driver uses to decide whether or not to stop reading. Usually it stops reading as soon as it has to switch to a new track.

The further tests deal with seeing what the performance is when the patches are used with Minix in "normal operation" – in other words without the floppy drive cache limit in place and with read-ahead enabled. With read-ahead enabled in addition to replication one could expect a further increase in performance as one feature can help the other. The test results in table 1 show a decrease for a small cache size and an increase for a larger cache size. It seems like that the replication code can be of help if the cache size is large enough. For the cache size of 125 blocks it is also possible that the read-ahead code was thrashing the cache. We later on realized that the read-ahead code was not bound by the floppy cache size limit. After changing the code to indeed obey to the cache limit we saw no difference in the timings. The timings for both cache sizes are slightly below the timings we saw earlier with read-ahead disabled.

	Cache 125	Cache 175
No replication	90.5 seconds	61.9 seconds
Replication enabled	92.7 seconds	58.2 seconds

Table 1: Tests with read-ahead enabled and the contiguous pattern

Next we tested what the effect would be if we disabled the floppy disk cache limit. The results are in table 2. The timings for no read-ahead are on par with the timings we have seen earlier with a cache size of 200, the largest cache size we have tested with. Since we have been testing with a data set of 200 Minix blocks, this does not come as a surprise. If we enable read-ahead and disable replication (essentially the default Minix configuration) we get a pretty fast timing – faster than all configurations with replication we have seen so far. We have to take into account here that reading unrelated blocks is not "punished" with cache trashing, because the cache is large enough. When replication is enabled we can cut another second off this timing. If the replicated blocks are stored in the proper order, as has been discussed before, there is a good possibility that the time might decrease more.

	With read-ahead	Without read-ahead
No replication	52.8 seconds	59.4 seconds
Replication enabled	51.9 seconds	55.4 seconds

Table 2: Tests with no floppy drive cache limit in effect and the contiguous pattern

All experiments so far have been done with a sequence of disk operations that has been carefully crafted by inserting patterns. Table 3 shows the results of experiments that have

been done with a fully random sequence. As all other sequences we have used, this one also consists of 1200 blocks. We see consistency with the other results so far, the timings with replication are faster than without and with read-ahead enabled things improve a little more. In table 2 we saw that the timing for just read-ahead enabled is faster than for just replication enabled. We see the same in table 3. It might be well possible that the performance for the replicated case lacks behind because it is bound by the number of blocks that has been replicated; in this case 32. The number of blocks the read-ahead code can read ahead is virtually unlimited. If we increase the number of blocks to be selected to 64 and test again (with read-ahead disabled) then we get a timing of 55.9 seconds – we outperform Minix' read-ahead algorithm in this case.

	With read-ahead	Without read-ahead
No replication	59.1 seconds	64.2 seconds
Replication enabled	56.1 seconds	60.3 seconds

Table 3: Tests with no floppy drive cache limit in effect and a random pattern of data

5.3 Conclusions and further research

Both the experiments with the concept implementation and the implementation in the Minix kernel show that it is certainly possible to achieve a gain in disk read performance using these methods. By applying the algorithm as described in this thesis on a file system that spans a full floppy disk, we were able to achieve a 7 to 9 percent gain in disk read performance. One important prerequisite is to have a large enough cache, for smaller cache sizes the increases in disk read performance disappeared. For these smaller cache sizes we have tried to get better performance by changing some of the parameters such as write-back and how many additional replicated blocks to read, however none of these changes seemed to have as large as an effect as we had wished.

One of the areas where our algorithm can be improved, as we have already mentioned in section 4.4, is how to order the replicated blocks on the disk. Especially for hard disks, where we cannot read all of the replicated blocks in one go, this ordering is important. The candidate selection procedure in the algorithm should get extra steps to determine a proper ordering. These steps would basically try to group, or cluster, the pairs we have found so far. For this clustering the original neighborhood information should be used. With such an order in place and read additional replicated blocks according to "mode 2", greater speed-ups might be achieved.

From the measurements we can also conclude that reading all replicated blocks that are on the disk does not make much sense. The disk device driver is much smarter and stops on time. When we override the disk driver and read all blocks we want on another manner, the runtime of the test scripts increases greatly. This seems to be caused by the need to move to other disk tracks, which appears to be very costly. It might be worth to read all replicated blocks that are on the same disk track as the requested block (even if already in the cache). Doing this correctly requires having knowledge of the disk's geometry, which is something only the disk driver knows at this point and only if advertised correctly by the drive. Properly implementing this thus seems to be a non-trivial task.

We have done all measurements with a pattern density of 45%. For such a high density,

the gains we have seen so far may be a bit on the small side. However, we have seen in the graphs showing the results of the concept implementation that the decrease in number of disk reads is more or less the same for the different pattern densities tested. Getting the same 7% gain for a pattern density of 5% is certainly not bad. In tests with completely random data and thus no patterns inserted we have also seen performance gains, certainly not bad for the first version of this algorithm.

The research done indicates that there are serious possibilities for achieving speed gains in this area. The first measurements done and in particular the code produced in this thesis should be good starting points for future research. Future research could include improving the ordering of replicated disk blocks, doing measurements with file systems stored on hard drives, and interchanging the algorithm developed in this thesis with another one – for example one incorporating techniques from neural networks or genetic algorithms.

6 Conclusions

In the first part of this thesis we have discussed some basics of operating systems. We have defined what an operating system kernel exactly is and we have globally looked at what tasks it has to take care of. In the remaining part we have closely studied Minix. Starting out by defining its overall structure, we continued by closely looking at each component. The code of each component has been briefly summarized. In some cases we felt it was necessary to go into more detail; sometimes making things a little more clear where Tanenbaum [8] was lacking.

After going through all components in detail, we have drawn some conclusions after reading the source code. Minix is claimed to be a true micro kernel and thus clean by design. We have identified a number of points where we feel this is not the case and Minix appears to work around some of the drawbacks that a micro kernel brings with it. The communication overhead is one of the major problems of micro kernels and research is still being done in this area. Most commercial deployments of micro kernels have turned into "hybrid kernels". We briefly summed up some areas that can be improved in the Minix kernel and saw that for most of these items work is already under way. Even though the focus of Minix has slightly been shifting from being a simple, educational operating system to a research operating system for resource-limited and reliable systems, it is still very suitable to study and do experiments with.

We explored the possibility of improving I/O performance by discovering patterns in disk operations in the second part of this thesis. The driving force behind this research is the fact that reading contiguous blocks is much faster than blocks that are scattered over the disk. A simple algorithm has been devised and shown to bring improvements to disk read performance in the form of less disk reads and thus a shorter time needed to complete the operations. Of this algorithm a simple proof of concept implementation in Perl has been written. This implementation has been shown to find relevant blocks in sequences of disk operations and, by simulating the cache, to indeed result in a lower amount of disk reads.

A possible implementation of this algorithm in the Minix kernel and the other code needed to make disk block replication on Minix file systems work has been written and described. Measurements done using this code have shown a improvement in I/O performance of about 6 to 7 percent. While mildly reasonable, considering that a pattern density of 45% was used this might be on the low side. These numbers do show us that it is indeed possible to achieve performance gains by replicating relevant disk blocks. We have done a number of suggestions for future research that could lead to an even greater improvement in performance. These suggestions include investigating the ordering of the replicated disk blocks (clustering) and improving or changing the algorithm. Any future research should be able to build on the theoretical and practical knowledge collected in this thesis.

Glossary

Context switch A switch from one process to another. This involves saving the entire context of the process and load the context of the next process to run.

Inode Short for index node. Each file is associated with an inode. Every inode has an unique number and it contains the attributes of the file and a table with all disk block numbers (for the Minix file system zone numbers) where the contents of the file are located.

Interrupt Hardware has the ability to send an "interrupt request" to the CPU. The CPU will receive such requests if interrupts have not been disabled (software can enable and disable interrupts on the CPU). In such a case the CPU will stop executing the current process (it is "interrupted") and handle the request of the hardware that sent the interrupt request first.

Kernel call In Minix, a call done between kernel processes. Such calls can only be done by kernel processes.

Kernel process In Minix, a process that is running in any of the protection levels belonging to kernel-space. This includes kernel processes running in user-mode.

Kernel task In Minix, a process running in the lowest protection level. Kernel tasks are not full separate processes as they all run in the same memory space; in fact they are threads.

Message passing Communication and synchronization primitive that works by sending and receiving messages.

Metadata blocks The blocks on the file system that contain information about the file system itself and not content from the actual files stored. This is information such as the super block and index tables listing the files stored on the file system.

Neighborhood size Controls the size of the neighborhoods used by the pair discovery algorithm. This can be visualized as a sliding window over the sequence of disk operations to process.

Pattern Density Parameter used by the pattern generator. The pattern density is the percentage of the resulting sequence of disk operations that should be generated from a pattern instead of randomly.

Process A program that is being executed on the system. A process includes the program code loaded from disk, state of the processor's registers, memory allocated to the process, etc. Multiple processes running the same program can be active at the same time, but they all have their own state and memory.

Replicated blocks Blocks that will be duplicated at the end of the file system.

RPC Remote Procedure Call. Allows for the ability to call a function that is located in another process that may also be running on another host.

Scheduling Operating systems employ a scheduler to determine in which sequence processes will be run. Several scheduling algorithms exist that each serve a different purpose.

Super block Generally the first block found on a file system. Contains important information about the file system's layout, such as the total size and where the inode of the root directory is located.

System call Interface for user-mode processes to the kernel. User-mode processes can make these calls causing a trap into the kernel. The call will be further executed by the kernel.

Zone (Minix) One or more consecutive blocks on a file system. This is a feature of the Minix file system that makes it possible to create bigger file systems without having to increase the size of the pointers to the disk blocks in the inodes.

References

[1] Apple Inc. *link(2) manual page*, August 2007. Mac OS X 10.4.10.

[2] M. Condict, Bolinger D., E. McManus, D. Mitchell, and S. Lewontin. Microkernel modularity with integrated kernel performance. tech. rep. *OSF Research Institute*, 1994.

[3] Ben Gras. endpoint_p - comp.os.minix. `http://groups.google.com/group/comp.os.minix/browse_thread/thread/6653fdf162e6190b/92e5adb46f576db8/`, November 2007.

[4] Jay Lepreau, Mike Hibler, Bryan Ford, Jeffrey Law, and Douglas B. Orr. In-kernel servers on mach 3.0: Implementation and performance. In *USENIX MACH Symposium*, pages 39–56, 1993.

[5] J. Liedtke. On micro-kernel construction. In *SOSP '95: Proceedings of the fifteenth ACM symposium on Operating systems principles*, pages 237–250, New York, NY, USA, 1995. ACM.

[6] G. Nutt. *Operating Systems*. Addison Wesley, third edition, 2004.

[7] A. S. Tanenbaum. Minix 1.1 readme. `http://www.minix-vmd.org/source/std/1.1/Floppies/Readme`, May 2007.

[8] A. S. Tanenbaum and A. S. Woodhull. *Operating Systems, Design and Implementation*. Prentice Hall, third edition, 2006.

[9] Author Unknown. Apple Developer Connection, The Kernel. `http://developer.apple.com/documentation/Porting/Conceptual/PortingUnix/additionalfeatures/chapter_10_section_8.html`, October 2007.

[10] Author Unknown. The Minix3 Operating System. `http://www.minix3.org/`, March 2007.

[11] Author Unknown. MS Windows NT Kernel-mode User and GDI White Paper. `http://www.microsoft.com/technet/archive/ntwrkstn/evaluate/featfunc/kernelwp.mspx?mfr=true`, October 2007.

[12] Author Unknown. POSIX. `http://en.wikipedia.org/wiki/POSIX`, May 2007.

[13] Author Unknown. The Linux source code, process.c. `http://lxr.linux.no/linux/arch/i386/kernel/process.c`, October 2007.

[14] Author Unknown. Release notes of MINIX 3.1.3. `http://www.minix3.org/download/releasenotes-3.1.3.html`, August 2008.

[15] A. S. Woodhull. Minix 2.0.4 downloads. `http://minix1.woodhull.com/mxdownld.html`, May 2007.

A Source code listing of read-blocks.c

```
1  /*
2   * read-blocks.c: Read contiguous or scattered blocks from disk.
3   *
4   * Copyright (C) 2008  Kristian Rietveld.
5   */
6
7  #include <stdio.h>
8  #include <stdlib.h>
9  #include <time.h>
10 #include <unistd.h>
11 #include <sys/types.h>
12 #include <fcntl.h>
13
14 #define MAX_BLOCK 360  /* 1440 / 4 = 360 */
15 #define BLOCK_SIZE 4096
16
17 static int
18 get_random_block (void)
19 {
20         int ret;
21
22         ret = ((float)rand () / (float)RAND_MAX) * MAX_BLOCK;
23
24         return ret;
25 }
26
27 static void
28 read_contiguous (int fd, int count)
29 {
30         int i;
31         int block = 64; /* We start reading at 64 */
32
33         for (i = 0; i < count; i++) {
34                 char buffer[BLOCK_SIZE];
35
36                 lseek (fd, block * BLOCK_SIZE, SEEK_SET);
37                 read (fd, buffer, 4096);
38
39                 block++;
40
41                 if (i % 10 == 0) {
42                         printf (".");
43                         fflush (stdout);
44                 }
```

```
45          }
46
47          printf ("\n");
48  }
49
50  static void
51  read_scattered (int fd, int count)
52  {
53          int i;
54
55          srand (time (NULL));
56
57          for (i = 0; i < count; i++) {
58                  int block;
59                  char buffer[BLOCK_SIZE];
60
61                  block = get_random_block ();
62
63                  lseek (fd, block * BLOCK_SIZE, SEEK_SET);
64                  read (fd, buffer, 4096);
65
66                  if (i % 10 == 0) {
67                          printf (".");
68                          fflush (stdout);
69                  }
70          }
71
72          printf ("\n");
73  }
74
75
76  int
77  main (int argc, char **argv)
78  {
79          int c;
80          int fd;
81          int scattered = 0, contiguous = 0, count = 0;
82          char *device = NULL;
83
84          /* Process command line arguments */
85          while ((c = getopt (argc, argv, "n:sc")) != EOF) {
86                  switch (c) {
87                          case 's':
88                                  scattered = 1;
89                                  break;
90
91                          case 'c':
```

```
92                                  contiguous = 1;
93                                  break;
94
95                          case 'n':
96                                  count = atoi (optarg);
97                                  break;
98                  }
99          }
100
101         device = argv[optind];
102
103         if (!device
104             || (scattered && contiguous)
105             || (!scattered && !contiguous)
106             || !count) {
107                 printf ("usage: %s -n count [-c | -s] <device>\n", argv[0]);
108                 printf ("   n: number of blocks to read\n");
109                 printf ("   c: read blocks contiguously\n");
110                 printf ("   s: read scattered blocks\n");
111
112                 return -1;
113         }
114
115         /* Open the device */
116         fd = open (device, O_RDONLY);
117         if (fd < 0) {
118                 perror ("open");
119                 return -1;
120         }
121
122         /* Operate */
123         if (scattered)
124                 read_scattered (fd, count);
125         else if (contiguous)
126                 read_contiguous (fd, count);
127
128         /* Done; clean up */
129         close (fd);
130
131         return 0;
132 }
```

B Source code listing of pattern-generator.pl

We do not list the full source code here, but only the algorithm that is of interest. The parts that have been omitted mainly deal with command argument parsing.

```
1  #
2  # Pattern inserters
3  #
4
5  #### Contiguous pattern
6
7  sub contiguous_pattern ($)
8  {
9          my $i = shift;
10
11         for (my $j = 0; $j < 10; $j++) {
12                 print "$j\n";
13         }
14
15         # Only increase with 9 because the for loop will increase with
16         # 1 already.
17         $$i += 9;
18 }
19
20 #### Common areas pattern; selects blocks from radius around point
21
22 my @common_areas;
23
24 sub common_area_initialize ()
25 {
26         # Select five fixed center points
27         for (my $i = 0; $i < 5; $i++) {
28                 my $x = int (rand (20));
29                 my $y = int (rand (10));
30
31                 my @point = ($x, $y);
32
33                 push @common_areas, \@point;
34         }
35 }
36
37 sub common_area_pattern ($)
38 {
39         my $i = shift;
40
41         # Select one of the five areas
42         my $area = int (rand (5));
```

```
43
44          for (my $j = 0; $j < 5; $j++) {
45                  # We select a point from the area with a radius of 2.
46                  my $dx = int (rand (4)) - 2;
47                  my $dy = int (rand (4)) - 2;
48
49                  my $x = $common_areas[$area][0] + $dx;
50                  my $y = $common_areas[$area][1] + $dy;
51
52                  $x = 0 if ($x < 0);
53                  $x = 19 if ($x >= 20);
54
55                  $y = 0 if ($y < 0);
56                  $y = 9 if ($y >= 10);
57
58                  my $block = $y * 20 + $x;
59
60                  print "$block\n";
61          }
62
63          # Only increase with 4 because the for loop will increase with
64          # 1 already.
65          $$i += 4;
66  }
67
68  #### Common areas pattern version 2; selects blocks from randomly
69  #    generated areas.
70
71  my @common_areas2;
72
73  sub common_area2_initialize ()
74  {
75          # Generate five sequences of random points (9 per seq.)
76          for (my $i = 0; $i < 5; $i++) {
77                  my @new_sequence = ();
78
79                  for (my $j = 0; $j < 10; $j++) {
80                          my $point = int (rand ($max_block));
81                          print STDERR "$point\n";
82                          push @new_sequence, $point;
83                  }
84
85                  push @common_areas2, \@new_sequence;
86          }
87  }
88
89  sub common_area2_pattern ($)
```

```
90  {
91          my $i = shift;
92
93          # Select one of the five sequences
94          my $area = int (rand (5));
95
96          for (my $j = 0; $j < 5; $j++) {
97                  # We select 5 points from the chosen sequence.
98                  my $x = int (rand (9));
99                  my $block = $common_areas2[$area][$x];
100
101                 print "$block\n";
102         }
103
104         # Only increase with 4 because the for loop will increase with
105         # 1 already.
106         $$i += 4;
107 }
108
109 # Initialization; we initialize the pattern generators if required
110
111 if ($pattern_type eq "common") {
112         &common_area_initialize ();
113 } elsif ($pattern_type eq "common2") {
114         &common_area2_initialize ();
115 }
116
117 # Engine
118
119 my $pattern_inserted = 0;
120
121 # Below we decide per insert whether we have to insert a pattern or
122 # a single random block.  Here, we first calculate how many % of the
123 # inserts should be patterns.
124
125 # divided by pattern_size (10 for contiguous pattern)
126 my $pattern_size;
127 for ($pattern_type) {
128         if (/cont/) {
129                 $pattern_size = 10;
130         } elsif (/common$/) {
131                 $pattern_size = 5;
132         } elsif (/common2$/) {
133                 $pattern_size = 5;
134         } else {
135                 die "Unknown_pattern";
136         }
```

```
137 }
138
139 my $pattern_inserts = $probability / $pattern_size;
140 my $pattern_percentage = ($pattern_inserts / ($pattern_inserts + (100 - $probability
        ))) * 100;
141
142 for (my $i = 0; $i < $blocks; $i++) {
143         if (int (rand (101)) < $pattern_percentage) {
144                 for ($pattern_type) {
145                         if (/cont/) {
146                                 &contiguous_pattern (\$i);
147                         } elsif (/common$/) {
148                                 &common_area_pattern (\$i);
149                         } elsif (/common2$/) {
150                                 &common_area2_pattern (\$i);
151                         }
152                 }
153                 $pattern_inserted++;
154         } else {
155                 printf "%d\n", int (rand ($max_block));
156         }
157 }
158
159 print STDERR "Patterns␣inserted:␣$pattern_inserted\n";
160
161 exit 0;
```

C Source code listing of discover-pairs.pl

We do not list the full source code here, but only the algorithm that is of interest. The parts that have been omitted mainly deal with command argument parsing.

```
1  # Read in the list of the blocks that are being operated on.
2  my @blocks = ();
3
4  while (<>) {
5          chomp;
6          push @blocks, $_;
7  }
8
9  # Now for each block see which blocks were also accessed.
10 my @accesses;
11
12 for (my $i = 0; $i <= $#blocks; $i++) {
13         my $start = $i - $neighborhood;
14         my $end = $i + $neighborhood;
15         my $thehash;
16
17         # Check bounds.
18         $start = 0 if $start < 0;
19         $end = $#blocks if $end > $#blocks;
20
21         if (not defined ($accesses[$blocks[$i]])) {
22                 $accesses[$blocks[$i]] = {};
23         }
24
25         for (my $j = $start; $j <= $end; $j++) {
26                 next if $blocks[$i] == $blocks[$j];
27
28                 if ($blocks[$i] < $blocks[$j]) {
29                         $accesses[$blocks[$i]]->{$blocks[$j]}++;
30                 } else {
31                         $accesses[$blocks[$j]]->{$blocks[$i]}++;
32                 }
33         }
34 }
35
36 # Find block pair with most common accesses.
37 my %counts;
38
39 for (my $i = 0; $i < $#accesses; $i++) {
40         foreach my $j (keys %{$accesses[$i]}) {
41                 my $count = $accesses[$i]->{$j};
42
```

```
43                 if (not defined ($counts{$count})) {
44                         $counts{$count} = ();
45                 }
46
47                 push @{$counts{$count}}, $i;
48         }
49 }
50
51 # Select pairs with highest counts as candidates for replication.
52 my %tmp;
53
54 foreach my $i (sort {$b <=> $a} (keys %counts)) {
55         last if $i <= $threshold;
56
57         foreach my $j (@{$counts{$i}}) {
58                 $tmp{$j} = 1;
59
60                 last if scalar keys %tmp >= $candidates;
61
62                 foreach my $q (keys %{$accesses[$j]}) {
63                         if ($accesses[$j]->{$q} == $i) {
64                                 $tmp{$q} = 1;
65                                 last if scalar keys %tmp >= $candidates;
66                         }
67                 }
68
69                 last if scalar keys %tmp >= $candidates;
70         }
71
72         last if scalar keys %tmp >= $candidates;
73 }
74
75 # Print selected candidates to standard out.
76 foreach my $i (keys %tmp) {
77         print "$i\n";
78 }
```

D Source code listing of simulate-minix-cache.pl

We do not list the full source code here, but only the algorithm that is of interest. The parts that have been omitted mainly deal with command argument parsing.

```
1  ###### HELPERS ##############################################
2
3  sub clear_cache ($)
4  {
5      my $i = 0;
6      my $cache = shift;
7
8      for ($i = 0; $i < $cache_size; $i++) {
9          @$cache[$i] = -1;
10     }
11 }
12
13 sub init_lru_chain ($)
14 {
15     my $i = 0;
16     my $lru_chain = shift;
17
18     for ($i = 0; $i < $cache_size; $i++) {
19         @$lru_chain[$i] = $i;
20     }
21 }
22
23 sub in_cache ($$)
24 {
25     my $i = 0;
26     my $cache = shift;
27     my $block = shift;
28
29     for ($i = 0; $i < $cache_size; $i++) {
30         if (@$cache[$i] == $block) {
31             return $i;
32         }
33     }
34
35     return -1;
36 }
37
38 # Remove given position from the LRU chain.
39 sub rm_lru ($$)
40 {
41     my $i = 0;
42     my $lru_chain = shift;
```

```
43      my $cache_position = shift;
44
45      for ($i = 0; $i < $cache_size; $i++) {
46          if (@$lru_chain[$i] == $cache_position) {
47              last;
48          }
49      }
50
51      @$lru_chain = @$lru_chain[0 .. $i - 1, $i + 1 .. $cache_size - 1];
52  }
53
54  sub calc_cache_utilization ($)
55  {
56      my $i = 0;
57      my $used = 0;
58      my $cache = shift;
59
60      for ($i = 0; $i < $cache_size; $i++) {
61          $used++ if (@$cache[$i] != -1);
62      }
63
64      return ($used / $cache_size) * 100.0;
65  }
66
67  # Return position in replication array if block is replicated, otherwise -1.
68  sub is_replicated ($$)
69  {
70      my $i = 0;
71      my $replicated = shift;
72      my $block = shift;
73
74      for ($i = 0; $i < scalar @$replicated; $i++) {
75          if (@$replicated[$i] == $block) {
76              return $i;
77          }
78      }
79
80      return -1;
81  }
82
83  ###### CODE ###############################################
84
85  my $disk_reads = 0;
86  my $disk_writes = 0;
87  my $i = 0;
88
89  my @cache = ();            # values are block numbers
```

```
90  my @lru_chain = ();        # values are buffer numbers; ie.
91                 # indices into @cache.
92  my @replicated = ();        # list of replicated blocks
93
94  # initialize the cache with -1
95  clear_cache (\@cache);
96  init_lru_chain (\@lru_chain);
97
98  # get in the replicated blocks
99  while (my $block = <>) {
100     chomp $block;
101     last if $block eq "";
102
103     push @replicated, $block;
104 }
105
106 # iterate over the block list
107 while (my $block = <>) {
108     chomp $block;
109     $i++;
110
111     # Is block in cache?
112     my $cache_position = in_cache (\@cache, $block);
113
114     # find free block in cache
115     if ($cache_position == -1) {
116         my $rep_pos;
117
118         # Take the front of the LRU chain
119         $cache_position = shift @lru_chain;
120
121         if ($cache[$cache_position] != -1) {
122             # We have to write this block back to disk.
123             # FIXME: only if the block is dirty!
124             $disk_writes++;
125         }
126
127         # read block from disk, adds disk read
128         $disk_reads++;
129         $cache[$cache_position] = $block;
130
131         # Immediately put at the rear of the LRU chain,
132         # we expect put_block() to have been called immediately.
133         push @lru_chain, $cache_position;
134
135         if ($replication_enabled
136             && ($rep_pos = is_replicated (\@replicated, $block)) != -1) {
```

```
137             if ($replace_all_enabled) {
138                 # In the "replace all" mode we do read in all replicated
139                 # blocks back into the cache, writing back as many
140                 # blocks as required.
141 
142                 my $i;
143 
144                 for ($i = 0; $i < scalar @replicated; $i++) {
145                     if (in_cache (\@cache, $replicated[$i]) == -1) {
146                         last if (!$writeback_enabled && $cache[$lru_chain[0]] != -1)
                                ;
147 
148                         $cache_position = shift @lru_chain;
149                         $cache[$cache_position] = $replicated[$i];
150                         push @lru_chain, $cache_position;
151                     } else {
152                         # Block already in the cache, update
153                         # LRU chain and bail out.
154                         rm_lru (\@lru_chain, $cache_position);
155                         push @lru_chain, $cache_position;
156                         last;
157                     }
158                 }
159             } else {
160                 # Read replicated blocks before and after
161                 # $block until we hit a block that is in the cache
162                 # already.
163                 my ($start, $end);
164 
165                 # forward
166                 $end = $rep_pos + 1;
167                 for ( ; $end < scalar @replicated; $end++) {
168                     $cache_position = in_cache (\@cache, $replicated[$end]);
169                     if ($cache_position == -1) {
170                         last if (!$writeback_enabled && $cache[$lru_chain[0]] != -1)
                                ;
171 
172                         $cache_position = shift @lru_chain;
173                         $cache[$cache_position] = $replicated[$end];
174                         push @lru_chain, $cache_position;
175                     } else {
176                         # Block already in the cache, update
177                         # LRU chain and bail out.
178                         rm_lru (\@lru_chain, $cache_position);
179                         push @lru_chain, $cache_position;
180                         last;
181                     }
```

```
182                 }
183
184                 # backward
185                 $start = $rep_pos - 1;
186                 for ( ; $start >= 0; $start--) {
187                     $cache_position = in_cache (\@cache, $replicated[$start]);
188                     if ($cache_position == -1) {
189                         last if (!$writeback_enabled && $cache[$lru_chain[0]] != -1)
                                ;
190
191                         $cache_position = shift @lru_chain;
192                         $cache[$cache_position] = $start;
193                         push @lru_chain, $cache_position;
194                     } else {
195                         # Block already in the cache, update
196                         # LRU chain and bail out.
197                         rm_lru (\@lru_chain, $cache_position);
198                         push @lru_chain, $cache_position;
199                         last;
200                     }
201                 }
202             }
203         } elsif ($readahead_enabled) {
204             # Read ahead; from $block we can read up to 32 blocks
205             # for "free".  We stop once one of the blocks is
206             # already in the cache.  This is not the full story
207             # of read ahead on Minix as it is impossible to
208             # replicate outside of the kernel.
209             #
210             # XXX: we also don't take bufs_in_use into account.
211             for (my $j = 0; $j < 32; $j++) {
212                 $block++;
213
214                 $cache_position = in_cache (\@cache, $block);
215                 if ($cache_position != -1) {
216                     # Block already in the cache, update
217                     # LRU chain and bail out.
218                     rm_lru (\@lru_chain, $cache_position);
219                     push @lru_chain, $cache_position;
220                     last;
221                 }
222
223                 # Get block into the cache as above.
224                 $cache_position = shift @lru_chain;
225                 $cache[$cache_position] = $block;
226                 push @lru_chain, $cache_position;
227             }
```

```
228         }
229     } else {
230         # The block is in the cache at the given position.
231 
232         # Sanity check this.
233         if ($cache[$cache_position] != $block) {
234             die "Cache inconsistency detected."
235         }
236 
237         # Make sure this buffer is at the end of the lru chain,
238         # so it will not be evicted.
239         rm_lru (\@lru_chain, $cache_position);
240         push @lru_chain, $cache_position;
241     }
242 }
243 
244 
245 if ($verbose) {
246     print "The sequence contained $i blocks and required $disk_reads "
247           . "reads,$disk_writes writes.\n";
248     print "Cache utilization: " . calc_cache_utilization (\@cache) . "%\n";
249 } else {
250     print "$disk_reads;$disk_writes\n";
251 }
252 
253 exit 0;
```

E Source code listing of the kernel adaptions

Below are all changes we have made to the Minix kernel in unified diff format. The diff includes all changes made to the file system server and all source code for the optimization server we have written.

```
diff -ruN -x '*[oa~]' src-3.1.2a/include/minix/com.h src/include/minix/com.h
--- src-3.1.2a/include/minix/com.h      2006-03-10 16:10:04.000000000 +0000
+++ src/include/minix/com.h     2008-04-24 12:28:58.000000000 +0000
@@ -505,4 +505,39 @@
 #  define GETKM_PTR          m1_p1


+#ifdef ENABLE_OS
+
+/*===========================================================================*
+ *                Messages used by Optimization server                       *
+ *===========================================================================*/
+
+#define OS_RQ_BASE             0x1300
+
+#define OS_UP            (OS_RQ_BASE + 0)
+#define OS_DOWN                  (OS_RQ_BASE + 1)
+#define OS_BLOCK         (OS_RQ_BASE + 2)    /* notification of block access */
+#define OS_GET_CANDIDATES (OS_RQ_BASE + 3)
+#define OS_CANDIDATES     (OS_RQ_BASE + 4)
+#define OS_GOT_CANDIDATES (OS_RQ_BASE + 5)
+
+enum {
+       OS_OPERATION_WRITE    = 0x000,
+       OS_OPERATION_READ     = 0x001,
+       OS_OPERATION_NO_CACHE = 0x010
+};
+
+/* Fields for OS_BLOCK message */
+#  define OS_OPERATION         m5_c1           /* see enum */
+#  define OS_DEV_NR            m5_c2           /* device */
+#  define OS_BLOCK_NR          m5_l1           /* block number */
+#  define OS_PID               m5_i1           /* pid */
+#  define OS_UID               m5_i2           /* uid */
+#  define OS_INODE             m5_l2           /* inode the operation is on */
+
+/* Fields for OS_CANDIDATES message */
+#  define OS_CANDIDATES_POINTER        m1_p1
+#  define OS_CANDIDATES_COUNT  m1_i1
+
+#endif /* ENABLE_OS */
```

```
+
 #endif /* _MINIX_COM_H */
diff -ruN -x '*[oa~]' src-3.1.2a/servers/fs/Makefile src/servers/fs/Makefile
--- src-3.1.2a/servers/fs/Makefile      2006-03-10 14:11:19.000000000 +0000
+++ src/servers/fs/Makefile     2008-05-31 17:24:36.000000000 +0000
@@ -9,20 +9,21 @@

 # programs, flags, etc.
 CC =   exec cc
-CFLAGS = -I$i $(EXTRA_OPTS)
+CFLAGS = -I$i $(EXTRA_OPTS) -DENABLE_OS -DDISABLE_READAHEAD -DENABLE_FDCACHE_LIMIT
 LDFLAGS = -i
 LIBS = -lsys -lsysutil -ltimers

 OBJ =  main.o open.o read.o write.o pipe.o dmap.o \
        device.o path.o mount.o link.o super.o inode.o \
        cache.o cache2.o filedes.o stadir.o protect.o time.o \
-       lock.o misc.o utility.o select.o timers.o table.o
+       lock.o misc.o utility.o select.o timers.o table.o \
+       replicate.o

 # build local binary
 install all build:     $(SERVER)
 $(SERVER):     $(OBJ)
        $(CC) -o $@ $(LDFLAGS) $(OBJ) $(LIBS)
-       install -S 512w $@
+       install -S 1024w $@

 # clean up local files
 clean:
diff -ruN -x '*[oa~]' src-3.1.2a/servers/fs/cache.c src/servers/fs/cache.c
--- src-3.1.2a/servers/fs/cache.c       2006-03-15 15:34:11.000000000 +0000
+++ src/servers/fs/cache.c      2008-05-31 15:47:04.000000000 +0000
@@ -20,9 +20,25 @@
 #include "file.h"
 #include "fproc.h"
 #include "super.h"
+#ifdef ENABLE_OS
+#  include <stdlib.h>
+#  include "inode.h"
+#  include "param.h"
+
+/* The maximum cache size is defined in const.h */
+static int fdcache_size = 0;
+
+/* rreplicated() needs this to not trash the floppy cache. */
+PUBLIC int get_fdcache_size (void)
```

```
89 +{
90 +  return fdcache_size;
91 +}
92 +#endif /* ENABLE_OS */
93
94  FORWARD _PROTOTYPE( void rm_lru, (struct buf *bp) );
95 +#ifndef ENABLE_OS
96  FORWARD _PROTOTYPE( int rw_block, (struct buf *, int) );
97 +#endif /* !ENABLE_OS */
98
99  /*===========================================================================*
100  *                                get_block                                  *
101 @@ -72,9 +88,44 @@
102         }
103    }
104
105 -  /* Desired block is not on available chain.  Take oldest block ('front'). */
106 -  if ((bp = front) == NIL_BUF) panic(__FILE__,"all_buffers_in_use", NR_BUFS);
107 -  rm_lru(bp);
108 +#ifdef ENABLE_FDCACHE_LIMIT
109 +  /* We want to limit the size of the floppy disk cache.  Basically,
110 +   * the rule is: if this is a read from the floppy drive, the limit has
111 +   * been reached and this is not a metadata block, then we have to evict
112 +   * a block from the cache and use that buffer.  Otherwise, just take
113 +   * the oldest block as usual.
114 +   */
115 +  if (is_floppy_drive (dev)) {
116 +    /* FIXME: also need "block type" check here (metadata or not?) */
117 +    if (fdcache_size >= MAX_FDCACHE_SIZE) {
118 +      /* We walk over the LRU chain to find the oldest non-empty
119 +       * floppy disk buffer.  The block will be flushed to disk below in
120 +       * case it is dirty.
121 +       */
122 +      bp = front;
123 +      /* bp should not be NULL and be a floppy block.
124 +       * we continue as long bp is not NULL and it is not a floppy bloc
125 +       */
126 +      while (bp != NIL_BUF && !is_floppy_drive (bp->b_dev))
127 +        bp = bp->b_next;
128 +    } else {
129 +      /* Take oldest block ('front'). */
130 +      bp = front;
131 +    }
132 +
133 +    if (bp == NIL_BUF)
134 +      panic(__FILE__,"all_buffers_in_use", NR_BUFS);
135 +
```

```
136 +
137 +    rm_lru(bp);
138 +  } else {
139 +#endif /* ENABLE_FDCACHE_LIMIT */
140 +    /* Desired block is not on available chain.  Take oldest block ('front'). */
141 +    if ((bp = front) == NIL_BUF) panic(__FILE__,"all␣buffers␣in␣use", NR_BUFS);
142 +    rm_lru(bp);
143 +#ifdef ENABLE_FDCACHE_LIMIT
144 +  }
145 +#endif /* ENABLE_FDCACHE_LIMIT */
146
147    /* Remove the block that was just taken from its hash chain. */
148    b = (int) bp->b_blocknr & HASH_MASK;
149 @@ -103,6 +154,10 @@
150    }
151
152    /* Fill in block's parameters and add it to the hash chain where it goes. */
153 +#ifdef ENABLE_FDCACHE_LIMIT
154 +  if (bp->b_dev != dev && is_floppy_drive (dev))
155 +    fdcache_size++;
156 +#endif /* ENABLE_FDCACHE_LIMIT */
157    bp->b_dev = dev;              /* fill in device number */
158    bp->b_blocknr = block;        /* fill in block number */
159    bp->b_count++;                /* record that block is being used */
160 @@ -116,9 +171,13 @@
161          if (get_block2(bp, only_search)) /* in 2nd level cache */;
162          else
163  #endif
164 -        if (only_search == PREFETCH) bp->b_dev = NO_DEV;
165 -        else
166 -        if (only_search == NORMAL) {
167 +        if (only_search == PREFETCH) {
168 +#ifdef ENABLE_FDCACHE_LIMIT
169 +          if (is_floppy_drive (bp->b_dev))
170 +            fdcache_size--;
171 +#endif /* ENABLE_FDCACHE_LIMIT */
172 +          bp->b_dev = NO_DEV;
173 +        } else if (only_search == NORMAL) {
174                rw_block(bp, READING);
175          }
176    }
177 @@ -246,10 +305,108 @@
178    if (bit < sp->s_zsearch) sp->s_zsearch = bit;
179  }
180
181 +#ifdef ENABLE_OS
182 +
```

```
183 +struct os_message;
184 +struct os_message {
185 +       message m;
186 +       struct os_message *next;
187 +};
188 +
189 +static int os_queue_length = 0;
190 +static struct os_message *head = NULL;
191 +
192 +PUBLIC void notify_os_send_top (void)
193 +{
194 +  if (os_pid == -1 || os_queue_length == 0)
195 +    return;
196 +
197 +  if (send (os_pid, &head->m) == 0) {
198 +    struct os_message *msg;
199 +
200 +    /* It went okay, pop the message from the list */
201 +    msg = head;
202 +    head = msg->next;
203 +
204 +    free (msg);
205 +    os_queue_length--;
206 +  }
207 +  /* otherwise, failure; leave it on the list */
208 +}
209 +
210 +PRIVATE void notify_os_queue_message (message *m)
211 +{
212 +  struct os_message *msg;
213 +
214 +  /* Allocate memory for the message */
215 +  msg = malloc (sizeof (struct os_message));
216 +  if (!msg)
217 +    panic (__FILE__, "notify_os_queue_message: Out of memory", -1);
218 +
219 +  msg->m = *m;
220 +  msg->next = NULL;
221 +
222 +  /* Append it to the end of the list */
223 +  if (!head) {
224 +    head = msg;
225 +  } else {
226 +    /* Look for the end of the list */
227 +    struct os_message *l;
228 +
229 +    for (l = head; l; l = l->next)
```

```
230 +      if (l->next == NULL)
231 +       break;
232 +
233 +    l->next = msg;
234 +  }
235 +
236 +  os_queue_length++;
237 +}
238 +
239 +PUBLIC void notify_os (int dev, int block_nr, int rw, int extra)
240 +{
241 +  int ret = 1;
242 +  int operation = 0x0;
243 +  message m;
244 +
245 +  /* If this event was not on major device 2 (floppy), return. */
246 +  if (!is_floppy_drive (dev) || os_pid == -1)
247 +    return;
248 +
249 +  /* Fill the message */
250 +  operation = rw | extra;
251 +
252 +  m.m_type = OS_BLOCK;
253 +  m.OS_OPERATION = operation;
254 +  m.OS_DEV_NR = (dev >> MAJOR) & BYTE;
255 +  m.OS_BLOCK_NR = block_nr;
256 +  m.OS_PID = fproc[who_p].fp_pid;
257 +  m.OS_UID = fproc[who_p].fp_realuid;
258 +
259 +  if (m_in.fd >= 0 && fproc[who_p].fp_filp[m_in.fd])
260 +    m.OS_INODE = fproc[who_p].fp_filp[m_in.fd]->filp_ino->i_num - 1;
261 +  else
262 +    m.OS_INODE = -1;
263 +
264 +  /* If the queue is empty try to send the message right-away */
265 +  if (os_queue_length == 0)
266 +    ret = send (os_pid, &m);
267 +
268 +  if (ret != 0) {
269 +    /* Queue the message, send later */
270 +    notify_os_queue_message (&m);
271 +  }
272 +}
273 +#endif
274 +
275  /*===========================================================================*
276   *                                rw_block                                   *
```

```
277   *===========================================================================*/
278 +#ifdef ENABLE_OS
279 +PUBLIC int rw_block(bp, rw_flag)
280 +#else
281  PRIVATE int rw_block(bp, rw_flag)
282 +#endif
283  register struct buf *bp;       /* buffer pointer */
284  int rw_flag;                   /* READING or WRITING */
285  {
286 @@ -270,11 +427,16 @@
287         pos = (off_t) bp->b_blocknr * block_size;
288         op = (rw_flag == READING ? DEV_READ : DEV_WRITE);
289         r = dev_io(op, dev, FS_PROC_NR, bp->b_data, pos, block_size, 0);
290 +
291         if (r != block_size) {
292             if (r >= 0) r = END_OF_FILE;
293             if (r != END_OF_FILE)
294               printf("Unrecoverable␣disk␣error␣on␣device␣%d/%d,␣block␣%ld\n",
295                         (dev>>MAJOR)&BYTE, (dev>>MINOR)&BYTE, bp->b_blocknr);
296 +#ifdef ENABLE_FDCACHE_LIMIT
297 +            if (is_floppy_drive (bp->b_dev))
298 +              fdcache_size--;
299 +#endif /* ENABLE_FDCACHE_LIMIT */
300                 bp->b_dev = NO_DEV;     /* invalidate block */
301
302                 /* Report read errors to interested parties. */
303 @@ -298,7 +460,13 @@
304    register struct buf *bp;
305
306    for (bp = &buf[0]; bp < &buf[NR_BUFS]; bp++)
307 -       if (bp->b_dev == device) bp->b_dev = NO_DEV;
308 +       if (bp->b_dev == device) {
309 +#ifdef ENABLE_FDCACHE_LIMIT
310 +          if (is_floppy_drive (bp->b_dev))
311 +            fdcache_size--;
312 +#endif /* ENABLE_FDCACHE_LIMIT */
313 +          bp->b_dev = NO_DEV;
314 +        }
315
316  #if ENABLE_CACHE2
317    invalidate2(device);
318 @@ -387,11 +555,20 @@
319                                  "fs:␣I/O␣error␣on␣device␣%d/%d,␣block␣%lu\n",
320                                         (dev>>MAJOR)&BYTE, (dev>>MINOR)&BYTE,
321                                         bp->b_blocknr);
322 +#ifdef ENABLE_FDCACHE_LIMIT
323 +                                if (is_floppy_drive (bp->b_dev))
```

```
+                                   fdcache_size--;
+#endif /* ENABLE_FDCACHE_LIMIT */
                                    bp->b_dev = NO_DEV;     /* invalidate block */
                            }
                            break;
                    }
+
                    if (rw_flag == READING) {
+#ifdef ENABLE_FDCACHE_LIMIT
+                           if (bp->b_dev != dev && is_floppy_drive (dev))
+                               fdcache_size++;
+#endif /* ENABLE_FDCACHE_LIMIT */
                            bp->b_dev = dev;        /* validate block */
                            put_block(bp, PARTIAL_DATA_BLOCK);
                    } else {
diff -ruN -x '*[oa~]' src-3.1.2a/servers/fs/const.h src/servers/fs/const.h
--- src-3.1.2a/servers/fs/const.h       2006-03-28 11:34:59.000000000 +0000
+++ src/servers/fs/const.h      2008-05-23 14:49:41.000000000 +0000
@@ -103,3 +103,7 @@
 #define V2_INODE_SIZE             usizeof (d2_inode)  /* bytes in V2 dsk ino */
 #define V2_INDIRECTS(b)   ((b)/V2_ZONE_NUM_SIZE)  /* # zones/indir block */
 #define V2_INODES_PER_BLOCK(b) ((b)/V2_INODE_SIZE)/* # V2 dsk inodes/blk */
+
+#ifdef ENABLE_OS
+#  define MAX_FDCACHE_SIZE 175
+#endif
diff -ruN -x '*[oa~]' src-3.1.2a/servers/fs/glo.h src/servers/fs/glo.h
--- src-3.1.2a/servers/fs/glo.h 2006-03-03 10:20:57.000000000 +0000
+++ src/servers/fs/glo.h        2008-04-24 12:28:58.000000000 +0000
@@ -26,6 +26,11 @@
 EXTERN int err_code;           /* temporary storage for error number */
 EXTERN int rdwt_err;           /* status of last disk i/o request */

+#ifdef ENABLE_OS
+/* Optimization server */
+EXTERN int os_pid;
+#endif
+
 /* Data initialized elsewhere. */
 extern _PROTOTYPE (int (*call_vec[]), (void) ); /* sys call table */
 extern char dot1[2];   /* dot1 (&dot1[0]) and dot2 (&dot2[0]) have a special */
diff -ruN -x '*[oa~]' src-3.1.2a/servers/fs/main.c src/servers/fs/main.c
--- src-3.1.2a/servers/fs/main.c        2006-03-15 15:34:11.000000000 +0000
+++ src/servers/fs/main.c       2008-04-24 12:28:58.000000000 +0000
@@ -48,7 +48,6 @@

   fs_init();
```

```
371
372 -
373    /* This is the main loop that gets work, processes it, and sends replies. */
374    while (TRUE) {
375         get_work();             /* sets who and call_nr */
376 @@ -62,6 +61,17 @@
377          } else if (call_nr == SYN_ALARM) {
378                 /* Alarm timer expired. Used only for select(). Check it. */
379                 fs_expire_timers(m_in.NOTIFY_TIMESTAMP);
380 +#ifdef ENABLE_OS
381 +       } else if (call_nr == OS_UP) {
382 +               printf ("FS:␣Optimization␣server␣up␣at␣%d␣(e=%d)\n",
383 +                       fp->fp_pid, who_e);
384 +               os_pid = who_e;
385 +               reply (who_e, 0);
386 +       } else if (call_nr == OS_DOWN) {
387 +               printf ("FS:␣Optimization␣server␣down\n");
388 +               os_pid = -1;
389 +               reply (who_e, 0);
390 +#endif
391          } else if ((call_nr & NOTIFY_MESSAGE)) {
392                 /* Device notifies us of an event. */
393                 dev_status(&m_in);
394 @@ -81,9 +91,16 @@
395
396                 /* Copy the results back to the user and send reply. */
397                 if (error != SUSPEND) { reply(who_e, error); }
398 +
399 +#ifndef DISABLE_READAHEAD
400                 if (rdahed_inode != NIL_INODE) {
401                         read_ahead(); /* do block read ahead */
402                 }
403 +#endif
404 +#ifdef ENABLE_OS
405 +               if (os_pid != -1)
406 +                       notify_os_send_top ();
407 +#endif
408         }
409    }
410    return(OK);                          /* shouldn't come here */
411 @@ -194,6 +211,10 @@
412    message mess;
413    int s;
414
415 +#ifdef ENABLE_OS
416 +  os_pid = -1;
417 +#endif
```

```
418 +
419    /* Initialize the process table with help of the process manager messages.
420     * Expect one message for each system process with its slot number and pid.
421     * When no more processes follow, the magic process number NONE is sent.
422 diff -ruN -x '*[oa~]' src-3.1.2a/servers/fs/mount.c src/servers/fs/mount.c
423 --- src-3.1.2a/servers/fs/mount.c       2006-03-03 10:20:57.000000000 +0000
424 +++ src/servers/fs/mount.c      2008-04-24 12:28:58.000000000 +0000
425 @@ -255,6 +255,11 @@
426    sp->s_isup = root_ip;
427    sp->s_rd_only = m_in.rd_only;
428    allow_newroot= 0;            /* The root is now fixed */
429 +
430 +#ifdef ENABLE_OS
431 +  sp->s_has_replicated = has_replicated_blocks (sp);
432 +#endif /* ENABLE_OS */
433 +
434    return(OK);
435  }
436
437 @@ -304,6 +309,13 @@
438          }
439    }
440
441 +#ifdef ENABLE_OS
442 +  /* Before the cache is cleared and the device is closed, we select
443 +   * candidates and replicate these blocks.
444 +   */
445 +  replicate_blocks (sp);
446 +#endif
447 +
448    /* Sync the disk, and invalidate cache. */
449    (void) do_sync();            /* force any cached blocks out of memory */
450    invalidate(dev);             /* invalidate cache entries for this dev */
451 diff -ruN -x '*[oa~]' src-3.1.2a/servers/fs/proto.h src/servers/fs/proto.h
452 --- src-3.1.2a/servers/fs/proto.h       2006-03-15 13:37:20.000000000 +0000
453 +++ src/servers/fs/proto.h      2008-04-25 11:49:27.000000000 +0000
454 @@ -18,6 +18,13 @@
455  _PROTOTYPE( void rw_scattered, (Dev_t dev,
456                          struct buf **bufq, int bufqsize, int rw_flag)   );
457
458 +#ifdef ENABLE_OS
459 +_PROTOTYPE( int get_fdcache_size, (void)       );
460 +_PROTOTYPE( void notify_os_send_top, (void)    );
461 +_PROTOTYPE( void notify_os, (int dev, int block_nr, int rw, int extra) );
462 +_PROTOTYPE( int rw_block, (struct buf *bp, int rw_flag)                        );
463 +#endif
464 +
```

```
465  #if ENABLE_CACHE2
466  /* cache2.c */
467  _PROTOTYPE( void init_cache2, (unsigned long size)                    );
468 @@ -186,6 +193,9 @@
469  _PROTOTYPE( int no_sys, (void)                                        );
470  _PROTOTYPE( int isokendpt_f, (char *f, int l, int e, int *p, int ft));
471  _PROTOTYPE( void panic, (char *who, char *mess, int num)              );
472 +#ifdef ENABLE_OS
473 +_PROTOTYPE( int is_floppy_drive, (dev_t dev)                          );
474 +#endif /* ENABLE_OS */
475
476  #define okendpt(e, p) isokendpt_f(__FILE__, __LINE__, (e), (p), 1)
477  #define isokendpt(e, p) isokendpt_f(__FILE__, __LINE__, (e), (p), 0)
478 @@ -214,3 +224,14 @@
479
480  /* cdprobe.c */
481  _PROTOTYPE( int cdprobe, (void)                                                );
482 +
483 +#ifdef ENABLE_OS
484 +/* replicate.c */
485 +_PROTOTYPE( void replicate_blocks, (struct super_block *sp)                    );
486 +_PROTOTYPE( int get_replicated_count, (struct super_block *sp)                 );
487 +_PROTOTYPE( int has_replicated_blocks, (struct super_block *sp)
             );
488 +_PROTOTYPE( int is_block_replicated, (struct super_block *sp, block_t block)    );
489 +_PROTOTYPE( block_t get_original_block, (struct super_block *sp, int replicated_pos
    )            );
490 +_PROTOTYPE( block_t get_replicated_block, (struct super_block *sp, int
    replicated_pos)          );
491 +_PROTOTYPE( block_t get_original_from_replicated_block, (struct super_block *sp,
    block_t replicated));
492 +#endif /* ENABLE_OS */
493 diff -ruN -x '*[oa~]' src-3.1.2a/servers/fs/read.c src/servers/fs/read.c
494 --- src-3.1.2a/servers/fs/read.c        2006-03-03 10:20:57.000000000 +0000
495 +++ src/servers/fs/read.c       2008-05-31 15:43:37.000000000 +0000
496 @@ -25,6 +25,11 @@
497  FORWARD _PROTOTYPE( int rw_chunk, (struct inode *rip, off_t position,
498         unsigned off, int chunk, unsigned left, int rw_flag,
499         char *buff, int seg, int usr, int block_size, int *completed));
500 +#ifdef ENABLE_OS
501 +FORWARD _PROTOTYPE( struct buf *rreplicated, (struct inode *rip,
502 +                      block_t baseblock, off_t position,
503 +                      unsigned bytes_ahead)           );
504 +#endif
505
506  /*===========================================================================*
507   *                              do_read                                      *
```

```
@@ -287,8 +292,16 @@
                if ((bp= new_block(rip, position)) == NIL_BUF)return(err_code);
        }
   } else if (rw_flag == READING) {
+#ifdef ENABLE_OS
+        if (rip->i_sp && rip->i_sp->s_has_replicated)
+            bp = rreplicated(rip, b, position, left);
+        else {
+            bp = rahead(rip, b, position, left);
+       }
+#else /* !ENABLE_OS */
        /* Read and read ahead if convenient. */
        bp = rahead(rip, b, position, left);
+#endif /* !ENABLE_OS */
   } else {
        /* Normally an existing block to be partially overwritten is first read
         * in.  However, a full block need not be read in.  If it is already in
@@ -308,6 +321,12 @@
        zero_block(bp);
   }

+#ifdef ENABLE_OS
+  notify_os (dev, bp->b_blocknr,
+            rw_flag == READING ? OS_OPERATION_READ : OS_OPERATION_WRITE,
+            0);
+#endif
+
   if (rw_flag == READING) {
        /* Copy a chunk from the block buffer to user space. */
        r = sys_vircopy(FS_PROC_NR, D, (phys_bytes) (bp->b_data+off),
@@ -483,6 +502,7 @@
   bp = get_block(dev, block, PREFETCH);
   if (bp->b_dev != NO_DEV) return(bp);

+#ifndef DISABLE_READAHEAD
   /* The best guess for the number of blocks to prefetch:  A lot.
    * It is impossible to tell what the device looks like, so we don't even
    * try to guess the geometry, but leave it to the driver.
@@ -545,6 +565,11 @@

        /* Don't trash the cache, leave 4 free. */
        if (bufs_in_use >= NR_BUFS - 4) break;
+#ifdef ENABLE_FDCACHE_LIMIT
+       if (is_floppy_drive (dev)
+           && get_fdcache_size () >= MAX_FDCACHE_SIZE)
+               break;
+#endif
```

```
555
556         block++;
557
558 @@ -555,6 +580,153 @@
559                 break;
560         }
561    }
562 +
563    rw_scattered(dev, read_q, read_q_size, READING);
564 +#else /* DISABLE_READAHEAD */
565 +  put_block(bp, FULL_DATA_BLOCK);
566 +#endif
567    return(get_block(dev, baseblock, NORMAL));
568  }
569 +
570 +#ifdef ENABLE_OS
571 +/*===========================================================================*
572 + *                      rreplicated                                          *
573 + *===========================================================================*/
574 +PRIVATE struct buf *rreplicated(rip, baseblock, position, bytes_ahead)
575 +register struct inode *rip;    /* pointer to inode for file to be read */
576 +block_t baseblock;             /* block at current position */
577 +off_t position;                        /* position within file */
578 +unsigned bytes_ahead;          /* bytes beyond position for immediate use */
579 +{
580 +  int pos;
581 +  int block_size;
582 +  int replicated_pos;
583 +  int replicated_count;
584 +  int block_spec, scale;
585 +  block_t block;
586 +  dev_t dev;
587 +  struct buf *bp;
588 +  int read_q_size;
589 +  static struct buf *read_q[NR_BUFS];
590 +
591 +  /* Variant on the rahead() function that reads in replicated
592 +   * blocks instead.
593 +   */
594 +  block_spec = (rip->i_mode & I_TYPE) == I_BLOCK_SPECIAL;
595 +  if (block_spec) {
596 +       dev = (dev_t) rip->i_zone[0];
597 +  } else {
598 +       dev = rip->i_dev;
599 +  }
600 +
601 +  /* Some sanity checks to make sure we are not going to totally
```

```
602 +   * fuck up.
603 +   */
604 +  if (!is_floppy_drive (dev))
605 +    panic (__FILE__, "FS␣panic:␣replicated␣read␣attempted␣on␣non-FDD\n", -1);
606 +
607 +  if (!rip->i_sp->s_has_replicated)
608 +    panic (__FILE__,
609 +           "FS␣panic:␣replicated␣read␣attempted␣on␣non-replicated␣file␣system\n",
610 +           -1);
611 +
612 +  block_size = get_block_size(dev);
613 +
614 +  block = baseblock;
615 +  replicated_pos = is_block_replicated (rip->i_sp, block);
616 +  if (replicated_pos < 0) {
617 +    /* The block is not replicated, use the normal method instead. */
618 +    return rahead (rip, baseblock, position, bytes_ahead);
619 +  }
620 +
621 +  /* FIXME: temporary assertion */
622 +  if (block != get_original_block (rip->i_sp, replicated_pos))
623 +    panic (__FILE__, "FS␣panic:␣replicated␣array␣mismatch\n", -1);
624 +
625 +  bp = get_block (dev, get_original_block (rip->i_sp, replicated_pos), PREFETCH);
626 +  if (bp->b_dev != NO_DEV) {
627 +    /* The requested block is already in the cache. */
628 +    return(bp);
629 +  }
630 +
631 +  /* Otherwise, do the actual read from the replicated area */
632 +  bp->b_blocknr = get_replicated_block (rip->i_sp, replicated_pos);
633 +
634 +  /* The baseblock is in the replicated blocks array at subscript
635 +   * replicated_pos.  Look if there is more to prefetch here.
636 +   *
637 +   * There are several methods we can use here, see the thesis.
638 +   */
639 +
640 +  read_q_size = 0;
641 +  read_q[read_q_size++] = bp;
642 +
643 +  /* Walk backwards in the candidates array */
644 +  replicated_count = get_replicated_count (rip->i_sp);
645 +  pos = replicated_pos - 1;
646 +
647 +  for ( ; pos >= 0; pos--) {
648 +       /* Don't trash the cache, leave 4 free. */
```

```
649 +       if (bufs_in_use >= NR_BUFS - 4) break;
650 +#ifdef ENABLE_FDCACHE_LIMIT
651 +       if (get_fdcache_size () >= MAX_FDCACHE_SIZE) break;
652 +#endif
653 +
654 +       bp = get_block (dev, get_original_block (rip->i_sp, pos), PREFETCH);
655 +       if (bp->b_dev != NO_DEV) {
656 +               /* Oops, block already in the cache, get out. */
657 +               put_block(bp, FULL_DATA_BLOCK);
658 +               break;
659 +       }
660 +
661 +        /* Otherwise, fetch from the replicated area */
662 +        bp->b_blocknr = get_replicated_block (rip->i_sp, pos);
663 +       read_q[read_q_size++] = bp;
664 +  }
665 +
666 +  /* Now, walk forwards */
667 +  pos = replicated_pos + 1;
668 +
669 +  for ( ; pos < replicated_count; pos++) {
670 +       /* Don't trash the cache, leave 4 free. */
671 +       if (bufs_in_use >= NR_BUFS - 4) break;
672 +#ifdef ENABLE_FDCACHE_LIMIT
673 +       if (get_fdcache_size () >= MAX_FDCACHE_SIZE) break;
674 +#endif
675 +
676 +       bp = get_block (dev, get_original_block (rip->i_sp, pos), PREFETCH);
677 +       if (bp->b_dev != NO_DEV) {
678 +               /* Oops, block already in the cache, get out. */
679 +               put_block(bp, FULL_DATA_BLOCK);
680 +               break;
681 +       }
682 +
683 +        /* Otherwise, fetch from the replicated area */
684 +        bp->b_blocknr = get_replicated_block (rip->i_sp, pos);
685 +       read_q[read_q_size++] = bp;
686 +  }
687 +
688 +  /* Read the data info the block buffers */
689 +  rw_scattered (dev, read_q, read_q_size, READING);
690 +
691 +  /* rw_scattered() above did call put_block() for all the of blocks.
692 +   * However, the pointers we have in read_q are still valid.  Fix up
693 +   * the blocks to point at the original block numbers.
694 +   */
695 +  for (pos = 0; pos < read_q_size; pos++) {
```

```
696 +    if (read_q[pos]->b_dev != NO_DEV) {
697 +      /* The cache buffer is still valid */
698 +      read_q[pos]->b_blocknr =
699 +        get_original_from_replicated_block (rip->i_sp, read_q[pos]->b_blocknr);
700 +    }
701 +  }
702 +
703 +  /* Getting the block as NORMAL is fail-safe; if the floppy disk
704 +   * driver decided to not read all blocks we requested through
705 +   * rw_scattered(), this will still read it from the disk if it is
706 +   * not in the cache.
707 +   */
708 +  return (get_block(dev, baseblock, NORMAL));
709 +}
710 +
711 +#endif /* ENABLE_OS */
712 diff -ruN -x '*[oa~]' src-3.1.2a/servers/fs/replicate.c src/servers/fs/replicate.c
713 --- src-3.1.2a/servers/fs/replicate.c   1970-01-01 00:00:00.000000000 +0000
714 +++ src/servers/fs/replicate.c  2008-05-31 17:15:33.000000000 +0000
715 @@ -0,0 +1,396 @@
716 +#ifdef ENABLE_OS
717 +
718 +#include "fs.h"
719 +#include <fcntl.h>
720 +#include <string.h>
721 +#include <stdlib.h>
722 +#include <minix/com.h>
723 +#include <sys/stat.h>
724 +#include "buf.h"
725 +#include "file.h"
726 +#include "super.h"
727 +#include "fproc.h"
728 +#include "inode.h"
729 +#include "param.h"
730 +
731 +#define INFO_MAGIC 0x00914470
732 +
733 +struct info_block {
734 +  unsigned int count;
735 +  unsigned int magic;
736 +};
737 +
738 +
739 +PRIVATE struct buf *get_info_block (struct super_block *sp)
740 +{
741 +  int block = sp->s_zones - 1;
742 +
```

```
743 +  return get_block (sp->s_dev, block, NORMAL);
744 +}
745 +
746 +
747 +PRIVATE int check_info_block (struct super_block *sp)
748 +{
749 +  struct buf *bp;
750 +  struct info_block info;
751 +
752 +  bp = get_info_block (sp);
753 +  memcpy (&info, bp->b_data, sizeof (struct info_block));
754 +  info.count = conv4 (sp->s_native, info.count);
755 +  info.magic = conv4 (sp->s_native, info.magic);
756 +
757 +  put_block (bp, PARTIAL_DATA_BLOCK);
758 +
759 +  if (info.magic == INFO_MAGIC)
760 +    return info.count;
761 +  /* else */
762 +  return 0;
763 +}
764 +
765 +PRIVATE void write_info_block (struct super_block *sp,
766 +                               unsigned int *candidates,
767 +                               int count)
768 +{
769 +  int i;
770 +  struct buf *bp;
771 +  struct info_block info;
772 +
773 +  bp = get_info_block (sp);
774 +
775 +  info.count = conv4 (sp->s_native, count);
776 +  info.magic = conv4 (sp->s_native, INFO_MAGIC);
777 +
778 +  memcpy (bp->b_data, &info, sizeof (struct info_block));
779 +  memcpy (bp->b_data + sizeof (struct info_block),
780 +          candidates, sizeof (unsigned int) * count);
781 +
782 +  /* Now walk over the candidates in the buffer and bitswap */
783 +  candidates = (unsigned int *)(bp->b_data + sizeof (struct info_block));
784 +  for (i = 0; i < count; i++)
785 +    candidates[i] = conv4 (sp->s_native, candidates[i]);
786 +
787 +  bp->b_dirt = DIRTY;
788 +  put_block (bp, PARTIAL_DATA_BLOCK);
789 +}
```

```
790 +
791 +/* This function manipulates the zone map and is therefore mostly taken
792 + * from super.c:alloc_bit().
793 + */
794 +PRIVATE int check_space (struct super_block *sp, int count)
795 +{
796 +  int reserved = 0;
797 +  block_t start_block;
798 +  bit_t map_bits;
799 +  bit_t current;
800 +  bit_t up_to;
801 +  bitchunk_t k;
802 +  unsigned int block, word, bit;
803 +  struct buf *bp;
804 +
805 +  /* FIXME: we need support for zone maps that are larger than a single
806 +   * block once we want to apply replication on larger file systems.
807 +   */
808 +  if (sp->s_zmap_blocks != 1) {
809 +    printf ("FS:␣cannot␣handle␣s_zmap_blocks␣!=␣1\n");
810 +    return -1;
811 +  }
812 +
813 +  start_block = START_BLOCK + sp->s_imap_blocks;
814 +  map_bits = sp->s_zones - (sp->s_firstdatazone - 1);
815 +
816 +  /* Start at the end of the map */
817 +  current = map_bits - 1;
818 +
819 +  block = current / FS_BITS_PER_BLOCK(sp->s_block_size);
820 +  word = (current % FS_BITS_PER_BLOCK(sp->s_block_size)) / FS_BITCHUNK_BITS;
821 +  bit = current % FS_BITCHUNK_BITS;
822 +
823 +  bp = get_block (sp->s_dev, start_block + block, NORMAL);
824 +  k = conv2 (sp->s_native, (int) bp->b_bitmap[word]);
825 +  if (k & (1 << bit)) {
826 +    /* The last block is occupied -- check if this is an info block */
827 +    reserved = check_info_block (sp);
828 +
829 +    if (!reserved) {
830 +      /* Not an info block so another block and occupied; bail out */
831 +      put_block (bp, MAP_BLOCK);
832 +      printf ("FS:␣last␣block␣already␣in␣use␣and␣not␣an␣info␣block\n");
833 +      return -1;
834 +    } else {
835 +      printf ("FS:␣found␣info␣block,␣reusing␣reserves\n");
836 +    }
```

```
837 +
838 +    current = map_bits - 2 - reserved;
839 +    up_to = map_bits - 1 - count;
840 +  } else {
841 +    current = map_bits - 2;
842 +    up_to = map_bits - 1 - count;
843 +  }
844 +
845 +  /* FIXME: Need to handle the case where we are going to need less
846 +   * blocks than are currently reserved.  We then want to free these
847 +   * blocks.
848 +   */
849 +  if (current < up_to) {
850 +    /* Enough have been reserved already. */
851 +    put_block (bp, MAP_BLOCK);
852 +    return 0;
853 +  }
854 +
855 +  /* Walk backwards in the map and see if enough blocks are free */
856 +  do {
857 +    /* FIXME: We have a very "cautious" implementation here that can
858 +     * be optimized later on.
859 +     */
860 +    word = (current % FS_BITS_PER_BLOCK(sp->s_block_size)) / FS_BITCHUNK_BITS;
861 +    bit = current % FS_BITCHUNK_BITS;
862 +
863 +    k = conv2 (sp->s_native, (int) bp->b_bitmap[word]);
864 +    if (k & (1 << bit)) {
865 +      printf ("FS:␣block␣%d␣occupied,␣cannot␣replicate.\n",
866 +              current + sp->s_firstdatazone);
867 +      put_block (bp, MAP_BLOCK);
868 +      return -1;
869 +    }
870 +
871 +    current--;
872 +  } while (current >= up_to);
873 +
874 +  /* If yes, walk forward and mark blocks (up_to until map_bits - 1,
875 +   * inclusive, we also mark the info block bit as occupied.)
876 +   */
877 +  for (current = up_to; current <= map_bits - 1; current++) {
878 +    /* FIXME: again a very cautious implementation */
879 +    word = (current % FS_BITS_PER_BLOCK(sp->s_block_size)) / FS_BITCHUNK_BITS;
880 +    bit = current % FS_BITCHUNK_BITS;
881 +
882 +    k = conv2 (sp->s_native, (int) bp->b_bitmap[word]);
883 +    k |= 1 << bit;
```

```
884 +    bp->b_bitmap[word] = conv2 (sp->s_native, (int)k);
885 +  }
886 +
887 +  /* Zone map has been modified, mark block as dirty */
888 +  bp->b_dirt = DIRTY;
889 +  put_block (bp, MAP_BLOCK);
890 +
891 +  return 0;
892 +}
893 +
894 +PRIVATE void copy_blocks (struct super_block *sp,
895 +                          unsigned int *candidates,
896 +                          int count)
897 +{
898 +  int i;
899 +  int start;
900 +
901 +  start = sp->s_zones - 1 - count;
902 +
903 +  printf ("FS: replicating ...\n");
904 +
905 +  for (i = 0; i < count; i++, start++) {
906 +    struct buf *bp;
907 +    block_t blocknr;
908 +    char dirt;
909 +
910 +    printf ("%d ", candidates[i]);
911 +    bp = get_block (sp->s_dev, candidates[i], NORMAL);
912 +
913 +    blocknr = bp->b_blocknr;
914 +    dirt = bp->b_dirt;
915 +
916 +    /* FIXME: we could later redo this loop and use rw_scattered() */
917 +    bp->b_blocknr = start;
918 +    rw_block (bp, WRITING);
919 +
920 +    /* FIXME: handle cache if b_dev is set to NO_DEV? */
921 +
922 +    bp->b_blocknr = blocknr;
923 +    bp->b_dirt = dirt;
924 +
925 +    put_block (bp, FULL_DATA_BLOCK);
926 +  }
927 +
928 +  printf ("[done]\n");
929 +}
930 +
```

```
931 +PRIVATE void replicate_candidates (struct super_block *sp,
932 +                                   unsigned int *candidates,
933 +                                   int count)
934 +{
935 +  /* Check if there is enough space for replication */
936 +  if (check_space (sp, count) < 0)
937 +    return;
938 +
939 +  copy_blocks (sp, candidates, count);
940 +
941 +  write_info_block (sp, candidates, count);
942 +}
943 +
944 +PUBLIC void replicate_blocks (struct super_block *sp)
945 +{
946 +  int ret;
947 +  message m;
948 +  unsigned int *candidates;
949 +  unsigned int size;
950 +
951 +  if (!is_floppy_drive (sp->s_dev) || os_pid == -1)
952 +    return;
953 +
954 +  /* This function is called on umount; free the replicated blocks
955 +   * array in the super_bock.
956 +   */
957 +  if (sp->s_replicated_blocks) {
958 +    free (sp->s_replicated_blocks);
959 +    sp->s_replicated_blocks = NULL;
960 +    sp->s_replicated_count = 0;
961 +  }
962 +
963 +  if (sp->s_rd_only) {
964 +    printf ("FS:␣Will␣not␣replicate␣blocks␣on␣read-only␣file␣system\n");
965 +    return;
966 +  }
967 +
968 +  m.m_type = OS_GET_CANDIDATES;
969 +  ret = sendrec (os_pid, &m);
970 +
971 +  if (ret != 0 || m.m_type != OS_CANDIDATES) {
972 +    printf ("FS:␣failed␣to␣request␣candidates␣from␣OS\n");
973 +    return;
974 +  }
975 +
976 +  printf ("FS:␣received␣message␣from␣OS;␣%d␣candidates\n",
977 +          m.OS_CANDIDATES_COUNT);
```

```
978 +
979 +  size = sizeof (unsigned int) * m.OS_CANDIDATES_COUNT;
980 +  candidates = malloc (size);
981 +  ret = sys_datacopy (m.m_source, (vir_bytes)m.OS_CANDIDATES_POINTER,
982 +                      SELF, (vir_bytes)candidates,
983 +                      (phys_bytes)size);
984 +
985 +  if (!ret) {
986 +    replicate_candidates (sp, candidates, m.OS_CANDIDATES_COUNT);
987 +  }
988 +
989 +  m.m_type = OS_GOT_CANDIDATES;
990 +  ret = send (os_pid, &m);
991 +
992 +  if (ret != 0) {
993 +    printf ("FS:␣couldn't␣send␣got-candidates␣message\n");
994 +  }
995 +
996 +  free (candidates);
997 +}
998 +
999 +PUBLIC int get_replicated_count (struct super_block *sp)
1000 +{
1001 +  if (!sp->s_has_replicated)
1002 +    return 0;
1003 +
1004 +  return sp->s_replicated_count;
1005 +}
1006 +
1007 +PUBLIC int has_replicated_blocks (struct super_block *sp)
1008 +{
1009 +  int i;
1010 +  int reserved;
1011 +  struct buf *bp;
1012 +
1013 +  /* We do not check for os_pid here, the optimization server does not
1014 +   * have to be active in order to read replicated blocks.
1015 +   */
1016 +  if (!is_floppy_drive (sp->s_dev))
1017 +    return 0;
1018 +
1019 +  /* FIXME: we should actually check if the info block is still marked
1020 +   * as occupied in the zone bitmap.
1021 +   */
1022 +
1023 +  reserved = check_info_block (sp);
1024 +  if (!reserved)
```

```
1025 +    return 0;
1026 +
1027 +  sp->s_replicated_blocks = malloc (sizeof (unsigned int) * reserved);
1028 +  if (!sp->s_replicated_blocks) {
1029 +         printf ("FS: could not allocate memory for replicated blocks list\n");
1030 +         return 0;
1031 +  }
1032 +
1033 +  sp->s_replicated_count = reserved;
1034 +
1035 +  bp = get_info_block (sp);
1036 +
1037 +  memcpy (sp->s_replicated_blocks, bp->b_data + sizeof (struct info_block),
1038 +          sizeof (unsigned int) * reserved);
1039 +
1040 +  put_block (bp, PARTIAL_DATA_BLOCK);
1041 +
1042 +  printf ("FS: Replicated blocks found on file system: ");
1043 +  for (i = 0; i < sp->s_replicated_count; i++)
1044 +    printf ("%d ", sp->s_replicated_blocks[i]);
1045 +  printf ("\n");
1046 +
1047 +  return 1;
1048 +}
1049 +
1050 +/* Returns the position in the replicated array where this block is
1051 + * replicated.  < 0 otherwise.
1052 + */
1053 +PUBLIC int is_block_replicated (struct super_block *sp, block_t block)
1054 +{
1055 +  int i;
1056 +
1057 +  /* For now, this function is only called from rreplicated which is
1058 +   * only called if the file system has replicated blocks.
1059 +   */
1060 +  if (!sp->s_replicated_blocks)
1061 +    panic (__FILE__, "FS panic: is_block_replicated() called, but no replicated
       blocks available.\n", -1);
1062 +
1063 +  for (i = 0; i < sp->s_replicated_count; i++)
1064 +    if (sp->s_replicated_blocks[i] == block)
1065 +      return i;
1066 +
1067 +  return -1;
1068 +}
1069 +
1070 +/* Return the original block number for the given replication position */
```

```
1071 +PUBLIC block_t get_original_block (struct super_block *sp, int replicated_pos)
1072 +{
1073 +  if (replicated_pos < 0 || replicated_pos >= sp->s_replicated_count)
1074 +    panic (__FILE__, "FS␣panic:␣get_original_block():␣invalid␣replicated_pos\n",
         -1);
1075 +
1076 +  return sp->s_replicated_blocks[replicated_pos];
1077 +}
1078 +
1079 +/* Return the replicated block number for the given replication position */
1080 +PUBLIC block_t get_replicated_block (struct super_block *sp,
1081 +                                     int replicated_pos)
1082 +{
1083 +  block_t block;
1084 +
1085 +  if (replicated_pos < 0 || replicated_pos >= sp->s_replicated_count)
1086 +    panic (__FILE__, "FS␣panic:␣get_replicated_block():␣invalid␣replicated_pos\n",
         -1);
1087 +
1088 +  if (!sp->s_has_replicated)
1089 +    panic (__FILE__, "FS␣panic:␣get_replicated_block():␣super␣has␣no␣replicated\n",
         -1);
1090 +
1091 +  block = sp->s_zones - 1 - sp->s_replicated_count + replicated_pos;
1092 +
1093 +  return block;
1094 +}
1095 +
1096 +/* Translate a replicated block number in the original block number */
1097 +PUBLIC block_t get_original_from_replicated_block (struct super_block *sp,
1098 +                                                   block_t replicated)
1099 +{
1100 +  int replicated_pos;
1101 +
1102 +  if (replicated < get_replicated_block (sp, 0)
1103 +      || replicated >= sp->s_zones - 1)
1104 +    panic (__FILE__, "FS␣panic:␣get_original_from_replicated_block():␣invalid␣
         replicated\n", -1);
1105 +
1106 +  replicated_pos = replicated - get_replicated_block (sp, 0);
1107 +
1108 +  return get_original_block (sp, replicated_pos);
1109 +}
1110 +
1111 +#endif /* ENABLE_OS */
1112 diff -ruN -x '*[oa~]' src-3.1.2a/servers/fs/super.c src/servers/fs/super.c
1113 --- src-3.1.2a/servers/fs/super.c       2005-12-20 14:23:44.000000000 +0000
```

```
+++ src/servers/fs/super.c      2008-04-24 12:28:58.000000000 +0000
@@ -305,6 +305,12 @@
   sp->s_version = version;
   sp->s_native  = native;

+#ifdef ENABLE_OS
+  sp->s_has_replicated = 0;
+  sp->s_replicated_blocks = NULL;
+  sp->s_replicated_count = 0;
+#endif /* ENABLE_OS */
+
   /* Make a few basic checks to see if super block looks reasonable. */
   if (sp->s_imap_blocks < 1 || sp->s_zmap_blocks < 1
                                || sp->s_ninodes < 1 || sp->s_zones < 1
diff -ruN -x '*[oa~]' src-3.1.2a/servers/fs/super.h src/servers/fs/super.h
--- src-3.1.2a/servers/fs/super.h       2005-08-22 15:23:47.000000000 +0000
+++ src/servers/fs/super.h      2008-04-24 12:28:58.000000000 +0000
@@ -52,6 +52,11 @@
   int s_nindirs;               /* # indirect zones per indirect block */
   bit_t s_isearch;             /* inodes below this bit number are in use */
   bit_t s_zsearch;             /* all zones below this bit number are in use*/
+#ifdef ENABLE_OS
+  int s_has_replicated;                /* whether the file system has replicated
    blocks */
+  block_t *s_replicated_blocks;
+  int s_replicated_count;
+#endif /* ENABLE_OS */
 } super_block[NR_SUPERS];

 #define NIL_SUPER (struct super_block *) 0
diff -ruN -x '*[oa~]' src-3.1.2a/servers/fs/utility.c src/servers/fs/utility.c
--- src-3.1.2a/servers/fs/utility.c     2006-03-10 16:10:05.000000000 +0000
+++ src/servers/fs/utility.c    2008-04-24 12:28:58.000000000 +0000
@@ -163,3 +163,12 @@
        return failed ? EDEADSRCDST : OK;
 }

+#ifdef ENABLE_OS
+PUBLIC int is_floppy_drive (dev_t dev)
+{
+  if (((dev >> MAJOR) & BYTE) == 2)
+    return 1;
+
+  return 0;
+}
+#endif /* ENABLE_OS */
diff -ruN -x '*[oa~]' src-3.1.2a/servers/os/Makefile src/servers/os/Makefile
```

```
--- src-3.1.2a/servers/os/Makefile      1970-01-01 00:00:00.000000000 +0000
+++ src/servers/os/Makefile     2008-04-24 12:28:58.000000000 +0000
@@ -0,0 +1,41 @@
+# Makefile for Optimization Server (OS)
+SERVER = os
+
+# directories
+u = /usr
+i = $u/include
+s = $i/sys
+m = $i/minix
+b = $i/ibm
+k = $u/src/kernel
+p = $u/src/servers/pm
+f = $u/src/servers/fs
+
+# programs, flags, etc.
+CC =   exec cc
+CFLAGS = -I$i -DENABLE_OS
+LDFLAGS = -i
+LIBS = -lsys -lsysutil
+
+OBJ = main.o discover.o
+
+# build local binary
+all build:     $(SERVER)
+$(SERVER):     $(OBJ)
+       $(CC) -o $@ $(LDFLAGS) $(OBJ) $(LIBS)
+       install -S 160k $@
+
+# install with other servers
+install: $(SERVER)
+       install -o root -c $? /sbin/$(SERVER)
+#      install -o root -cs $? $@
+
+# clean up local files
+clean:
+       rm -f $(SERVER) *.o *.bak
+
+depend:
+       /usr/bin/mkdep "$(CC) -E $(CPPFLAGS)" *.c > .depend
+
+# Include generated dependencies.
+include .depend
diff -ruN -x '*[oa~]' src-3.1.2a/servers/os/discover.c src/servers/os/discover.c
--- src-3.1.2a/servers/os/discover.c    1970-01-01 00:00:00.000000000 +0000
+++ src/servers/os/discover.c   2008-05-31 17:24:18.000000000 +0000
```

```
1207 @@ -0,0 +1,371 @@
1208 +#include <stdio.h>
1209 +#include <stdlib.h>
1210 +#include <string.h>
1211 +
1212 +#include "proto.h"
1213 +
1214 +/* Parameters for the algorithm */
1215 +#define N_SIZE 5
1216 +#define WINDOW_SIZE 2*(N_SIZE)+1
1217 +#define COUNT_THRESHOLD 5
1218 +#define MAX_CANDIDATES 32
1219 +
1220 +
1221 +struct NeighborInfo;
1222 +struct BlockInfo;
1223 +
1224 +struct NeighborInfo {
1225 +       int block;
1226 +       int count;
1227 +
1228 +       struct BlockInfo *parent;
1229 +       struct NeighborInfo *next;
1230 +};
1231 +
1232 +struct BlockInfo {
1233 +       int block;
1234 +       struct NeighborInfo *neighbors;
1235 +
1236 +       struct BlockInfo *next;
1237 +};
1238 +
1239 +static struct BlockInfo *accesses = NULL;
1240 +
1241 +
1242 +static void
1243 +block_info_add_neighbor (struct BlockInfo *info,
1244 +                         int               neighbor)
1245 +{
1246 +       struct NeighborInfo *n;
1247 +
1248 +       for (n = info->neighbors; n; n = n->next) {
1249 +               if (n->block == neighbor) {
1250 +                       n->count++;
1251 +                       return;
1252 +               }
1253 +       }
```

```
1254 +
1255 +       /* Neighbor not found in list; add it */
1256 +       n = malloc (sizeof (struct NeighborInfo));
1257 +       n->block = neighbor;
1258 +       n->count = 1;
1259 +       n->parent = info;
1260 +       n->next = info->neighbors;
1261 +       info->neighbors = n;
1262 +}
1263 +
1264 +static struct BlockInfo *
1265 +get_block_by_number (int number)
1266 +{
1267 +       struct BlockInfo *info;
1268 +
1269 +       for (info = accesses; info; info = info->next) {
1270 +               if (info->block == number)
1271 +                       return info;
1272 +       }
1273 +
1274 +       /* Block not found in list, create a new one */
1275 +       info = malloc (sizeof (struct BlockInfo));
1276 +       info->block = number;
1277 +       info->neighbors = NULL;
1278 +       info->next = accesses;
1279 +       accesses = info;
1280 +
1281 +       return info;
1282 +}
1283 +
1284 +/* This function assumes that we have created the count array and
1285 + * thus all neighborinfo blocks have already been freed.
1286 + */
1287 +static void
1288 +free_block_info (void)
1289 +{
1290 +       if (!accesses)
1291 +               return;
1292 +
1293 +       while (accesses) {
1294 +               struct BlockInfo *b = accesses;
1295 +               accesses = accesses->next;
1296 +
1297 +               free (b);
1298 +       }
1299 +
1300 +       accesses = NULL;
```

```
1301 +}
1302 +
1303 +
1304 +struct CountInfo;
1305 +
1306 +struct CountInfo {
1307 +       int count;
1308 +       struct NeighborInfo *blocks;
1309 +
1310 +       struct CountInfo *next;
1311 +};
1312 +
1313 +static struct CountInfo *
1314 +count_info_new (struct NeighborInfo *neighbor)
1315 +{
1316 +       struct CountInfo *info;
1317 +
1318 +       info = malloc (sizeof (struct CountInfo));
1319 +
1320 +       info->count = neighbor->count;
1321 +       info->blocks = neighbor;
1322 +       info->next = NULL;
1323 +
1324 +       return info;
1325 +}
1326 +
1327 +static void
1328 +count_info_add_neighbor (struct CountInfo     **count,
1329 +                         struct NeighborInfo   *neighbor)
1330 +{
1331 +       struct CountInfo *i;
1332 +       struct CountInfo *info;
1333 +       struct CountInfo *prev = NULL;
1334 +
1335 +       /* Look if there is an entry for neighbor->count yet.  Use the
1336 +        * fact that count is sorted as an advantage.
1337 +        */
1338 +       for (i = *count; i; prev = i, i = i->next) {
1339 +               if (i->count == neighbor->count) {
1340 +                       neighbor->next = i->blocks;
1341 +                       i->blocks = neighbor;
1342 +
1343 +                       return;
1344 +               }
1345 +
1346 +               if (neighbor->count > i->count) {
1347 +                       /* We need to insert a new countinfo here. */
```

```
1348 +                       info = count_info_new (neighbor);
1349 +
1350 +                       neighbor->next = NULL;
1351 +
1352 +                       if (!prev) {
1353 +                               info->next = *count;
1354 +                               *count = info;
1355 +                       } else {
1356 +                               info->next = prev->next;
1357 +                               prev->next = info;
1358 +                       }
1359 +
1360 +                       return;
1361 +               }
1362 +       }
1363 +
1364 +       /* Insert new count info */
1365 +       info = count_info_new (neighbor);
1366 +
1367 +       neighbor->next = NULL;
1368 +       if (!*count)
1369 +               *count = info;
1370 +       else
1371 +               /* We hit the end of the list, prev points at last element. */
1372 +               prev->next = info;
1373 +}
1374 +
1375 +static struct CountInfo *
1376 +create_count_array (void)
1377 +{
1378 +       struct BlockInfo *b;
1379 +       struct CountInfo *count = NULL;
1380 +
1381 +       /* Create count array, do descending insert sort for new counts. */
1382 +       for (b = accesses; b; b = b->next) {
1383 +               while (b->neighbors) {
1384 +                       struct NeighborInfo *n = NULL;
1385 +
1386 +                       n = b->neighbors;
1387 +                       b->neighbors = b->neighbors->next;
1388 +
1389 +                       if (n->count > COUNT_THRESHOLD)
1390 +                               count_info_add_neighbor (&count, n);
1391 +                       else
1392 +                               free (n);
1393 +               }
1394 +       }
```

```
1395 +
1396 +       return count;
1397 +}
1398 +
1399 +static void
1400 +free_count_array (struct CountInfo *count)
1401 +{
1402 +       if (!count)
1403 +               return;
1404 +
1405 +       while (count) {
1406 +               struct CountInfo *c = count;
1407 +               count = count->next;
1408 +
1409 +               while (c->blocks) {
1410 +                       struct NeighborInfo *n = c->blocks;
1411 +                       c->blocks = c->blocks->next;
1412 +
1413 +                       free (n);
1414 +               }
1415 +
1416 +               free (c);
1417 +       }
1418 +}
1419 +
1420 +static void
1421 +add_candidate (int  candidates[MAX_CANDIDATES],
1422 +              int  value,
1423 +              int *index)
1424 +{
1425 +       int i;
1426 +
1427 +       for (i = 0; i < *index; i++)
1428 +               if (candidates[i] == value)
1429 +                       return;
1430 +
1431 +       candidates[*index] = value;
1432 +       (*index)++;
1433 +}
1434 +
1435 +static void
1436 +add_pair (int block1,
1437 +         int block2)
1438 +{
1439 +       int sub;
1440 +       int index_block;
1441 +       struct BlockInfo *block;
```

```
1442 +
1443 +       if (block1 == block2)
1444 +               return;
1445 +
1446 +       /* The rule is that pairs are always saved under the block
1447 +        * with the lowest number.
1448 +        */
1449 +       if (block1 < block2) {
1450 +               index_block = block1;
1451 +               sub = block2;
1452 +       } else {
1453 +               index_block = block2;
1454 +               sub = block1;
1455 +       }
1456 +
1457 +       block = get_block_by_number (index_block);
1458 +       block_info_add_neighbor (block, sub);
1459 +}
1460 +
1461 +static void
1462 +walk_window_for_position (unsigned int window[WINDOW_SIZE],
1463 +                          int          position,
1464 +                          int          upper_bound)
1465 +{
1466 +       int i;
1467 +
1468 +       for (i = -N_SIZE; i <= N_SIZE; i++) {
1469 +               /* Skip non-existing and irrelevant positions */
1470 +               if (position + i < 0 || i == 0 || position + i >= upper_bound)
1471 +                       continue;
1472 +
1473 +               add_pair (window[position], window[position + i]);
1474 +       }
1475 +}
1476 +
1477 +static void
1478 +analyze_window (unsigned int window[WINDOW_SIZE],
1479 +                int          index,
1480 +                int          done)
1481 +{
1482 +       if (index < N_SIZE+1)
1483 +               /* Can't analyze anything yet */
1484 +               return;
1485 +
1486 +       if (done) {
1487 +               int i;
1488 +               int end;
```

```
1489 +
1490 +               /* If we do not have all blocks for a single neighborhood
1491 +                * yet, we've bailed out above.
1492 +                *
1493 +                * Here, we will do the analysis of the "tail" of the window.
1494 +                */
1495 +
1496 +               if (index < WINDOW_SIZE)
1497 +                       end = index;
1498 +               else
1499 +                       end = WINDOW_SIZE;
1500 +
1501 +               for (i = index - N_SIZE; i < end; i++)
1502 +                       walk_window_for_position (window, i, end);
1503 +
1504 +               return;
1505 +       }
1506 +
1507 +       walk_window_for_position (window, index - (N_SIZE + 1), WINDOW_SIZE);
1508 +}
1509 +
1510 +static int window_index = 0;
1511 +static unsigned int window[WINDOW_SIZE];
1512 +
1513 +/* public functions */
1514 +void
1515 +process_block (int block)
1516 +{
1517 +       if (window_index < WINDOW_SIZE) {
1518 +               window[window_index] = block;
1519 +               window_index++;
1520 +       } else {
1521 +               memmove (window, &window[1],
1522 +                        (WINDOW_SIZE - 1) * sizeof (unsigned int));
1523 +               window[WINDOW_SIZE - 1] = block;
1524 +       }
1525 +
1526 +       analyze_window (window, window_index, 0);
1527 +}
1528 +
1529 +unsigned int *
1530 +select_candidates (int *candidates_count)
1531 +{
1532 +       int i;
1533 +       int index = 0;
1534 +       int candidates[MAX_CANDIDATES];
1535 +       unsigned int *ret;
```

```
1536 +       struct CountInfo *count;
1537 +       struct CountInfo *info;
1538 +
1539 +       /* We want to have an index by count, descending.  When then
1540 +        * walk over the index selecting blocks with high counts until
1541 +        * we have enough.
1542 +        *
1543 +        * Keeping a max count in the BlockInfo structures won't work;
1544 +        * one block can have multiple neighbors with high counts.  Instead
1545 +        * we create a new list indexed by count that points at block
1546 +        * info structures.  We use a count threshold here to save
1547 +        * memory.
1548 +        */
1549 +
1550 +       count = create_count_array ();
1551 +
1552 +       /* Select candidates */
1553 +       for (info = count; info; info = info->next) {
1554 +               struct NeighborInfo *n;
1555 +
1556 +               for (n = info->blocks; n; n = n->next) {
1557 +                       add_candidate (candidates, n->block, &index);
1558 +
1559 +                       if (index >= MAX_CANDIDATES)
1560 +                               goto done;
1561 +
1562 +                       add_candidate (candidates, n->parent->block, &index);
1563 +
1564 +                       if (index >= MAX_CANDIDATES)
1565 +                               goto done;
1566 +               }
1567 +       }
1568 +
1569 +done:
1570 +       free_count_array (count);
1571 +       free_block_info ();
1572 +
1573 +       *candidates_count = index;
1574 +       ret = malloc (sizeof (unsigned int) * index);
1575 +       memcpy (ret, candidates, sizeof (unsigned int) * index);
1576 +
1577 +       return ret;
1578 +}
1579 diff -ruN -x '*[oa~]' src-3.1.2a/servers/os/glo.h src/servers/os/glo.h
1580 --- src-3.1.2a/servers/os/glo.h 1970-01-01 00:00:00.000000000 +0000
1581 +++ src/servers/os/glo.h        2008-04-24 12:28:58.000000000 +0000
1582 @@ -0,0 +1,7 @@
```

```
1583 +/* Global variables. */
1584 +
1585 +/* The parameters of the call are kept here. */
1586 +extern int who;                         /* caller's proc number */
1587 +extern int callnr;              /* system call number */
1588 +extern int dont_reply;          /* normally 0; set to 1 to inhibit reply */
1589 +
1590 diff -ruN -x '*[oa~]' src-3.1.2a/servers/os/inc.h src/servers/os/inc.h
1591 --- src-3.1.2a/servers/os/inc.h 1970-01-01 00:00:00.000000000 +0000
1592 +++ src/servers/os/inc.h        2008-04-24 12:28:58.000000000 +0000
1593 @@ -0,0 +1,28 @@
1594 +/* Header file including all needed system headers. */
1595 +
1596 +#define _SYSTEM            1    /* get OK and negative error codes */
1597 +#define _MINIX             1   /* tell headers to include MINIX stuff */
1598 +
1599 +#include <ansi.h>
1600 +#include <sys/types.h>
1601 +#include <limits.h>
1602 +#include <errno.h>
1603 +
1604 +#include <minix/callnr.h>
1605 +#include <minix/config.h>
1606 +#include <minix/type.h>
1607 +#include <minix/const.h>
1608 +#include <minix/com.h>
1609 +#include <minix/syslib.h>
1610 +#include <minix/sysutil.h>
1611 +#include <minix/keymap.h>
1612 +#include <minix/bitmap.h>
1613 +
1614 +#include <stdlib.h>
1615 +#include <stdio.h>
1616 +#include <string.h>
1617 +#include <unistd.h>
1618 +#include <signal.h>
1619 +
1620 +#include "proto.h"
1621 +#include "glo.h"
1622 diff -ruN -x '*[oa~]' src-3.1.2a/servers/os/main.c src/servers/os/main.c
1623 --- src-3.1.2a/servers/os/main.c        1970-01-01 00:00:00.000000000 +0000
1624 +++ src/servers/os/main.c       2008-04-24 12:28:58.000000000 +0000
1625 @@ -0,0 +1,322 @@
1626 +/* Optimization server
1627 + *
1628 + * Started August 23, 2007
1629 + *
```

```
1630 + * vim:sw=2:
1631 + */
1632 +
1633 +#include "inc.h"       /* include master header file */
1634 +
1635 +#include <time.h>
1636 +
1637 +#undef ENABLE_LOGGING
1638 +
1639 +/* Allocate space for the global variables. */
1640 +int who_e;             /* caller's proc number */
1641 +int callnr;            /* system call number */
1642 +int sys_panic;         /* flag to indicate system-wide panic */
1643 +int my_pid;
1644 +
1645 +extern int errno;      /* error number set by system library */
1646 +
1647 +/* Declare some local functions. */
1648 +FORWARD _PROTOTYPE(void init_server, (int argc, char **argv)          );
1649 +FORWARD _PROTOTYPE(void exit_server, (void)                           );
1650 +FORWARD _PROTOTYPE(void sig_handler, (void)                           );
1651 +FORWARD _PROTOTYPE(void get_work, (message *m_ptr)                    );
1652 +FORWARD _PROTOTYPE(void reply, (int whom, message *m_ptr)             );
1653 +FORWARD _PROTOTYPE(void do_block, (message *m_ptr)             );
1654 +FORWARD _PROTOTYPE(void do_get_candidates, (message *m_ptr)           );
1655 +FORWARD _PROTOTYPE(void do_got_candidates, (message *m_ptr)           );
1656 +
1657 +#ifdef ENABLE_LOGGING
1658 +FORWARD _PROTOTYPE(int log_open, (void)               );
1659 +FORWARD _PROTOTYPE(void log_close, (void)             );
1660 +FORWARD _PROTOTYPE(void log_write, (int is_write, int block, int dev, int pid, int
        uid, int inode) );
1661 +#endif /* ENABLE_LOGGING */
1662 +
1663 +/*===========================================================================*
1664 + *                              main                                         *
1665 + *===========================================================================*/
1666 +PUBLIC int main(int argc, char **argv)
1667 +{
1668 +/* This is the main routine of this service. The main loop consists of
1669 + * three major activities: getting new work, processing the work, and
1670 + * sending the reply. The loop never terminates, unless a panic occurs.
1671 + */
1672 +  message m;
1673 +  int result;
1674 +  sigset_t sigset;
1675 +
```

```
1676 +  /* Initialize the server, then go to work. */
1677 +  init_server(argc, argv);
1678 +
1679 +  /* Main loop - get work and do it, forever. */
1680 +  while (TRUE) {
1681 +
1682 +    /* Wait for incoming message, sets 'callnr' and 'who'. */
1683 +    get_work(&m);
1684 +
1685 +    switch (callnr) {
1686 +      case PROC_EVENT:
1687 +       sig_handler();
1688 +       continue;
1689 +
1690 +      case OS_BLOCK:
1691 +       do_block (&m);
1692 +       continue;
1693 +
1694 +      case OS_GET_CANDIDATES:
1695 +       do_get_candidates (&m);
1696 +       continue;
1697 +
1698 +      case OS_GOT_CANDIDATES:
1699 +       do_got_candidates (&m);
1700 +       continue;
1701 +
1702 +      default:
1703 +       report("OS","warning,␣got␣illegal␣request␣from:", m.m_source);
1704 +       result = EINVAL;
1705 +    }
1706 +
1707 +    /* Finally send reply message, unless disabled. */
1708 +    if (result != EDONTREPLY) {
1709 +      m.m_type = result;               /* build reply message */
1710 +      reply(who_e, &m);                /* send it away */
1711 +    }
1712 +  }
1713 +
1714 +  return(OK);                          /* shouldn't come here */
1715 +}
1716 +
1717 +/*===========================================================================*
1718 + *                                init_server                                *
1719 + *===========================================================================*/
1720 +PRIVATE void init_server(int argc, char **argv)
1721 +{
1722 +  /* Initialize the data store server. */
```

```
1723 +  int i, s;
1724 +  int ret;
1725 +  struct sigaction sigact;
1726 +  message m;
1727 +
1728 +  my_pid = getpid ();
1729 +
1730 +  /* Install signal handler. Ask PM to transform signal into message. */
1731 +  sigact.sa_handler = SIG_MESS;
1732 +  sigact.sa_mask = ~0;                  /* block all other signals */
1733 +  sigact.sa_flags = 0;                  /* default behaviour */
1734 +  if (sigaction(SIGTERM, &sigact, NULL) < 0)
1735 +      report("DS","warning, sigaction() failed", errno);
1736 +
1737 +#ifdef ENABLE_LOGGING
1738 +  log_open ();
1739 +#endif
1740 +
1741 +  /* Let the file system server know about our PID */
1742 +  m.m_type = OS_UP;
1743 +
1744 +  ret = sendrec (FS_PROC_NR, &m);
1745 +  if (ret < 0)
1746 +    printf ("OS: Notification to file system server failed ...\n");
1747 +}
1748 +
1749 +/*===========================================================================*
1750 + *                                sig_handler                                *
1751 + *===========================================================================*/
1752 +PRIVATE void sig_handler()
1753 +{
1754 +/* Signal handler. */
1755 +  sigset_t sigset;
1756 +  int sig;
1757 +
1758 +  /* Try to obtain signal set from PM. */
1759 +  if (getsigset(&sigset) != 0) return;
1760 +
1761 +  /* Check for known signals. */
1762 +  if (sigismember(&sigset, SIGTERM)) {
1763 +      exit_server();
1764 +  }
1765 +}
1766 +
1767 +/*===========================================================================*
1768 + *                                exit_server                                *
1769 + *===========================================================================*/
```

```
1770 +PRIVATE void exit_server()
1771 +{
1772 +  int ret;
1773 +  message m;
1774 +
1775 +  /* Notify the file system server that we are going down */
1776 +  m.m_type = OS_DOWN;
1777 +
1778 +  ret = sendrec (FS_PROC_NR, &m);
1779 +  if (ret < 0)
1780 +    printf ("OS:_Notification_to_file_system_server_failed\n");
1781 +
1782 +#ifdef ENABLE_LOGGING
1783 +  log_close ();
1784 +#endif
1785 +
1786 +  /* Done. Now exit. */
1787 +  exit(0);
1788 +}
1789 +
1790 +/*===========================================================================*
1791 + *                                get_work                                   *
1792 + *===========================================================================*/
1793 +PRIVATE void get_work(m_ptr)
1794 +message *m_ptr;                                 /* message buffer */
1795 +{
1796 +  int status = 0;
1797 +  status = receive(ANY, m_ptr);   /* this blocks until message arrives */
1798 +  if (OK != status)
1799 +    panic("DS","failed_to_receive_message!", status);
1800 +  who_e = m_ptr->m_source;        /* message arrived! set sender */
1801 +  callnr = m_ptr->m_type;       /* set function call number */
1802 +}
1803 +
1804 +/*===========================================================================*
1805 + *                                reply                                      *
1806 + *===========================================================================*/
1807 +PRIVATE void reply(who_e, m_ptr)
1808 +int who_e;                              /* destination */
1809 +message *m_ptr;                                 /* message buffer */
1810 +{
1811 +  int s;
1812 +  s = send(who_e, m_ptr);    /* send the message */
1813 +  if (OK != s)
1814 +    panic("DS", "unable_to_send_reply!", s);
1815 +}
1816 +
```

```
1817 +
1818 +PRIVATE void do_block (m_ptr)
1819 +message *m_ptr;
1820 +{
1821 +#if 0
1822 +  int r;
1823 +  char buffer[256];
1824 +
1825 +  r = sys_datacopy (m_ptr->m_source, (vir_bytes)m_ptr->OS_FILENAME,
1826 +                    SELF, (vir_bytes)buffer,
1827 +                    (phys_bytes)255);
1828 +#endif
1829 +
1830 +#ifdef ENABLE_LOGGING
1831 +  log_write (m_ptr->OS_OPERATION,
1832 +             m_ptr->OS_BLOCK_NR,
1833 +             m_ptr->OS_DEV_NR,
1834 +             m_ptr->OS_PID,
1835 +             m_ptr->OS_UID,
1836 +             m_ptr->OS_INODE);
1837 +#endif
1838 +
1839 +  /* FIXME: we only log reads for now.
1840 +   * FIXME: also, we assume that the file server makes sure this is an fdd
1841 +   *        block.
1842 +   */
1843 +  if ((m_ptr->OS_OPERATION & 0x1) == OS_OPERATION_READ)
1844 +    process_block (m_ptr->OS_BLOCK_NR);
1845 +}
1846 +
1847 +
1848 +static unsigned int *candidates = NULL;
1849 +
1850 +PRIVATE void do_get_candidates (m_ptr)
1851 +message *m_ptr;
1852 +{
1853 +  message reply;
1854 +  int count;
1855 +  int i;
1856 +
1857 +  if (candidates)
1858 +    free (candidates);
1859 +
1860 +  /* FIXME: we assume that the FS will only call this for the FDD */
1861 +  candidates = select_candidates (&count);
1862 +
1863 +  reply.m_type = OS_CANDIDATES;
```

```
1864 +  reply.OS_CANDIDATES_POINTER = (char *)candidates;
1865 +  reply.OS_CANDIDATES_COUNT = count;
1866 +
1867 +  send (m_ptr->m_source, &reply);
1868 +}
1869 +
1870 +PRIVATE void do_got_candidates (m_ptr)
1871 +message *m_ptr;
1872 +{
1873 +  printf ("OS:␣freeing␣candidates\n");
1874 +  if (candidates)
1875 +    free (candidates);
1876 +  candidates = NULL;
1877 +}
1878 +
1879 +#ifdef ENABLE_LOGGING
1880 +/*
1881 + * Logging
1882 + */
1883 +PRIVATE FILE *log_file = NULL;
1884 +
1885 +PRIVATE int log_open (void)
1886 +{
1887 +  time_t current;
1888 +
1889 +  if (log_file)
1890 +    return;
1891 +
1892 +  log_file = fopen ("/root/os.log", "a");
1893 +  if (!log_file) {
1894 +    perror ("fopen");
1895 +    return -1;
1896 +  }
1897 +
1898 +  current = time (NULL);
1899 +  fprintf (log_file, "--␣Log␣file␣opened␣at␣%s", ctime (&current));
1900 +
1901 +  return 0;
1902 +}
1903 +
1904 +PRIVATE void log_close (void)
1905 +{
1906 +  time_t current;
1907 +
1908 +  if (!log_file)
1909 +    return;
1910 +
```

```
1911 +  current = time (NULL);
1912 +  fprintf (log_file, "--␣Log␣file␣closed␣at␣%s", ctime (&current));
1913 +
1914 +  fclose (log_file);
1915 +  log_file = NULL;
1916 +}
1917 +
1918 +PRIVATE void log_write (operation, block, dev, pid, uid, inode)
1919 +  short operation;
1920 +  int block;
1921 +  short dev;
1922 +  int pid;
1923 +  int uid;
1924 +  int inode;
1925 +{
1926 +  int is_read;
1927 +  int no_cache;
1928 +  time_t current;
1929 +  struct tm *lt;
1930 +
1931 +  if (!log_file)
1932 +    return;
1933 +
1934 +  current = time (NULL);
1935 +  lt = localtime (&current);
1936 +
1937 +  is_read = (operation & 0x1) == OS_OPERATION_READ;
1938 +  no_cache = (operation & 0x10) == OS_OPERATION_NO_CACHE;
1939 +
1940 +  fprintf (log_file, "[%02d:%02d:%02d]:␣%s(no_cache=%d)␣%d:%d:%d␣by␣%d␣running␣%d\n
         ",
1941 +           lt->tm_hour, lt->tm_min, lt->tm_sec,
1942 +           is_read ? "READ" : "WRITE",
1943 +           no_cache,
1944 +           dev, block, inode,
1945 +           uid, pid);
1946 +}
1947 +#endif /* ENABLE_LOGGING */
1948 diff -ruN -x '*[oa~]' src-3.1.2a/servers/os/proto.h src/servers/os/proto.h
1949 --- src-3.1.2a/servers/os/proto.h       1970-01-01 00:00:00.000000000 +0000
1950 +++ src/servers/os/proto.h      2008-04-24 12:28:58.000000000 +0000
1951 @@ -0,0 +1,8 @@
1952 +/* Function prototypes. */
1953 +
1954 +/* main.c */
1955 +_PROTOTYPE(int main, (int argc, char **argv));
1956 +
```

```
1957 +/* discover.c */
1958 +_PROTOTYPE(void process_block, (int block));
1959 +_PROTOTYPE(unsigned int *select_candidates, (int *candidates_count));
```