Weenix

Brown University Department of Computer Science
Authored by *Operating Systems* Teaching Assistant Staff
Edited by Dan Kimmel and J. Rassi

September 22, 2013

Contents

Part I

Introduction

Chapter 1

The Weenix OS

1.1 Motivation

The Weenix operating system is a project for people interested in writing parts of a Unix kernel. The operating system was originally written in 1998 by teaching assistants for Brown University's operating systems course, taught by Professor Tom Doeppner, and has been maintained by that course's staff ever since. While the operating system is mostly based on early versions of Unix, it incorporates many recent developments in operating systems.

This book is intended to be a guide for students who are working on the Weenix operating system and those interested in contributing to it. Part 1 is an introduction to Weenix, including the basics of setting up your development environment, using the build system, and running the OS in a virtual machine. Part 2 contains five chapters, each of which specify an assignment. In the operating systems course taught at Brown, students have a choice between doing either:

- All five assignments.
- A separate threads library assignment followed by two of the Weenix assignments (VFS and S5FS).

Students pursuing the former option are assigned a mentor from the course staff who will be a source of clarification or hints if the student wishes. The latter option is provided to allow students who are not interested in a career in operating systems development to learn the basics of operating systems without spending the better part of their semester hacking the Weenix kernel. Each assignment chapter begins with an overview of the assignment, then describes the expected implementation in more detail, and ends with tips for testing. Part 3 describes the inner workings of some key parts of the kernel not typically explored during the assignments. This is mainly a resource for those wishing to contribute to the Weenix project. Finally, several appendices are included

which provide more details about development tools, online resources, and the protocols of contributing new code.

1.2 How to Read This Book

We expect that there are at least two groups of people who will want to read this book. If you are someone who wants to see Weenix run *right now*, jump ahead to Quick Start: Getting Weenix Running. Everyone else should begin by reading Project Management and then can read the assignments, manual, and appendices in whatever order they please. However, if you plan to do the assignments, we recommend that you do them in order, as they build on top of one another.

1.3 History

The Weenix OS, and the course that it grew up with, have a long and illustrious history. Although the Brown operating systems course has been taught since the 1960s, Weenix was first written in the spring of 1998, running on what was known only as the Brown Simulator 2.0. The two pieces of software were written by Keith Adams, Michael Castelle, Caroline Dahllof, Jason Lango, and Dave Powell. The name Weenix ("little *nix") was invented by Keith Adams, and the OS was designed based on early versions of Unix. In 2004, a competing Brown operating system based on Windows (named HipOS) was created by Hari Khalsa, whose effort was apparently completely vanquished. During 2007-2009, Weenix moved off of the Brown Simulator and onto Xen virtual machines, an effort spearheaded by David Pacheco, Joel Weinberger, Dan Kuebrich, and Dimo Bounov. In 2010, Weenix found its current home running on Bochs, a virtual machine including a simulated real machine. Chris Siden, Alvin Kerber, and Shaun Verch were the major contributors for this move.

The features of Weenix now include:

- Intelligent multitasking
- Virtual memory
- Terminal emulation
- Polymorphic file system support
- Advanced device support (including APIC and PATA with BMIDE)

1.4 Acknowledgements

Weenix would not have been possible without the help of many earnest, devoted individuals. Thanks first to Professor Tom Doeppner, for his patient guidance and leadership.

Thanks also to the many course TAs since the creation of Weenix.

'98-'99 Mike Castelle, David Powell, Caroline Dahllof, Tim Rowley

'99-'00 Thomas Crulli, Matt Eccleston, Keith Adams, Matt Ahrens, Tim Rowley

'00-'01 Matt Ahrens, Adam Leventhal, Matt Amdur, Pete Demoreuille, Ioannis Tsochantaridis

'01-'02 Pete Demoreuille, Peter Griess, Fareed Behmaram-Mosavat, Dan Polivy, Dmitriy Genzel

'02-'03 Kit Colbert, Sean Cannella, Albert Huang, Eric Schrock, Mark Ture

'03-'04 Susannah Raub, Dan Stowell, Luke Peng, Pat Sunday, David Reiss

'04-'05 Mark Johnson, Stacy Wong, Sean Smith, Dan Spinosa, Hari Khalsa

'05-'06 Lucia Ballard, Eric Tamura, Edwin Chang, Pat Culhane, Adam Fenn

'06-'07 Dan Leventhal, Aaron Myers, Adam Cath, Dave Pacheco, Joel Weinberger

'07-'08 Colin Gordon, Itay Neeman, Dan Kuebrich, Aurojit Panda, Lincoln Quirk, Owen Strain

'08-'09 Andres Douglas, Tim O'Donnell, Dimitar Bounov, Brandon Diamond, Travis Fischer, Allan Shortlidge

'09-'10 Spencer Brody, Robert Mustacchi, Chris Siden, Travis Webb

'10-'11 Chris Siden, Venkatasubramanian Jayaraman, Alvin Kerber, Matt Mallozzi, Ryan Zelen

'11-'12 Shaun Verch, Dan Kimmel, J. Rassi, Basil Crow, Irina Calciu, Nickolay Ratchev, Andrew Ayer, Marcelo Martins

In particular, many thanks to Keith Adams, Adam Fenn, Dave Pacheco, Dimitar Bounov, Alvin Kerber, and Chris Siden for their major contributions to the project.

Finally, thanks to the innumerable students who have implemented Weenix over the years. We hope the knowledge you gained in the process has served you well.

Chapter 2

Project Management

2.1 Introduction

The first section of this chapter guides the reader through the process of producing a Weenix assignment from the project's complete source code (useful for course instructors), and the second explains how to set up a development environment for implementing the assignment yourself (useful for students).

2.2 Preparing Weenix Assignments

This section will walk you though a sample run of Weenix, and then shows you how to remove sections of code from the full implementation to allow you or your students to re-implement them.

2.2.1 Quick Start: Getting Weenix Running

See instructions in the top-level `README` file.

2.2.2 Getting the Source

Requests for a copy of Weenix may be made via email to `weenix-devel@lists.cs.brown.edu`.

2.2.3 Dependencies

The following is required to run Weenix:

- `gcc`, the GNU C compiler
- `make`, the GNU build system
- `qemu`,the processor emulator that Weenix runs on top of

- `genisoimage` *or* `mkisofs` (necessary to generate the disk image used by the emulator)
- `python`, at least 2.6 (but 2.x only)
- `bash`

The following is optional, but recommended:

- `git`, the version control system
- `gdb`, the GNU debugger, and `xterm`
- `cscope` (for easier browsing of the Weenix source)

2.2.4 Configuration

Configuration for Weenix is available through editing the files `Config.mk` and `Global.mk`, which contain `VARIABLE=value` assignments in make syntax. `Global.mk` contains directives relevant to the environment that Weenix is running in (e.g. alternative locations for utilities like `python` or `gdb`), while `Config.mk` is used to configure values that affect the behavior of Weenix itself (such as what modules to enable during the compilation process, or the size of the virtual disk to create).

In particular, you should be aware of the directives that affect what assignment to enable. These are the first directives of `Config.mk`. For example, to enable the S5FS assignment, set the first three to `1` (`DRIVERS=1`, `VFS=1`, and `S5FS=1`).

2.2.5 Blanking Solutions

This section will teach you how to blank the solution for individual assignments. You will wish to do this before you attempt to complete the Weenix assignments (otherwise, you will find that they have already been done for you!) or before you assign them to others.

The script `make-weenix-repo.py` in the `tools/` directory performs this task. It removes the contents of specially-marked functions, putting a `NOT_YET_IMPLEMENTED()` marker in their place. It also initializes a fresh git repository in the root of the source tree for you (or your students) to clone from.

1. Obtain an unmodified copy of the Weenix source tree.

   If you skip this step, you will lose all of your work: the `make-weenix-repo.py` will obliterate any git history that exists in the current source tree, as well as the contents of assignments that are to be removed.

2. Change into the `tools/` directory.

3. Run `./make-weenix-repo.py --cutextra all --cutsource [ASSIGNMENTS]`, where [ASSIGNMENTS] is a comma-delimited list of the projects you want to assign (or "all", without quotes).

   For example, suppose you want to implement the S5FS and VM assignments. You should blank the solutions for these assignments, and leave the rest (PROCS, DRIVERS, and VFS) intact. This can be accomplished with the command:

   ```
   $ ./make-weenix-repo.py --cutextra all --cutsource S5FS,VM
   ```

   The order of the assignments is: PROCS, DRIVERS, VFS, S5FS, and lastly VM. Remember that each assignment depends on its predecessor in order to run. For example, if you wish to start with the VM assignment, do not blank any other assignments, or you will find yourself unable to run the VM code you write.

   As another example, suppose you want start fresh from the first assignment. Run:

   ```
   $ ./make-weenix-repo.py --cutextra all --cutsource all
   ```

4. Update the assignment directives in `Config.mk` to reflect the first assignment you will work on.

   These are described in detail in the Configuration section.

   For example, if you are starting with the VFS assignment, set your assignment directives to:

   ```
   DRIVERS=1
       VFS=1
     S5FS=0
       VM=0
   DYNAMIC=0
   ```

   If you are making use of the git repository, you will need to commit this change. For example:

   ```
   $ git commit -m 'Updated Config.mk assignments directives' Config.mk
   ```

Congratulations: you are now ready to start the assignment. If you need to distribute the assignment to others, have them copy (or `git clone`, if you are using git) this source tree to be their fresh copy to start working on.

2.3 Implementing Weenix Assignments

This section introduces some essential concepts in Weenix to e.g. a student who is interested in completing a Weenix assignment.

2.3.1 A Message to the Reader

Although this section is ostensibly about getting to know the Weenix codebase, we realize this is also the last part of the book we can be certain everyone will read, so we would like to give a few pieces of advice.

- It is possible that this is your first exposure to a large code base. You should definitely spend some time poring over the existing code, thinking about what to implement. Take a look in Appendix B to learn more about working in large codebases.
- For this and future assignments, it may be helpful to draw out a call graph (which functions call which) for the code you will be writing. Taking notes about the code as you read it is a useful skill for any codebase.
- Become an expert at using your chosen development tools. This will save you countless hours of time and energy, whether you learn to use a debugger (see Appendix C), text editor/IDE, cscope, version control, or something else. Many of these are mentioned in the appendix, so take a look there for some tips for getting started.
- Be sure to ask questions! The course staff is friendly and has a great working knowledge of the OS because they have implemented it themselves. Beyond that, getting Weenix working is far more rewarding if you understand *why* something is done a particular way, and the repercussions of choosing a different way to do something.
- Take breaks to shower. We will not presume to know the details of your friendships and love life, but we can only guess that following this advice will pay off in the end. `:-)`

2.3.2 Build System

The Weenix build system is based on the popular Unix utility `make`. It is meant to aid development by automating the compilation and linking of binaries, although it is used to automate several other things in the Weenix codebase as well.

In order to build the full operating system, `make` must be run in the top directory of the Weenix repository. Each `make` target corresponds to particular a set of commands listed in the `Makefile` located in the current directory. In the case of the top-level `Makefile`, `make all` simply descends into the `kernel` and `user` directories and executes `make` within each. The kernel and the userspace binaries can be built separately, by calling `make` from their respective directories. There are many different targets, such as individual object (`.o`) files, linked binaries and disk images, but the most useful targets are listed below.

To compile all the code using eight concurrent tasks, a user should run:

```
$ make -j 8 [all]
```

or potentially just:

```
$ make all
$ make       % defaults to target "all"
```

if the pretty output is more important than the speed of compilation.

Sometimes, such as after a change in the configuration settings or compiler flags, it is necessary to compile all the source code from scratch. However, `make` will only rebuild a file if it has been modified since the last time it was built. To delete all the constructed binaries and force `make` to compile from scratch, a user may run:

```
$ make clean; make all
```

2.3.3 Assignment Specification

To find out what to implement for a newly opened assignment, use the following command:

```
$ make nyi
```

This will produce a list of all the functions that are not yet written in the codebase with labels telling what file they are in and what assignment they are associated with.

Note that this command is only to be used with blanked assignments (otherwise, there's nothing to implement!).

2.3.4 Running

The `weenix` script has been provided for running the operating system inside a virtual machine or hardware simulator. It should be invoked from the topmost directory of the Weenix repository. To run it, one may simply use the command:

```
$ ./weenix
```

Until there is a functional on-disk file system, this is all that is necessary to run Weenix. However, at that point, the ability to create disks becomes very important. It is necessary to create a fresh disk the first time S5FS runs, and also after any time the disk is corrupted by a buggy kernel or unclean shutdown. The following command will run Weenix with a newly constructed disk:

```
$ ./weenix -n
```

At some point, it may be helpful to use a debugger to aid in the development of the operating system. The following command will begin Weenix in a virtual machine and attach `gdb` to it (if the virtual machine in use supports this option).

```
$ ./weenix -d gdb
```

These options (and their exciting longer names `--new-disk` and `--debug`) are also visible at the command line.

```
$ ./weenix -h
```

2.3.5 Making Changes

Any time someone works on a large enough project, they will inevitably make difficult-to-reverse mistakes or accidentally delete their work. The best way to prepare oneself for this inevitability is to use a version control system. The Weenix repository uses the source control system `git`, and everyone who undertakes a project of this magnitude should become at least competent enough to make commits, revert a file back to a previous commit, and make feature branches for each new addition. The following list of steps are a highly recommended way to implement a new piece of Weenix.

1. Make a new branch to develop in with a descriptive name.
2. Write code, committing relatively frequently to avoid losing any work.
3. Write tests for the new feature, running them frequently.
4. Make sure all the tests pass and that they comprehensively cover the set of requirements.
5. Push the changes back onto the main branch once everything is functional.
6. Back up the entire project frequently using an external hard drive or (even better) an online backup and recovery service.

Even if you don't follow these steps, please, for the love of all things holy, use source control. If you mess up your repository due to lack of experience, somebody will be able to help you out of the mess and you'll be back where you were before, but if you mess up an unprotected directory, there is no going back! Having a student who loses all their Weenix code is awful, and *being* that student has been a recurring theme in many of our worst nightmares.

Part II

Assignments

Chapter 3

Processes and Threads

3.1 Introduction

This assignment, colloquially referred to as "Kernel 1," will provide the basic building blocks for the Weenix operating system: threads, processes, and synchronization primitives. These objects are implemented fully in kernel code, and interactions with user-level processes and threads will not be possible until you implement virtual memory in a later assignment.

Writing an operating system can be complicated, so much of the code to do basic kernel operations such as memory allocation and page table management has already been written. These are either beyond the scope of the project or are too time consuming for too little value. While modifications to the support code are fully possible, significant changes to the provided interfaces make errors far more likely and make it harder to ask for help. However, it is important for you to be able to take ownership of the code. Though it is usually too time-consuming to write the entire system from scratch, by the end of the project, you should have a good (if high-level) understanding of all the code involved. This will certainly be true by VM, which will involve putting the final touches on the kernel.

At the end of this first assignment, the kernel will be able to run several threads and processes concurrently in kernel mode. It is important to emphasize that a strong test suite is critical. For this assignment, test code must be added directly into the boot sequence for the kernel, but in later assignments there will be several additional ways to add tests in a cleaner, more modular way.

3.2 Kernel Memory Management

As mentioned above, this has been fully implemented for you, but take a moment to look around `include/kernel/mm/kmalloc.h`, `include/kernel/mm/slab.h` and `kernel/mm/slab.c` to familiarize yourself with the memory management interface. The functions `kmalloc()` and `kfree()` exist as kernel-space worka-

likes for the standard library `malloc()` and `free()` functions. However, these should be used sparingly (for example, for incidental memory usage during testing). Most of the kernel memory management you will be doing using the `slab_allocators`. A slab allocator, as used in Solaris and Linux, works like a cache for memory chunks of a particular size. This reduces both the loss of memory through fragmentation and the overhead from manipulating the heap directly. Refer to `include/kernel/mm/slab.h` for the slab allocation and freeing functions you will be using.

3.3 Boot Sequence

Most of the boot sequence is handled by the support code. The last thing the boot loader does is execute the function `kmain()`, which initializes the support subsystems and then calls `bootstrap()`.

At this point, we are still in the boot sequence, which means that we do not have a thread context in which to properly execute. We cannot block, and we cannot execute user code. The goal of `bootstrap()` is to set up the first kernel thread and process, which together are called the idle process, and execute its main routine. This should be a piece of cake once you have implemented threads and the scheduler.

The idle process performs further initialization and starts the init process. For now, all of your test code should be run in the init process. When your operating system is nearly complete, it will execute a binary program in user mode, but for now, be content to put together your own testing system for threads and processes in kernel mode.

3.4 Processes and Threads

Weenix is capable of running multiple processes and threads. There are a few important things to note about the Weenix threading model.

- There is no kernel mode preemption in Weenix (all threads run until they explicitly yield control of the processor). It is possible (but not required) to implement user mode preemption. Think about why this is much easier to do.

- Weenix is only running on one processor, so synchronization primitives don't require atomic compare-and-swap instructions or memory barriers.

- Each process only has one thread associated with it, however the threading code is actually structured so that it would be easy to have multiple threads per process. For example, each process keeps a list of its threads, even though that list currently never has more than one entry. If a function seems unnecessary to you, think of it in the context of multiple threads per process. For instance, when exiting a thread, you must alert the process

that one of its threads exited, even though each process should only ever have one thread.

The lifecycle of threads and processes is relatively straightforward. When a process is created, a thread should be added to it and the process should be made runnable by adding its thread to the run queue. If a process exits and it still has child processes, it should reassign those children to the init process, which, after performing some system setup, will sit in a loop collecting orphaned processes. When a thread attempts to exit the current process, the process code must cancel any other threads in the same process (and join with them, if there are multiple) so that they can do any cleanup necessary for their current tasks. Once each thread finishes cleaning up its current task, it makes a call to the process code to indicate it is exiting so that the process can do any final cleanup on the thread data structure.

Once all its threads have exited, a process can exit and one of its ancestors (either its parent or the init process) will call `do_waitpid()` and clean it up fully. The reason for this somewhat odd deallocation system is that some elements of the process and thread data structures can only be cleaned up from another thread's context. During this assignment, you should determine what these items are, and clean up everything else in the process as quickly as possible. Also note that processes which are children of the idle process will not be cleaned up using this method; they are dealt with separately.

As a side note, you will be using linked lists extensively throughout Weenix. By this point, you may have looked at the code and seen that some data structures contain list objects or link objects. We provide you with a circular doubly-linked list implementation where the links are stored inside of the objects that are in the list (hence the link fields inside the objects). Most of this list implementation is provided as a set of macros which can be found in the codebase. You even get a couple of neat list iteration "primitives" that look like they're straight out of your favorite scripting language!

The trickiest parts of this segment are `do_waitpid()` and `proc_kill_all()`, not because they are conceptually difficult, but because it is very easy to accidentally introduce bugs that you will discover much later. As such, you should test the edge cases of these functions early and often throughout the development of your operating system.

3.5 Scheduler

Once you have created processes and threads, you will need a way to run these threads and switch between them. In most operating systems, you must worry about thread priorities or quality of service assurances, which requires wait queue as well as run queue optimizations, but the scheduler for Weenix consists only of first-in-first-out queues, a function to switch between the contexts of two threads, and a few higher-level functions to abstract away the details of the queues from the caller.

In particular, there is a single run queue from which threads are dequeued (only when the running thread yields control explicitly) and switched onto the processor. There are also many wait queues in the system, most of which will be used as a part of some mutex. When a thread reaches the front of its wait queue, it can be woken up, which involves putting it on the run queue, waiting for it to reach the head of the run queue, and then being switched onto the processor by some other thread running the switching routine.

Switching between threads is the most tricky code to write in this area, and should be done carefully so as not to interact poorly with hardware interrupts. In order to write a correct switch function, you must take the first thread off of the run queue, set the global variables for current thread and current process to point at the new thread and its process, then switch into the new context, all with interrupts blocked (using the interrupt priority level, or IPL). If the run queue is empty, it's possible that all otherwise-runnable threads are waiting for hardware interrupts, so you must have a way to check this as well. Don't forget that hardware interrupts, when not masked, can occur between any two code instructions.

3.6 Synchronization Primitives

Since the kernel is multi-threaded, we need some way to ensure that certain critical paths are not executed simultaneously by multiple threads. Once your scheduling functions work, you should be able to implement synchronization primitives as well. The only synchronization primitive used in Weenix is the mutex, whose implementation uses a single FIFO thread queue, although you may also implement condition variables, semaphores, barriers, etc. if you wish to use them.

3.7 Testing

It is your responsibility to think of boundary conditions which could potentially cause problems in your kernel. Test code is an important part of software development, and if you cannot demonstrate that your kernel works properly, we will assume that it does not.

As mentioned earlier, you should run all your test code from the init process's main function (for lack of a better location), although you can add a new file to hold your tests. To be in the best shape possible before moving on, you should be able to test all of the following situations.

- Run several threads and processes concurrently. Devise a way to show that multiple threads are running and that they are working properly.
- Demonstrate that threads and processes exit cleanly.
- Ensure all the edge cases of `do_waitpid()` work correctly.

- Try exiting your kernel both by running `proc_kill_all()` and by allowing all threads to terminate normally.
- Demonstrate that the synchronization primitives work.
- Create several child processes and force them to terminate out of order, making sure they are cleaned up properly.

Keep in mind that this is not an exhaustive list, but that you should certainly be able to demonstrate each of these tests passing by the end of this assignment.

Chapter 4

Drivers

4.1 Introduction

Now that you have processes and threads working properly, you can begin writing the device drivers for terminals, disks, and the memory devices `/dev/zero` and `/dev/null`. The code you will write for this part is in the `drivers/` directory.

There are two different types of devices: block devices and character devices. The disk devices you will be writing are block devices, while the terminals and memory devices are character devices. These are similar because they share the same interface, but there are significant differences in their implementation. In order to make the relationships between devices easier to manage we use some magic.

4.2 Object-Oriented C

As you look through the struct definitions of the different devices in the header files, you should notice that the structs for more specific types of devices (e.g. ttys and disks, referred to as the sub-struct from here on) contain the structures for the generic devices (e.g. block and byte devices, referred to as the super-struct) as fields. These are not pointers; they occupy memory that is part of the sub-struct. This way, we can use memory offsets to cast both from sub-struct to super-struct, and vice versa. So, given a pointer to the struct of a byte device which you know is a terminal device, just subtract the size of the rest of the terminal device struct and you have a pointer to the beginning of the terminal device struct. There is a macro provided for the purpose of doing these pointer conversions called `CONTAINER_OF`. In many cases, a more specific macro is defined using `CONTAINER_OF` which converts from a super-struct to a specific sub-struct (for an example, see `bd_to_tty` in `drivers/tty/tty.c`).

You should also notice that one of the fields in the device structs is a pointer to a struct containing only function pointers. The generic device types (e.g.

block and character devices) each specify that they expect certain operations to be available (e.g. `read()` and `write()`). This function pointer struct contains pointers to the functions which implement the expected operations so that we can perform the correct type of operation without actually knowing what type of device struct we are working with. The definitions of these function pointer structs are in the C source files of their respective types. Essentially, we are manually implementing a simple virtual function table (which is how C++ implements polymorphism).

4.3 TTY Device

Each tty device is represented by a `tty_device_t` struct. A tty consists of a driver and a line discipline. The driver is what interfaces directly with the hardware (keyboard and screen), while the line discipline interfaces with the user (by buffering and formatting their tty I/O). In Weenix, the main purpose of the tty subsystem is to glue together the low level driver and the high level line discipline. The advantage of this is that when a program wants to perform an I/O operation on a tty, it does not need to know the specifics of the hardware or the line discipline. All of these implementation details are abstracted away and dealt with by the functions in `drivers/tty/tty.c`.

Once you have a working virtual file system (after VFS) you will access the terminals through files, specifically in the files `/dev/tty0`, `/dev/tty1`, etc. Then you can read and write to the terminals using the `do_read` and `do_write` functions. Until then, you will need to use the `bytedev_lookup` function to get devices and then explicitly call the device's read/write functions in order to test your code. A convenient way to do this is by using the kernel shell. For more details, see the testing section at the end of this chapter.

4.3.1 Line Discipline

The line discipline is the high-level part of the tty. It provides the terminal semantics you are used to. Essentially, there are two things the line discipline is responsible for – buffering input it receives and telling the tty what characters to print. Buffering input is what allows users to edit a line in a terminal before pressing enter, or, as we call it, cooking the buffer. The buffer for a line discipline is split into two sections, raw and cooked, so that the buffer can be filled circularly. Before a circularly-contiguous segment of the buffer is cooked, the user is able to edit it by editing the current line of text on the screen. When the user presses enter, that segment of the buffer is cooked, a newline is echoed, and the text becomes available to the running program via the read system call. This is why the read system does not return until it receives a newline when reading from a terminal. For simplicity, do not store more input than you can put in the primary buffer (even though you could theoretically use the buffer that the waiting program provided as well).

The other important job of the line discipline is telling the tty which characters to echo. After all, when you type into a terminal, the characters you press appear on the screen. From the user's perspective, this all happens automatically. From your perspective (you being a kernel hacker), this must be done manually from the tty and the line discipline. The line discipline is also responsible for performing any required processing on characters which will be output to the tty via the write system call. In Weenix, the only characters that are not just echoed are the newline, backspace, and Ctrl+D characters.

The line discipline interface is located in `drivers/tty/ldisc.h`, and the default implementation of a line discipline, which is the only one you will be implementing, is located in `drivers/tty/n_tty.c`. (It is named `N_TTY` because that is the name of the default line discipline in the Linux kernel.)

4.3.2 TTY Driver

The tty driver is the low level part of the tty, and is responsible for communicating with hardware. When a key is pressed, the appropriate tty driver is notified. If a handler was registered with the driver via the `register_callback_handler` function, the key press is sent off to the handler. In our case, the tty subsystem registers the `tty_global_driver_callback` function with the driver, which calls the line discipline's `receive_char` method. Then, after any high level processing, any characters which need to be displayed on the screen are sent directly to the driver via the `provide_char` function.

The tty driver is already implemented for you. For anyone feeling adventurous, feel free to take a look at `drivers/tty/keyboard.c`, `drivers/tty/screen.c`, and `drivers/tty/virtterm.c`.

4.4 Disk Device Driver

A `block_device` is associated with each disk device. This structure contains information about what threads are waiting to perform I/O operations on the disk, any necessary synchronization primitives, the device ID, and any other information required by the ATA protocol. In Weenix, you can assume that all disk blocks are page-sized. We have defined the `BLOCK_SIZE` macro for you to be the same size as the size of a page of memory; use it instead of a hard-coded value.

4.5 Memory Devices

You will also be writing two memory devices, a data sink and source. These will not really be necessary until VFS. Still, these fit will with the other device drivers so they are included in this part of the assignment.

If you have played around with a Linux/UNIX machine, you might be familiar with /dev/null and /dev/zero. These are both files that represent memory devices. Writing to `/dev/null` will always succeed (the data is discarded), while

reading from it will return `EOF` immediately. Writing to`/dev/zero` is the same as writing to `/dev/null`, but reading any amount from `/dev/zero` will always return as many zero bytes (`'\0'`) as you tried to read. The former is a data sink and the later is a data source. The low level drivers for both of these will be implemented in `drivers/memdevs.c`.

4.6 Testing

As always, it is important to stress test your terminal code.

- Have two threads read from the same terminal, which will cause each thread to read every other line. If you're not sure why this is, ask.
- Make sure that you can input and output more data than can be held in the internal terminal buffer.
- Ensure that you can have two threads simultaneously writing to the same terminal.
- Stress test your disk code. This will not be needed until the S5FS assignment, but it is a good idea to make sure it works now.
- Make sure that you can have multiple threads reading, writing, and verifying data from multiple disk blocks.
- Think of ways that you can write to a disk and display the data stored there.
- Note that Weenix does not currently support Caps Lock

It is very important that you get this code working flawlessly, since it can be a constant source of headaches later on if you don't. Note that the disk driver only works with page-aligned data, so you should use `page_alloc()` to allocate the memory used in your test cases, not `kmalloc()`.

Since you should now have a functional tty, you should try using the kernel shell to test it out. Once you are confident in your tty code, try implementing your own kshell commands to run further kernel tests. Below is what to add to `initproc_run` (`#include "test/kshell/kshell.h"` at the top of `kernel/main/kmain.c`):

```
 /* Add some commands to the shell */
kshell_add_command("test1", test1, "tests something...");
kshell_add_command("test2", test2, "tests something else...");

/* Create a kshell on a tty */
int err = 0;
kshell_t *ksh = kshell_create(ttyid);
KASSERT(ksh && "did not create a kernel shell as expected");
```

```
/* Run kshell commands until user exits */
while ((err = kshell_execute_next(ksh)) > 0);
KASSERT(err == 0 && "kernel shell exited with an error\n");
kshell_destroy(ksh);
```

Chapter 5

Virtual File System

5.1 Introduction

The virtual file system, known as the "VFS," provides a common interface between the operating system kernel and the various file systems. The VFS interface allows one to add many different types of file systems to one's kernel and access them through the same UNIX-style interface: one can access one's MS-DOS files via the `vfat` file system just as easily as one would access various device drivers through the `dev` file system, kernel internal information through the `proc` file system or standard on-disk files through the S5FS file system. For instance, here are three examples of writing to a "file":

```
$ cat foo.txt > /home/bar.txt
$ cat foo.txt > /dev/tty0
$ cat foo.txt > /proc/123/mem
```

All of these commands look very similar, but their effect is vastly different. The first command writes the contents of `foo.txt` into a file on disk via the local file system. The second command writes the contents to a terminal via the device file system. The third command writes the contents of `foo.txt` into the address space of process 123.

Polymorphism is an important design property of VFS. Generic calls to VFS such as `read()` and `write()`, are implemented on a per-file system basis. Before we explain how the VFS works we will address how these "objects" are implemented in C.

5.1.1 Constructors

File systems are represented by a special type (a `fs_t` struct) which needs to be initialized according to its specific file system type. Thus for each file system, there is a routine that initializes file system specific fields of the struct. The convention we use in Weenix for these "constructors" is to have a function called `<fsname>_mount()` which takes in a `fs_t` object. Note that the `fs_t` to

be initialized is passed in to the function, not returned by the function, allowing us to leave the job of allocating and freeing space for the struct up to the caller. This is pretty standard in C. Additionally, some objects have a corresponding "destructor" `<fsname>_umount()`. Construction does the expected thing with data members, initializing them to some well-defined value. Destruction (if the destructor exists) is necessary to clean up any other data structures that were set up by the construction (such as freeing allocated memory, or reducing the reference count on a vnode).

5.1.2 Virtual Functions

Virtual functions (functions which are defined in some "superclass" but may be "overridden" by some subclass specific definition) are implemented in the Weenix VFS via a struct of function pointers. Every file system type has its own function implementing each of the file system operations. Our naming convention for these functions is to prefix the function's generic name with the file system type, so for example the `read()` function for the `s5fs` file system would be called `s5fs_read()`. Every file system type has a struct of type `fs_ops_t` which lists all of the operations which can be performed on that file system. In the constructor for the file system, pointers to these `fs_ops_t` are added to the `fs_t` struct being initialized. One can then call these functions through the pointers, and you have instant polymorphism.

5.1.3 Overview

This section describes how the VFS structures work together to create the virtual file system.

Each process has a file descriptor table associated with it (the `proc_t` field `p_files`). Elements in the array are pointers to open file objects (`file_t` structs) in a system-wide list of all `file_t` objects that are currently in use by any process. You can think of this as the system file table discussed in the "Operating Systems Design" lectures. Each process's array is indexed by the file descriptors the process has open. If a file descriptor is not in use, the pointer for that entry is `NULL`. Otherwise, it must point to a valid `file_t`. Note that multiple processes or even different file descriptor entries in the same process can point to the same `file_t` in the system file table. Each `file_t` contains a pointer to an active `vnode_t`. Once again, multiple system file table entries can point to the same `vnode_t`. You can think of the list of all active `vnote_t`s as the active inode table discussed in the "Operating Systems Design" lectures. Through the `vnode_t` function pointers you communicate with the underlying file system to manage the file the vnode represents.

With all of these pointers sitting around it becomes hard to tell when we can clean up our allocated `vnode_t`s and `file_t`s. This is where reference counting comes in.

Reference Counting

As discussed in the overview of VFS, there are a lot of pointers to `vnode_t`s and `file_t`s, but we need to make sure that once all of the pointers to a structure disappear, the structure is cleaned up, otherwise we will leak resources! To this end `vnode_t` and `file_t` both have reference counts associated with them. This reference count tells Weenix when a structure is no longer in use and should be cleaned up.

Rather than allocating space for these structures directly, the `*get()` functions described below look up the structures in system tables and create new entries if the appropriate ones do not already exist. Similarly, rather than directly cleaning these structures up, Weenix uses the `*put()` functions to decrement reference counts and perform cleanup when necessary. Other systems in Weenix use these functions together with the `*ref()` functions to manage reference counts properly.

For every new pointer to a `vnode_t` or a `file_t`, it may be necessary to increment the reference count with the appropriate `*ref()` method if the new pointer will outlive the pointer it was copied from. For example, a process's current directory pointer outlasts the method in which that pointer is copied from the filesystem, so you must use `vref()` on the `vnode_t` the filesystem gives you to ensure the `vnode_t` won't be deallocated prematurely.

Keeping reference counts correct is one of the toughest parts of the virtual file system. In order to make sure it is being done correctly, some sanity checking is done at shutdown time to make sure that all reference counts make sense. If they do not, the kernel will panic, alerting you to bugs in your file system. Below we discuss a few complications with this system.

Resident Page References

Because reading and writing data on a disk can be expensive operations to initiate, disk drivers do them in large chunks (usually block-sized). File systems take advantage of this by caching data blocks in memory so that future reads can access the data without going back to disk and future writes can be performed in the cache (which is much faster than writing directly to the disk, see the documentation on paging for information on how changes to the cache get propagated to the disk). In Weenix, this caching is done by `mmobj_t`s. Every time a block is cached, it has a pointer to the vnode that "owns" its data (and therefore adds to the vnode's reference count). The `vn_nrespages` field of the `vnode_t` structure keeps track of how many cached blocks belonging to the vnode are currently resident in memory. Therefore when `vn_refcount - vn_nrespages = 0` the only pointers to a vnode are its cached blocks so Weenix *could* safely uncache all blocks belonging to that vnode and then cleanup the vnode; however, this is not the most efficient behavior.

It is possible that even if Weenix is not currently using a vnode it will need the vnode again in the near future. It would be a waste to uncache all of the data already read into memory. Therefore the `vput()` will actually keep the

vnode and all of its cached data in memory (if the pageout daemon decides to clean up all of the cached data before any new references to the vnode are created, dropping the vnode's reference count to zero, then the vnode will still be properly cleaned up). However this brings up another complication. If a file is deleted from disk (fully unlinked) *and* the vnode is only being referenced by cached pages, then it is impossible for the VFS system to ever reference that vnode again, so the call to `vput()` should uncache all pages and clean up the vnode. For this reason the fs_ops_t structure provides a `query_vnode` function which `vput()` uses to ask the file system implementation if a file has been deleted.

Mount Point References

If mounting is implemented then the vnode's structure contains a field `vn_mount` which points either to the vnode itself or to the root vnode of a file system mounted at this vnode. If the reference count of a vnode were incremented due to this self-reference then the vnode would never be cleaned up because its reference count would always be at least 1 greater than the number of cached data blocks. Therefore the Weenix convention is that the `vn_mount` pointer does not cause the reference count of the vnode it is pointing to to be incremented. This is true even if `vn_mount` is not a self-reference, but instead points to the root of another file system. This behavior is acceptable because the `fs_t` structure always keeps a reference to the root of its file system. Therefore the root of a mounted file system will never be cleaned up.

It is important to note that the `vn_mtpt` pointer in `fs_t` *does* increment the reference count on the vnode where the file system is mounted. This is because the mount point's vnode's `vn_mount` field is what keeps track of which file system is mounted at that vnode. If the vnode where to be cleaned up while a file system is mounted on it Weenix would lose the data in `vn_mount`. By incrementing the reference count on the mount point's vnode Weenix ensures that the mount point's vnode will not be cleaned up as long as there is another file system mounted on it.

Mounting

Before a file can be accessed, the file system containing the file must be "mounted" (a scary-sounding term for setting a couple of pointers). In standard UNIX, a superuser can use the system call `mount()` to do this. In your Weenix there is only one file system, and it will be mounted internally at bootstrap time.

The virtual file system is initialized by a call to `vfs_init()` by the idle process. This in turn calls `mountproc()` to mount the file system of type `VFS_ROOTFS_TYPE`. In the final implementation of Weenix the root file system type will be `s5fs`, but since you have not implemented that yet you will be using `ramfs`, which is an in-memory file system that provides all of the operations of a S5FS except those that deal with pages (also, `ramfs` files are limited to a single page in size).

The mounted file system is represented by a `fs_t` structure that is dynamically allocated at mount time.

Note that you do not have to handle mounting a file system on top of an existing file system, or deal with mount point issues, but you may implement it for fun if you so desire, see the additional features section.

5.1.4 Getting Started

The virtual file system (VFS) is an interface providing a clearly defined link between the operating system kernel and the various file systems. The VFS makes it simple to add many different file systems to your kernel and give them a single UNIX-style interface: with a VFS, you can play music by writing to `/dev/audio`, you can list your processes by reading `/proc/`, and things you don't want to see to `/dev/null`. Linux supports lots of different file systems – e.g. users who don't want to copy all of their Windows files over to a Linux file system can keep them safely on an NTFS partition.

In this assignment, as before, we will be giving you a bunch of header files and method declarations. You will supply most of the implementation. You will be working in the `fs/` module in your source tree. You will be manipulating the kernel data structures for files (`file_t` and `vnode_t`) by writing much of UNIX's system call interface. We will be providing you with a simple in-memory file system for testing (`ramfs`). You will also need to write the special files to interact with devices.

Remember to turn the `VFS` project on in `Config.mk` and `make clean` your project before you try to run your changes.

The following is a brief check-list of all the features which you will be adding to Weenix in this assignment.

- Setting up the file system: `fs/vfs.c - fs/vnode.c - fs/file.c`
- The `ramfs` file system: `fs/ramfs/ramfs.c` (provided in the support code)
- Path name to vnode conversion: `fs/namev.c`
- Opening files: `fs/open.c`
- VFS syscall implementation: `fs/vfs_syscall.c`

Make sure to read `include/fs/vnode.h`, `include/fs/file.h`, and `include/fs/vfs.h`.

You can wait until the VM project to implement the special file functions which operate on pages.

5.2 The `ramfs` File System

The `ramfs` file system is an extremely simple file system that provides a basic implementation of all the file system operations except those that operate on a page level. There is no need for the page operations until VM, by which point

you will have implemented S5FS. Note that `ramfs` files also cannot exceed one page in size. All of the code for `ramfs` is provided for you.

5.3 Pathname to `vnode` Conversion

At this point you will have a file system mounted, but still no way to convert a pathname into the vnode associated with the file at that path. This is where the `fs/namev.c` functions come into play.

There are multiple functions which work together to implement all of our name-to-vnode conversion needs. There is a summary in the source code comments. Keeping track of vnode reference counts can be tricky here, make sure you understand exactly when reference counts are being changed. Copious commenting and debug statements will serve you well.

5.4 Opening Files

These functions allow you to actually open files in a process. When you have open files you should have some sort of protection mechanism so that one process cannot delete a file that another process is using. You can do that either here or in the S5 layer. Although the choice is up to you, it is somewhat easier to do it in the S5 layer and not worry about it here.

5.5 System Calls

At this point you have mounted your `ramfs`, can look up paths in it, and open files. Now you want to write the code that will allow you to interface with your file system from user space. When a user space program makes a call to `read()`, your `do_read()` function will eventually be called. Thus you must be vigilant in checking for any and all types of errors that might occur (after all you are dealing with "user" input now) and return the appropriate error code.

Note that for each of these functions, we provide to you a list of error conditions you must handle. It is Weenix convention that you will handle error conditions by returning `-errno` if there is an error (not `-1` as will eventually be returned to the user). A return value less than zero is assumed to be an error. You should read corresponding system call man pages for hints on the implementation of these functions. This may look like a lot of functions, but you write one system call, the rest seem much easier. Pay attention to the comments, and use debug statements.

5.6 Testing

You should have lots of good test code. What does this mean? It means that you should be able to demonstrate fairly clearly – via TTY I/O and debug state-

ments – that you have successfully implemented as much as possible without having S5FS yet. Moreover, you want to be sure that your reference counts are correct (when your Weenix shuts down, `vfs_shutdown()` will check reference counts and panic if something is wrong). Note that you *must* make sure you are actually shutting down cleanly (i.e. see the "Weenix halted cleanly" message), otherwise this check might not have happened successfully. Basically, convince *yourself* that this works and get ready for the next assignment – implementing the real file system.

Chapter 6

System V File System

6.1 Introduction

The System V File System, or "S5FS," is a file system based on the original Unix file system. Although it lacks some of the complex features of a modern file system, Weenix uses it because it provides an excellent introduction to the issues involved in writing an on-disk file system.

In completing the S5FS assignment, you will provide its implementation for the full VFS interface. You will come across many different S5FS objects that interact with each other. Most of these object types are on-disk data structures – that is, the memory they occupy is actually being saved to disk (the only S5FS type which is not backed by disk is the struct representing the file system itself).

6.2 Disk Layout

6.2.1 Superblock

The first block of the disk (block number zero) is called the superblock, which contains metadata about the file system. The important data fields inside it are the inode number of the root directory, the number of the first free inode, the

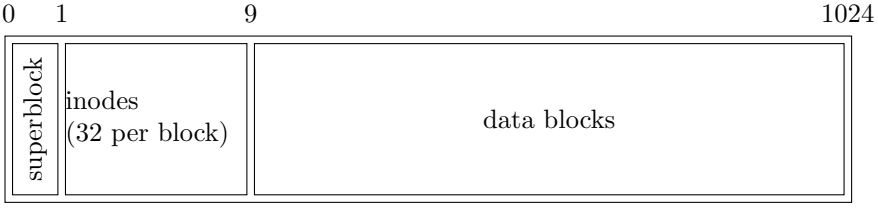

Figure 6.1: The default disk layout for Weenix.

first section of the free block list, the number of free blocks currently referenced in that section of the free block list, and the total number of inodes in use by the filesystem. The less important fields are the "magic number" for S5FS disks (used as a sanity check for the OS to determine that the disk you are reading is formatted as an S5FS disk), and the version number of S5FS we are using. The in-memory copy of the superblock is stored in a `s5_super_t` struct. For more information about the structure of the free block list, check out the section on data blocks below.

6.2.2 Inodes

Next, there is an array containing all the inodes for the file system. Each inode requires 128 bytes of space, and each disk block is 4096 bytes, so we store 32 inodes per block. Inodes are referred to by their index in the array, and are stored in memory as `s5_inode_t` structs. Each inode is either free, or it corresponds to some file in the file system.

If an inode is not free, it represents a file presently on disk. The inode holds of the size of the file, the type of the file (whether it is a directory, character device, block device, or a regular file), the number of links to the file from other locations in the file system, and where the data blocks of the file are stored on disk.

If an inode is free, it need only be marked as empty and contain the inode number of the next free inode in the free list (or `-1` if it is the last element in the list). This link is stored in the `s5_freesize` field (you can think of that field as being a `union` whose interpretation depends on whether the inode is free or not).

The link count on an inode has a slightly different meaning based on whether or not Weenix is currently running. If Weenix is shut down, the link count simply reflects the number of directory entries which point to this file. However, while the OS is running, calling `vget()` on some file for the first time will result in a call to S5FS's implementation of `read_vnode()`, which will increment the link count of the inode by one as long as the file is referenced by a vnode. Note that the link count will only be incremented when the file is first read from disk, not every time `vget()` is called on that file. Once the vnode's reference count drops to zero, the call to `vput()` will call S5FS's implementation of the `delete_vnode()` function, which will decrement the link count of the inode. This extra link count is used to prevent the inode from being deleted from disk as long as some vnode still references it (even if the file has been unlinked from disk) so that any process that is using the file should have no problem reading it. As such, the link count for a new file should be two: one link from its parent directory and one from Weenix. Note that this is slightly different for directories; see the section on directories for details.

As mentioned above, the location of the data blocks for a file are also stored in the inode for the file. The inode itself keeps track of `S5_NDIRECT_BLOCKS` data block numbers directly, but this is not usually enough for a large-ish file. Luckily, S5FS inodes for "large" files also contain a pointer to an indirect block, which

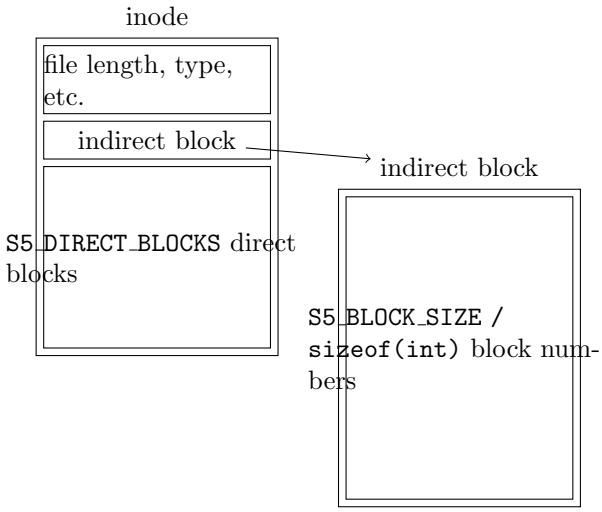

Figure 6.2: An inode with an indirect block (data blocks not pictured).

is a disk block filled with more data block numbers (in case it isn't clear: by "pointer" in this context we mean a disk block number, not a memory address). It is able to store up to `S5_BLOCK_SIZE / sizeof(int)` more block numbers. In a production file system, you should be able to support arbitrarily long files, which would require arbitrarily long indirect block chains (or, more likely, B-trees), but in Weenix we choose to only implement a single indirect block for simplicity. This means that there is a maximum file size; make sure you handle this error case correctly.

While the default disk size gives you space for several hundred data blocks, the indirect block will allow a single file to use thousands of blocks. This might seem unnecessary, however it allows for the implementation of sparse files. If a file has a big chunk of zeros in it, Weenix will not waste actual space on the disk to represent them; it just sets the block index to zero. When reading a file, if a block number of zero is encountered, then that entire block should consist of zeroes. Remember that zero is guaranteed to be an invalid data block number because it is the block number of the superblock.

6.2.3 Data Blocks

Data blocks are where actual file contents are stored. They occur on disk after the inode array and fill the remainder of the disk. For simplicity, disk blocks and virtual memory pages are the same number of bytes in Weenix, although this is not necessarily true for other operating systems.

The contents of the data blocks are obviously dependent on what file they are filled with (except for directories, which also use data blocks but have a special format described below) unless they are in the free block list. Instead of

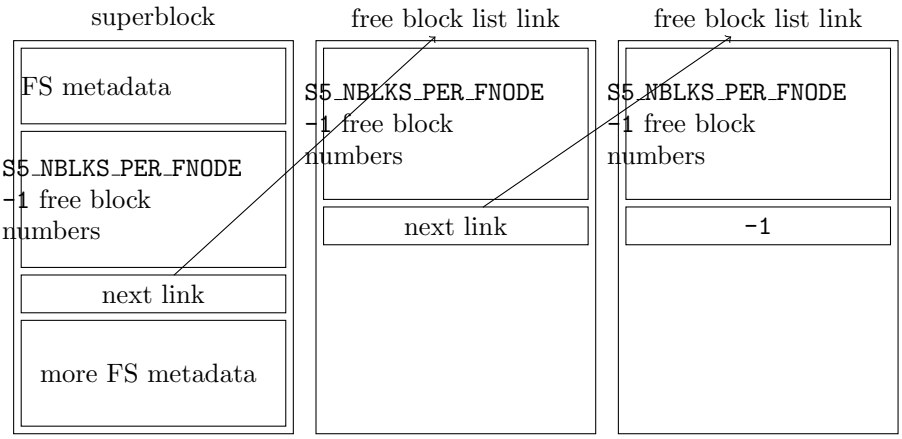

Figure 6.3: The free block list.

a free list where each link only points to one more link, which would be wildly inefficient due to frequent seek delays, S5FS uses a list where each link in the list contains the numbers of many free blocks, the last of which points to the next link in the free list. The first segment of the free list is stored in the superblock, where up to `S5_NBLKS_PER_FNODE` blocks are stored. The last element of this array is a pointer to a block containing `S5_NBLKS_PER_FNODE` more free blocks, the last of which is a pointer to a block with more free pointers, and so on. The last free block in the list has a `-1` in the last position to indicate there are no more free blocks. After the second-to-last free block in the superblock's array is used, the next set of free blocks should be copied out of the next block, and then the block they were just copied from can be returned as the next free page.

6.2.4 Directories

S5FS implements directories as normal files that have a special format for their data. The data stored in directory files is essentially just a big array of pairs of inode numbers and the filenames corresponding to those inode numbers. Filenames in S5FS are null-terminated strings of length less than or equal to `S5_NAME_LEN` (including the null character). Any entry with a zero-length name indicates an empty or deleted entry. Note that every directory contains one entry for "`.`" and one for "`..`", corresponding to the current directory and the parent directly, respectively, from the beginning of its existence to the moment it is deleted. The link count for a newly-created directory should be three (one reference from its parent directory, one from itself, and one from the running copy of Weenix that just created it).

6.3 Caching

At this point, you know a lot about how the on-disk filesystem looks and could probably inspect the disk block-by-block and understand what files are stored there. However, while working on this part of Weenix, you will not need to directly read and write from the disk, even in the most low-level functions. Instead, you will use the VM caching system to read blocks from disk into memory. You can then manipulate these pages in memory, and the pageout daemon will automatically handle writing them back to disk.

The Weenix caching system uses two different types of objects: page frames, which are each responsible for tracking one page/block of data, and memory objects, which are each associated with a number of page frames that hold the data for that memory object. In the codebase, the names for these objects are `pframe_t`s and `mmobj_t`s, respectively. Each memory object represents some data source, which could be a file, device, or virtual memory region. This means that page frames are used to reference the blocks of files, blocks of a device, and blocks of segments of memory. Specifically, page frames store some metadata about the page they hold and a reference to that page in memory. If a particular page of, say, a file hasn't been paged into memory, there will not yet be a page frame for that page.

To get a particular page frame from a memory object, you should call `pframe_get()` on the memory object you wish to get the page from. The data stored in the page is stored at the location pointed to by the page frame's `pf_addr` field. When a page frame has been modified, you should mark it as dirty so that it will be cleaned (the changes will be written back to disk) if necessary. The cleaning process uses callbacks in the disk's memory object to write the data back to disk.

To use an inode from disk, you must get its page from the disk memory object (the `S5_INODE_BLOCK()` macro will tell you which disk block to get) and then use the `S5_INODE_OFFSET()` macro to index into the page. When you are changing a file or the filesystem data structures, make sure that you remember to dirty the indirect block, inode, and superblock if necessary. Note the presence of the `s5_dirty_inode()` and `s5_dirty_super()` calls for this purpose. Remember that you should never clean pages yourself as either the pageout daemon or the Weenix shutdown sequence will take care of that automatically.

While working on S5FS, you may notice that there are two very similar methods for accessing the data on disk: calling `pframe_get()` on the memory object for the block device (the disk) and on the memory object for the file. Therefore, it can sometimes be confusing which one to use. Although this may sound like common sense, it is important that you use a file's memory object every time you are dealing with a file, and use the block device's memory object when you are implementing pieces of the filesystem that are "low-level enough" to know how a file is split across multiple blocks on disk. If you accidentally use the disk memory object instead of the file memory object, you will be short-circuiting a couple layers of abstraction that will be necessary later on.

6.3.1 Page Frame States

As alluded to above, there are several states which a page frame can be in. At boot time, all page frames are *free*, meaning they are waiting to be allocated. Some of the page frames will eventually be *allocated*, at which point they are added to the allocated page frame list and filled with data. If a change is made inside their associated page of memory, they should be marked as *dirty*.

The allocated page frame list is maintained to ease the implementation of the pageout daemon, which traverses the list and frees page frames and their associated memory pages. Note that any dirty pages will be cleaned (written out to disk) during this step, and non-dirty pages will simply be freed. The pageout daemon should only run when the system is out of memory, although you are allowed to call it other times as well if you choose. However, the use of the pageout daemon raises the question: how will you prevent a page frame from being freed while you are using it?

To solve this potential issue, page frames can also be *pinned* while they are in use. Pages can be pinned multiple times; as long as the pin count of a page frame is greater than zero, it is guaranteed not to be cleaned or freed. This will also be important when you implement virtual memory, since page frames for some types of virtual memory regions will not be backed by disk and can therefore never be safely cleaned or freed. When a page is pinned, it is removed from the allocated frame list and added to the pinned list, and the reverse happens when the last pin is removed.

The only time during S5FS where you should pin pages is when it is possible that your code will block after finding a page frame and before using that frame, since those are the conditions under which the pageout daemon could free the page, causing a memory corruption bug that will be difficult to reproduce. If you are unsure whether some code path might block, it is safer to pin any page frames you are using than not to do so. When you are done using the page frames, make sure you unpin them so that you do not run out of memory.

Finally, to protect page frames from making ill-formed state transitions, page frames can be marked as *busy* while they are in the middle of a state transition. Any concurrent attempts to modify a page frame (not necessarily the page it points to) should block until it is no longer busy.

6.3.2 Lazy Cleaning

One noteworthy feature of Weenix's caching system is that it does not uncache any pages of a file until either the system is out of memory (the pageout daemon may write back some or all of the changes at this point) or Weenix shuts down (the page frame system writes back its changes). This feature was added to improve cache performance for files that are closed and reopened frequently.

Each time `vput()` is called on a file, the S5FS implementation of `query_vnode()` is run to check if the file is still referenced by some directory on disk or some vnode in memory. If it has been deleted from every directory it was present in and you are `vput()`ing the final memory reference to the file, the blocks of that

file that are cached in memory can simply be freed (even if they are dirty) since nobody will ever need to use them again. Otherwise, they are left to be cleaned up at some later point. If you notice that your filesystem is working but none of your changes are being propagated to disk, you may want to check to see if this code path is running.

6.4 Error Handling

You always need to check for things that can go wrong. When an error condition occurs, you should return `-errno` where `errno` is the error number that indicates the type of error that occurred. For example, if you run out of free data blocks when attempting to write to a new block of a file, you should return `-ENOSPC`. You should *always* check the return value from non-void functions you call, and if the returned value is negative you often need to propagate it up. Returning `-1` is not correct.

However, it is also important that your VFS code check for as many errors as possible, so that each file system that it runs need not check the same error cases. If you know that some error condition should always be dealt with at the VFS layer, you should assert that it does not occur in the S5FS layer to identify bugs in your implementation while you develop.

We have tried to indicate some errors that you should check for in the comments, but we have probably not mentioned all of them, so you should go over your code thoroughly to make sure that you handle all possible errors.

6.5 Getting Started

You need to implement:

- The high-level system calls for S5FS (the VFS interface). Many of these will have a name such as `s5fs_[name]` with a corresponding VFS call named `do_[name]`.
- The low-level subroutines. These subroutines represent common functionality that may be useful to reuse in many of the high-evel system calls.
- A few memory management routines (such as `pframe_get()`). This is to understand a little better how the caching system works.

6.6 Testing

Your test cases should demonstrate clearly that all functions have been tested properly. While much of the functionality of S5FS will be tested by the tests you used for VFS, there are several cases that may require a bit more thought:

- Indirect blocks. (The `hamlet` file on your virtual disk is provided as an example of a file which needs an indirect block, but don't forget to check the case where you need to allocate one at runtime as well.)
- Sparse blocks.
- Running out of inodes, data blocks, or file length.
- Shutting down and rebooting does not erase your changes.

Be sure that your link counts are correct (`calculate_refcounts()` will calculate the counts for you). Note that you *must* make sure you are actually shutting down cleanly (i.e. see the "Weenix halted" message). Reference count issues will prevent Weenix from shutting down cleanly.

To ease the difficulty of debugging your file system code, we have provided a couple of utilities to help you develop. The `fsmaker` utility will come in handy for inspecting blocks, inodes, and other data structures on your virtual machine's disk. To read more about how to use `fsmaker`, run `fsmaker --help` from the root of your development directory. Your disks are stored in files on the host operating system (the `user/disk*.img` files), and must be passed to `fsmaker` as an argument. Also, running the `./weenix` script with the `-n` option will create a brand new disk for you and fill it with a bunch of sample files and directories. To begin this assignment, you must use this option at least once, otherwise you will not have a disk to work with.

Chapter 7

Virtual Memory

7.1 Introduction

At this point, your Weenix contains a threading library, some thin wrappers around device drivers, and basic file system support with a caching layer. By the end of this assignment, Weenix will be a full operating system. With the addition of virtual memory, your kernel will start managing user address spaces, running user-level code, and servicing system calls. After completing this project, everything you did before will seem insignificant.

This assignment is substantial, and also very prone to difficult bugs. Before you begin, make sure the rest of your kernel is functioning exactly as you expect. You will undoubtedly uncover bugs in old code throughout the course of this assignment, but minimizing the number you find before you start will be helpful. Make sure to start early and ask questions frequently; it is very easy to get lost in this assignment.

Because VM bugs can spring up in code you wrote months ago, this is where you will probably find out whether or not your implementations of the previous assignments are up to par. We would like to point out that there are several Weenix- and OS-specific debugging tools and techniques in Appendix C which will be *extremely* useful if you have not been using them so far.

7.2 Virtual Memory Maps

The first thing you should do in this assignment is write the code for managing a process' virtual address space. The virtual address space for a process (also known as its "memory map") is stored as a linked list of virtual memory areas (also referred to as "memory regions"), each of which correspond to some memory object which provides pages of memory to the process on demand. As you have likely already realized, this means that everything from files to disks can be mapped into the address space of Weenix processes, and it should now make even more sense why we used memory objects extensively in the last assignment

instead of reading and writing directly to disk. Of course, some memory areas will not correspond to existing data (the stack and heap, for instance). We will explain how that works in the section on anonymous objects.

In order to manage address spaces, you must maintain each process' list of virtual memory areas. Each memory region is essentially a range of virtual page numbers, a memory object, and the page offset into that memory object that the first page of the virtual memory area corresponds to. Make sure that you understand why these numbers are all stored at page resolution instead of byte (address) resolution. You must keep the areas sorted by the start of their virtual page ranges and ensure that no two ranges overlap with each other. There will be several edge cases (which are better documented in the code) where you will have to unmap a section of a virtual memory area, which could require splitting an area or truncating the beginning or end of an area.

While there is very little conceptually difficult code to write in this section of the assignment, off-by-one bugs are extremely common and become very difficult to track down later on, so unit-testing this code is a good idea.

7.3 Page Fault Handler

After your memory maps are working, you will need a way to actually load the data into memory when a process attempts to access it. This is done by the page fault handler. The page fault handler is triggered by a processor interrupt when a process attempts to access an address for which it has no lookup entry in the page table or the permissions on that entry do not allow the type of access that is being attempted.

At this point in the project, any page faults that have occurred have resulted in a kernel panic. That is because Weenix does not support kernel-level page faults, meaning that the entire kernel address space must reside in memory at all times. This functionality is written into a wrapper for the page fault handler you will write which short-circuits kernel page faults. More details on how this function works can most easily be found in the code.

The combination of the page fault handler and the virtual memory maps should be enough to get a very simple page fault to occur in a userland program. At this point, you can set up a userland program to run from inside the `init` process by running `kernel_execve()`, passing the path to any program on your (virtual) disk as an argument. Similar to the `exec()` system call, this will replace the memory map of the current process to set up another program to run, but it will be better than `exec()` in this case because the setup is done exclusively in kernel space, so it can be used before you have a fully functional userspace. When the program begins, it should cause a page fault to be generated. This is your first step towards having a functional userland.

A simple implementation of the page fault handler will be enough to start with, but eventually this will be a relatively logic-heavy function. First, the page fault handler should search for the virtual memory area containing the address that was faulted on. Then, the permissions of this area should be checked against

the flags variable that is passed to the handler, which tells the handler whether the attempted access is a read or a write. If the memory area containing the accessed page is not found or the permissions would make this access illegal, the current process is killed with an exit status of `EFAULT`. Of course, if Weenix supported UNIX signals, it would send a `SIGSEGV` signal instead.

Once the virtual memory area is found, Weenix must search for the missing page and map it into the page table of the current process so that the access can be retried using the virtual address of the page that's being added and the physical address where it resides. Fetching the missing page will require a lot of help from the page frame caching system, namely for looking up the page and dirtying it if the access is a write. This, in turn, will rely on two new types of memory objects you will need to implement.

7.3.1 The Memory Management Unit

In order to map the virtual address to its corresponding page of memory, you will need to use the page table functions. A good portion of memory management is done for you, but you will have to fill in page table entries when page faults occur, flush the translation lookaside buffer (TLB) when necessary, and manage copy-on-write pages yourself. You will also need to make sure that pages which are not backed by files remain pinned, so they do not get paged out by accident.

7.4 Memory Management

As you have implemented it currently, the caching layer of Weenix works exclusively for pages of files or disks that have been mapped into memory. You will extend it by creating memory objects which will provide two new types of memory which are not backed by any on-disk structures.

7.4.1 Anonymous Objects

So far, you have used the memory objects of your block device and files to fill page frames as you needed data from disk, but it does not make sense to back some virtual memory areas, such as a process' stack, with data on disk. What you often want is objects which initialize pages by filling them with zeroes and pin their pages in memory as long as the process is using them. These are known as anonymous objects since they are not backed by any persistent data (which would have a filename associated with it). Anonymous objects are relatively simple to implement, so look for a better description of how they work in the code comments.

Notably, anonymous objects cannot be paged out in Weenix. The designers chose not to implement this feature because memory pressure will rarely be an issue in your operating system, and implementing a swap space is not terribly interesting or vital to implement as a result.

7.4.2 Shadow Objects

Anonymous objects are easy to implement, however you will also need a much more sophisticated form of memory object called a shadow object to implement `fork()`. These will be used to implement copy-on-write for privately-mapped blocks that are accessed after forking. Because of how involved shadow objects must be, you should refer to lecture slides or the book for more general information about how and why they are used. The rest of this section will only cover how to implement them in Weenix.

To implement shadow objects, it will be extremely helpful to understand how the methods of memory objects are called during a page lookup or dirty operation. If you don't remember this well from the last assignment, we recommend that you go back and either re-read the relevant sections of the last assignment or search through the code paths in question and draw a graph showing what functions in the page frame system call what functions in the related memory objects.

The main difference between shadow objects and other types of memory management objects is that shadow objects can be part of arbitrarily long chains of memory objects. Therefore, many calls to shadow objects will be rerouted to the object that is being shadowed, or occasionally to the root object in a tree of memory objects, which cannot be a shadow object. At a high level, this is similar to how file memory objects forward requests to the disk memory object, but in practice it ends up seeming a lot more recursive when implementing shadow objects since there is no translation layer as there was between file block numbers and disk block numbers. However, shadow objects are still responsible for storing some data and, more importantly, causing copy-on-write to work after a `fork()` has taken place.

One potential problem with shadow objects is that the chains must be cleaned up when the process that creates them exits to avoid temporary memory leaks. Ideally, this could happen at process exit. A process exiting might cause a shadow object's refcount to drop to one, at which point the pages attached to the object could be reassigned to the single shadow object beneath it, and the object itself could be deallocated. However, this would require the shadow object to know what its remaining child is and, at the moment, shadow objects do not maintain a list of their children.

This apparent design flaw leaves two other avenues for shadow chain cleanup. First, there is a shadow daemon known as `shadowd` which was built for this purpose. It should be invoked when the kernel is out of memory (this code is already written) or when a shadow object which can be cleaned up is created (you can tell this by checking for it when you remove either of its child shadow objects). To enable the shadow daemon, just set `SHADOWD=1` in the project environment settings. The second method would be to collapse shadow chains during `fork()`. This requires a relatively easy traversal of the forking process' object chains, where you shift the pages from any objects with a single child down to their child and then deallocate the objects.

7.5 System Calls

System calls are the only way user processes can communicate with the Weenix kernel directly. The way that system calls are generated from user space is by generating a software interrupt (using the x86 `int` instruction) with the arguments to the system call stored in the registers or on the stack. This causes an interrupt in the kernel, where the number in an agreed-upon register designates which system call is being used, and then the corresponding system call function is actually called to handle the request after the arguments have been parsed out of their registers. Most of the system calls have already been written for you, however, in order to give you some understanding of the process involved, you will need to write a few yourself.

7.5.1 Kernel System Call Interface

You will need to implement the kernel targets for `read()`, `write()`, and `getdents()`. While most of what you need to do should be pretty self-explanatory after reading through other system call implementations, you must also write two helper functions to check accesses to user memory from within the kernel.

7.5.2 Accessing User Memory

The code to handle traps and access user memory from the kernel has been written for you. However, many of these functions need to check to see if a region of user memory is a valid section of the process' address space. To check this, you will need to implement `addr_perm()` and `range_perm()`, which will rely on your virtual memory map code.

7.5.3 Running Userland Programs

Once you have implemented the page fault handler, anonymous objects, and `write()`, you should be able to run a variety of simple user-level programs. Of course, the first you should try to get running should also be one of the simplest, so we recommend `hello`, which should print "Hello, world!" to the screen. To run correctly, this will require a mostly-functional page fault handler to fill in pages as the process attempts to access them; otherwise, the operating system will probably go into an infinite loop, trying to access the same address over and over using the page fault handler, but never adding the correct entry to the page table. Some other simple programs that you should be able to run are `args` and `uname`.

If you are having trouble getting `hello` to run and suspect that your anonymous object or `write()` implementations might be at fault, you should try the `segfault` program instead and ensure that it exits with a status of `EFAULT`. If it doesn't (if, for instance, you run into the infinite loop problem described above), this means the bug is probably in your page fault handler.

After getting `hello` or `segfault` running, congratulations! You've just gotten your first userland program working! Celebration techniques are myriad, but we recommend dancing around a bit, and maybe taking a shower.

At this point, it will be useful to look at the debugging appendix covering how to inspect the progress of a user-level process using a debugger. Although you may not need it yet, we assume that you will want it very soon.

7.5.4 VM-Related System Calls

After you get some initial test programs running, you can start to think about implementing a variety of VM-related system calls. For the functions in this section, we recommend that you check out the documentation in the `man` pages for more information.

`mmap()` and `munmap()` are the most simple and obvious of the functions in this category. They allow user processes to map files into memory, create private or shared memory regions, and remove areas of their address space. The majority of these functions will end up being error-checking, since you wrote the main logic for them in the virtual memory map code. Note that the Weenix memtests expect you to use the `VMMAP_DIR_HILO` flag.

`brk()` is similar in conceptual difficulty. Calling `brk()` changes the length of the memory region acting as the heap, but the pointer passed as an argument to `brk()` is not required to lie on a page boundary, and the beginning of the heap sometimes starts halfway through the last page of another memory region. This means that the edge cases for `brk()` can be a bit annoying, but there's nothing conceptually difficult to grasp here. There are some robust user-level tests for much of this functionality, so rather than spending a lot of time getting it right before testing, we recommend starting with something naive and gradually fixing it to pass the tests after you can run them in userland.

7.6 `fork()`

Although it is also a VM-related system call, `fork()` is an entirely different animal from `mmap()` and friends. A good implementation of the previous sections is essential; `fork()` is complicated enough without having to debug the rest of your VM code at the same time. The `man` pages, while useful as always, will not be as helpful for `fork()` as for the other system calls, so most of the documentation for `fork()` is given here.

`fork()` is a moderately complicated system call. We present it here as one long algorithm, but it will make your life much easier if you break it down into separate subroutines. Close attention to detail will help you; an under-debugged `fork()` can cause subtle instabilities and bugs later on.

Bugs in the virtual memory portion of `fork()` tend to cause bizarre behavior: user process memory may not be what it ought to be, so almost anything can happen. The user process may end up executing what should be data, jumping into the middle of a random subroutine, etc. These sorts of bugs are *very*

difficult to track down. For this reason, you should code more defensively than you may be used to. Assert everything you can, `panic()` at the first sign of trouble, and include apparently unnecessary sanity checks.

Above all, be sure you really understand the algorithm before you start coding. If you try to implement it before you understand what you are trying to do, you will write buggy code. In all likelihood you will then forget that you have written buggy code, and waste time debugging code that you should have thrown away. We know this because it has happened to us.

Note that these steps are not all in the correct order; consider the order in which you do them, keeping in mind what kind of cleanup you will need to do if one of them fails. Look out for steps which cannot be undone.

- Create a new process using `proc_create()`.
- Copy the `vmmap_t` from the parent process into the child using `vmmap_clone()` (which you should write if you haven't already). Remember to increase the reference counts on the underlying memory objects.
- For each private mapping in the original process, point the virtual memory areas of the new and old processes to two new shadow objects, which in turn should point to the original underlying memory object. This is how you know that pages corresponding to this mapping are copy-on-write. Be careful with reference counts. Also note that for shared mappings, there is no need to make a shadow object.
- Unmap the userland page table entries and flush the TLB using `pt_unmap_range()` and `tlb_flush_all()`. This is necessary because the parent process might still have some entries marked as "writable", but since we are implementing copy-on-write we would like access to these pages to cause a trap to our page fault handler so it can dirty the page, which will invoke the copy-on-write actions.
- Set up the new process thread context. You will need to set the following:
  - `c_pdptr` - the page table pointer
  - `c_eip` - function pointer for `userland_entry()`
  - `c_esp` - the value returned by `fork_setup_stack()`
  - `c_kstack` - the top of the new thread's kernel stack
  - `c_kstacksz` - size of the new thread's kernel stack
- Copy the file table of the parent into the child. Remember to use `fref()` here.
- Set the child's working directory to point to the parent's working directory. Once again, don't forget reference counts.
- Use `kthread_clone()` (which you should write if you haven't yet) to copy the thread from the parent process into the child process.

- Set any other fields in the new process which need to be set.
- Make the new thread runnable, which will add it to the run queue.

Remember that the only difference between the parent and child processes is the return value of `fork()`. By 32-bit x86 convention, this value is returned in the `eax` register, which should be set in the context values of both threads. You should also revisit your implementation of the `proc_exit()` function to make sure that your implementation is releasing all resources it should.

Note that a simpler, less correct implementation of `fork()` can function without actually using shadow objects, as long as you don't care what happens to whichever process (parent or child) wakes up last from the syscall. If you're having trouble getting shadow objects to work correctly, you can write `fork()` without them for testing purposes.

7.7 Odds and Ends

Finally, there are a number of other functions which you might remember seeing in earlier assignments spread throughout the kernel which you need to find and either write or update. These functions are all fairly small, but if you miss one, some things will break. Two examples are `special_file_mmap()` and `proc_kill_all()`. Once you get the last of these finished, you should be able to test your kernel with any binary file you find on the Weenix file system.

7.8 Testing

Testing your code at this point becomes rather difficult, since you must be able to create data and text in user land and execute it. This is an order of magnitude more difficult than creating kernel-mode threads as you have in past assignments. Thankfully most of the gory details have been taken care of for you (take a look at `kernel/api/elf32.c` and `user/ld-weenix/` if you are a masochist).

7.8.1 Userland Tests

Once you have functioning userland execution and a working `fork()` function, you are ready to complete your Weenix system by running the userland binaries we provide for you. All you need to do is call `kernel_execve()` in your init process. You should execute the binary `/sbin/init`, which should start 3 shells (one in each terminal window). These shells will allow you to execute any of the provided binaries (roughly in order of difficulty):

- `/usr/bin/segfault` - Even simpler than hello, this should just segfault on address `0x0`. Good if you're having a lot of trouble getting hello to run.

- `/usr/bin/hello` - A simple "Hello world!" test. Getting this to execute properly should be a big step in VM.
- `/usr/bin/args` - Prints command arguments.
- `/bin/uname` - Prints system information.
- `/usr/bin/kshell` - Traps into the kernel and starts a kshell.
- `/bin/ls` - List the contents of a directory.
- `/sbin/halt` - Kills all processes and shuts the system down.
- `/bin/sh` - The shell itself. Yay subshell fun!
- `/usr/bin/spin` - Executes "`while(1);`".
- `/usr/bin/forkbomb` - A forkbomb test which should theoretically run forever.
- `/usr/bin/stress` - A test to stress various parts of your system and then run a forkbomb.
- `check` - Contains checks for various test cases (this is a shell built-in command).
- `/usr/bin/vfstest` - Lots of VFS tests (error conditions, etc.).
- `/usr/bin/memtest` - Lots of memory management tests (mmap and brk).
- `/usr/bin/eatmem` - Devours kernel memory.
- `/bin/ed` - ed is the standard text editor.

The shell also has a bunch of builtins. Type `help` in a shell to see a list of them. In particular, `repeat` and `parallel` can be very useful for stress testing your kernel.

7.8.2 A Relatively Difficult Test Suite

In addition to just having commands which work individually, you should be able to stress the hell out of your system. Run lots of difficult commands (`forkbomb`, `eatmem`, etc.) simultaneously, use different terminals at once, `halt` in the middle of all this, and so on. This type of testing can frequently be quite random, so here is a more systematic list of some things you can try. Make sure you start with a fresh disk.

- `cat hamlet`
- `cat hamlet > hamlet2`

- `halt` (to shut down, then make sure the changes still exist on disk when you reboot)
- `rm hamlet`
- `cat /dev/null > foo`
- `ln foo bar`
- `cat README > foo`
- `cat bar`
- `check all` (do this three or four times in a row)
- `vfstest`
- `memtest` (do this three or four times in a row)
- `parallel vfstest -- vfstest`
- `parallel memtest -- memtest`
- `vfstest` and then `halt` while running (use `repeat` to re-run `vfstest` if it finishes too quickly)
- `memtest` and then `halt` while running
- `forkbomb` and then `halt` while running
- `forkbomb` and then `eatmem`

An easy way to make these tests harder is to check the kernel memory allocators for any leaks. See the debugging appendix to see how to do this. You may also want to test what happens when Weenix runs out of disk data blocks. To do this, set the `DISK_BLOCKS` variable to 2048 if it is not already, and then re-run Weenix with a fresh disk and execute the commands below.

- `cat hamlet >> hamlet` (this should reach the maximum file size and then exit)
- `vfstest` (this should work - the disk is not yet filled)
- `cat README >> README` (this should fill the disk but not quite reach the maximum file size)
- `vfstest` (this should fail to run most of the tests due to no disk data blocks being available)

7.8.3 Dynamic Linking

Once you feel everything is in good shape, enable the `DYNAMIC` variable and recompile the project from scratch. This will cause your userland libraries to be dynamically linked, which puts much more stress on your VM (especially `mmap()` code). This essentially adds another layer of indirection between the executable being run and the library calls it's attempting to run, where `ld-weenix` can link the library calls that are used into the original binary at runtime. Unfortunately, this also makes it even more difficult to debug what the user process is doing; if you are interested in setting breakpoints in the user process with dynamic linking, check out the debugging appendix for more information.

Turning dynamic linking on will make the above tests even more thorough. For instance, the test using `forkbomb` and `eatmem` simultaneously is notoriously hard to get right in the presence of dynamic linking - a phantom bug might cause your `pageoutd` to thrash back and forth between two pages if you're not careful.

7.9 Conclusion

We hope you've enjoyed working on Weenix and that you learned a lot. Good luck finishing your project, and don't forget to read the debugging appendix if you run into problems or need more ideas about where to look!

Part III

Appendices

Appendix A

Extra Assignments

The features listed on this page are not extra credit. **There is no extra credit in Weenix.** Your grade is based on how well you complete the core Weenix requirements. Therefore trying to implement these things can only hurt your grade and distract you from what is really important in life. The only thing to be gained by foolishly ignoring this warning is a deeper understanding of Weenix and bragging rights. **We will look unfavorably on someone who has implemented some of these, but has problems with the core Weenix projects.**

Some items on this page include descriptions on when it is feasible to implement them, be aware however that you should always keep around a copy of your work which does not contain your work on extra features in case you break something. **Having a broken Weenix because you tried to implement one of these features is not acceptable.** This is particularly important to remember because you might implement a feature between VFS and S5FS only to find later when working on VM that you broke something badly.

Also note that the course staff will only be able to provide limited help with these features. Some of the features have been implemented in the staff version of the code, others have been attempted and abandoned, still others are random whims which may or may not be impossible.

A.1 Realistic projects

If you ignored the warning above then this is the place to start. These are features which are known to be possible because someone has either done so or it has been planned. This means you will get some description here on how to implement it and there might even be helpful code already in Weenix.

A.1.1 Multithreaded processes

This task has a bunch of places in code that are marked by the __MTP__ symbol (which must be enabled by setting "MTP = 1" in Config.mk), and the kernel is designed around it anyway. This is a relatively straightforward addition to the kernel.

A.1.2 Current working directory

This will add a system call which looks up the full path name of the current working directory of the current process and is marked in the codebase with the symbol __GETCWD__, which can be enabled by setting "GETCWD = 1" in Config.mk. This should not be too difficult at all.

A.1.3 File system mounting

One important missing feature in our VFS implementation is mount points. We have a root file system (ramfs if you are working on VFS, s5fs if you are working on S5FS), but normal UNIX allows you to access multiple file systems by "mounting" additional file systems on directories in your virtual file system layer.

For more information on mount points we refer you to the mount(2) man pages, the umount(2) man pages, the lecture slides, and the text book.

Before you begin, there is already a significant amount of code in Weenix for this feature; however, by default it is not compiled. To compile this code change the line "MOUNTING = 0" in Config.mk to "MOUNTING = 1". Also, remember to run:

```
$ make clean
```

You should use cscope to search for all instances of the C symbol __MOUNTING__ in Weenix to see exactly what has changed to allow mounting to happen. Notice that struct fs in fs/vfs.h now has a new field called vn_mount.

The biggest change caused by changing the MOUNTING flag is the behavior of vget() (even though very little code changed, the behavior is completely different). The easiest way to think of the behavior of the new vget() is like this:

```
vnode_t* new_vget(args) {
    vnote_t* vn = old_vget(args);
    if (!error) {
        return vn->vn_mount;
    } else {
        return error;
    }
}
```

The purpose of this change is to make the integration of mounting as seamless as possible. Normally `vn->vn_mount == vn` and therefore most of the time this new behavior is identical to the old one; however, simply by setting the `vn_mount` field the `vget()` function automatically traverses into mounted file systems for us. The only cases we need to worry about are when we are leaving a mounted file system (e.g. following a `..` path from the root of a mounted file system).

The easier part of implementing mounting is filling in some functions which have been left blank (but fully commented!) for you. In `fs/vfs.c` this is:

```
int vfs_mount(struct vnode *mtpt, fs_t *fs);
int vfs_umount(fs_t *fs);
```

in `fs/vfs_syscall.c` there is also:

```
int do_mount(const char *source, const char *target,
             const char *type);
int do_umount(const char *target);
```

You will also want to read the Hackers Guide section about reference counting for information about special conventions used when reference counting mount point `vnode_t`s.

Now comes the hard part: implementing these functions is not enough. As was noted in the previous section, setting up the `vn_mount` field will allow us to enter mounted file systems. However following `..` paths out of mounted file systems still needs to be special cased. You will need to think about the code you wrote for VFS and which code will need to be different in order to handle mounting correctly. The amount of code you will need to write is small, but you need to find the right functions to write it. All of the code should go into functions which you wrote for VFS. You do not need to modify functions which you wrote for other projects or functions which you did not write originally. Also remember some of the system calls have errors specific to mounting which you might not have worried about before (e.g. What happens when you try to link a file from one file system onto another? Why is this an error?).

A.1.4 User preemption

Definitely possible. People have done it for the old version of Weenix (running on the Brown Simulator) but due to the changes in how timer interrupts work between the old Brown Simulator and the new architectures it is not trivial to port their work over. The basic idea is that a timer interrupt is scheduled to occur every several milliseconds. The interrupt context then looks at the thread which was interrupted. If the thread was in user space it is safe to just put that thread on the run queue and switch into the context of another thread (thus preempting the user land process). If the interrupted thread was in kernel land we do not want to arbitrarily preempt it (preemptable kernels lead to kernel hacker hell). Instead, we set a flag on the thread to mark it as preempted and allow it to continue. When a thread returns from kernel land into user land the

flag should get checked and if it is set the preemption should happen at that point. Make sure you understand all of this before you start or things will get messy.

So, do you gain anything from doing this? Actually, the effects are very visible and very satisfying. You will be able to run `vfstest` and `cat hamlet` at the same time and they will all look like they are running at the same time (assuming you set a good preemption time) while your fellow students will have very visible times when one operation stops working while the other is running. Also, your Weenix will not hang while you `cat /dev/zero` to `/dev/null`.

To get started on this enable the compile time flag and check out the code in `kernel/util/time.c`. It sets up the timer interrupts but you have to fill in what to do once they happen.

A.1.5 Asynchronous disk driver

It would be cool to support an asynchronous disk driver. To do this would require a decent amount of restructuring in the driver code, but is doable given a bit of time and effort. After this is done, you could potentially add asynchronous user I/O support as well (although this is a harder problem because in most Unix distributions the notification of a completion is done through signals).

A.1.6 Better scheduling

After you have finished implementing processes and threads, you should know the basics, but it might be a bad idea to try making too many big changes to Weenix before you have really gotten exposed to everything.

The current Weenix scheduler is rather primitive as it has no sense of priorities and makes no attempt to do any "clever" scheduling, it is purely first come first serve. Here are some helpful suggestions:

- Professor Doeppner's lecture (only available to Brown students) on scheduling contains plenty of information on different scheduling algorithms (you might also check the textbook).
- The CS167 students implement a slightly more advanced scheduler for their threading library assignment. You might be able to get some inspiration from that handout or the support code.
- If you decide to pursue user space preemption you might be able to tie it in nicely with a more advanced scheduler (e.g. threads which have been running for a long time and keep getting preempted should get a lower priority than threads which have just recently woken up after being asleep. As described in the lectures on scheduling, this can help because the long running programs tend to be background processes, while things which have just woken up are probably user processes which just received input).

These haven't been implemented yet, although doing so theoretically wouldn't be too hard. The downside is that it will probably not have very visible effects, but you might be able to think of a good way to show off its effects that we have not.

A.1.7 Block device special files

Tricky and yet not so tricky. Talk to Robert Mustacchi about this if you're interested.

A.2 "Abandon all hope, ye who enter here"

If you ignored the Dante quote above then this is the place to start. These are the features that we know are either very difficult or completely impossible. However, we would hate to stop a truly foolish individual from banging their head against the keyboard until one of these features pops out.

A.2.1 Signals

Signals are cool, but they get pretty ugly pretty fast when you start doing stack manipulations in assembly. Not a serious suggestion currently as it would just severely complicate the codebase.

A.2.2 Networking stack

If you've taken a networking class where you implemented TCP/IP, you could try this one out. However, before you start, don't forget to write an Ethernet driver from scratch.

A.2.3 Multi-core/processor support

Two big problems with this, both of which are impossible to overcome.

A.2.4 X window system

Depends on signals, graphical display driver with higher resolution than $80x25$, mouse driver, etc.

A.2.5 Users

Bonus points for capability-based security.

A.2.6 64-bit support

This one is obviously trivial. It's amazing this hasn't been done yet.

A.2.7 Kernel preemption

Would require totally redesigning the Weenix kernel to make it threadsafe. If this seems reasonable, you are misunderstanding something. Userspace preemption is similar and doesn't require a full rewrite of Weenix. Also note that most production kernels don't even support this (because even after you get it working, it's an absolute nightmare to maintain, and it typically adds very little value).

Appendix B

Tools for Working in a Large Codebase

B.1 Version Control with `git`

Git is a fantastic tool for version control. There are many tutorials available online that give a good introduction to Git, but one of the most highly recommended is on the Git website itself. If you prefer to work by example, try the Git immersion tutorial.

B.2 Taking notes

At risk of seeming low-tech, using code tags such as `TODO`, `XXX`, and `FIXME` with helpful notes about what you were thinking when you noticed something was broken or unfinished will be incredibly useful. This also allows you to search for remaining tasks by using `grep` (or, if you're using Git) `git grep`.

Even simpler than that, taking notes about the code you're developing or drawing call trees to visualize the expected call stacks can be incredibly helpful for learning your way around a new codebase.

B.3 Text Editors and IDEs

Because the fights between text editors are near-religious in severity, we won't recommend a particular code editor here, but we do recommend that you find a "serious" text editor with some built-in niceness to work in while you code. These include everything from Vim and Emacs (which typically run at the command line and can run commands without leaving the editor) to Eclipse (which does about a thousand things in addition to being a code editor). Some of the nice things that you've probably learned to love about IDEs can be

approximated quite well in more stripped-down environments as well if you use `cscope` to index your project.

B.3.1 Integration with `cscope`

Basically every major text editor has a `cscope` extension or bundle which allows you to do forward ("Where is this function defined?") and reverse ("Where is this function called?") searches, automatic text completion for function and variable names, etc. If you choose to use a text editor without these functions built-in, we highly recommend using `cscope` to make your job easier. Note that there is a `cscope` build target which will automatically update the `cscope` index every time you recompile (or you can also re-run it manually), so you just need to set up your editor to use the generated index file.

B.4 Debugging with `gdb`

If you have never used a debugger before, we highly recommend you learn to use one, either through a graphical interface like the one in many IDEs, or from the command line. This will be a development tool you will not want to live without once you learn to use it.

`gdb` is the most widely used debugger for C code, with support for kernel development using many virtual machines. There are many wonderful reasons to use `gdb` to debug Weenix in particular, so please read the appendix on debugging for more information.

Appendix C

Debugging

As you begin to develop your kernel, you will undoubtedly find problems in the code you have written. We have collected some techniques here which we found useful or which are largely undocumented elsewhere because they are fairly advanced or are specific to working on Weenix in particular.

C.1 Beginning Debugging

These are the simplest debugging techniques, which can be used at any stage of the project. They require the least amount of learning, but they also are not nearly as powerful as the debugging techniques listed later on.

C.1.1 Printing

By far the easiest technique for debugging is printing things out during the execution of your program. Of course, printing inside the kernel requires significant overhead (as you will learn when you write the TTY driver). Luckily, inside the support code we have provided printing methods which use the serial port of the virtual machine to print to a terminal outside of the computer. You can think of this as being like a printer connected to the serial port if you wish. The way to use this is through the `dbg` or `dbgq` macro.

```
#define dbg(dbgmodes, printfargs)
#define dbgq(dbgmodes, printfargs)
```

The difference between the two is that `dbg` additionally displays the file and line number information describing where the debug statement is located, but `dbgq` displays only the debugging message.

Debug messages are organized into many debug modes, each of which can be separately hidden or color-coded in the debug output. You can find a list of debug modes in `kernel/include/util/debug.h`. The string names next to each mode are used to configure which debug modes are displayed (the default

can be set in the Weenix configuration file and updated while Weenix is being debugged via the dbg command), and there is also a special string name which refers to all debug modes. For example:

```
dbg(DBG_THR, ("Creating a kernel thread "
              "with stack at 0x%p\n", stack));
```

In the snippet above, `DBG_THR` is the mode macro for the threading subsystem. Notice that the printfargs have an extra set of parentheses around them. This debug message would print something like:

```
kthread.c:123 kthread_stack_create(): Creating a kernel thread
with stack at 0x804800"
```

It is also possible to define a debugging message which is part of multiple modes by taking their bitwise or.

```
dbg(DBG_THR | DBG_VM, ("Creating a kernel thread with stack at "
                       "0x%p\n", stack));
```

C.1.2 Printing Using Info Functions

There are several info functions in the kernel which are provided exclusively for visualizing information in the debug console. To call one of these, use `dbginfo()`, like so:

```
dbginfo(DBG_VMMAP, vmmap_mapping_info, curproc->p_vmmap);
```

You can substitute in other functions besides `vmmap_mapping_info` - there are a bunch of functions ending with "`_info`" in Weenix which can be passed here.

C.1.3 Using Assertions

Another simple but widely-applicable approach is to intentionally crash your program when some condition is not met, which is known as "asserting" that condition's truth. This allows you to know what the failure point was, which is sometimes helpful for figuring out what caused the failure in the first place, but mostly it's helpful just for letting you know that there was a failure instead of failing silently.

We provide assert functionality with the `KASSERT()` macro, which tests the condition you pass it and prints out an error message on crash that tells you the condition that failed and where in the codebase the `KASSERT()` was. A trick you may find helpful is that any string will be evaluated to true in C (because it's just a pointer) so you can do things like this:

```
KASSERT(a == 1 && "A should be equal to 1");
```

This may be more helpful than not using a string, since the message will be printed when the assertion fails. Of course, you can also just put a comment next to the assertion in the code.

Because assertions are helpful for checking things that should always be true, we recommend placing them at the beginning of any function whose arguments must be of a certain form to ensure that the caller is checking and filtering all error cases correctly (this is especially helpful for syscalls).

C.2 Intermediate Debugging

Although the techniques above are very useful, it will quickly become apparent that a more robust way to debug the kernel is frequently necessary. Luckily, debuggers provide us with an excellent way to do this. `gdb` is the definitive debugger for C code on Unix systems, and the rest of this appendix will center on its usage. Most of the following information is applicable to all stages of Weenix development, except where noted.

C.2.1 Prerequisites

If you have never used `gdb` before, we recommend finding RMS's (Richard Stallman's) `gdb` Debugger Tutorial online and reading it. Before you read more advanced or Weenix-specific debugging tips, make sure you can use the following `gdb` features (most important ones listed first, more specialized ones later).

- Compiling with symbols.
- Breakpoints, stepping, continuing.
- Viewing and traversing the call stack.
- Inspecting variables.
- Inspecting the contents at a particular memory address.
- Using watchpoints.[1]
- Conditional breakpoints.[2]
- Inspecting the contents of registers.

Once you understand these basics, you should be able to debug pretty much any simple user-level program that you have the code for. However, there are a few more specific topics that are helpful in some cases.

[1] Watchpoints do not work with all the simulators which Weenix will run on.

[2] Conditional breakpoints do not work with all simulators which Weenix will run on. To get similar functionality, add an `if` statement to your code with the condition you care about and set a breakpoint inside there.

C.2.2 Debugging Multiple Threads

Although this does not relate directly to Weenix, you will frequently be debugging multithreaded programs. Debugging in the presence of multiple threads is usually the same as debugging single-threaded programs, but the place where it differs is in cases like deadlock. If you are debugging a program in `gdb` and it deadlocks, you can hit `Ctrl-C` to pause the program. Then, you can inspect the stack trace as usual. If you need to check the other threads (perhaps the first one you were given wasn't the one in a deadlock, or maybe you want to see what other threads are contributing to the deadlock) you can see a list of threads using "`info threads`" and can switch to thread `N` by typing "`thread N`" (substitute the real thread number for `N` in that command).

Another note about multiple threads (particularly if you might be canceling them) is that you must be careful what calls you make during critical sections of code. For instance, if you want to ensure that a call to cancel some thread doesn't take effect until after a certain section of code is complete, there is a set of standard library calls which make system calls which might act as a cancellation point. Look in the `man` page for `pthread_cancel()` for a list of these library calls. Note that `printf()` is one of them, so you might want to use `gdb` to debug instead of printing out messages to tell you when certain events happen.

C.2.3 Using the Weenix `gdb` Scripts

Because Weenix has been debugged using `gdb` for some time, there are a few Python scripts which run as custom commands inside the debugger which you can use to help debug your OS. These are all given under the command "`kernel`", and you can find more detailed information about the commands available by running "`help kernel`" inside of `gdb`, but here are a few of the highlights.

- To turn kernel memory checking on, add the line "`set kernel memcheck on`" to the beginning of `init.gdb`. This allows you to run `kernel page` and `kernel slab` at shutdown, which tell you how much memory has not been cleaned up. Note that turning memory checking on may slow Weenix down a little bit, and that a handful of memory segments simply cannot be freed (page tables and stacks are common culprits).
- You can access the debug info functions mentioned above through the `kernel info` command.
- `kernel list` will tell you all the elements of a linked list if you used the macro list implementation.
- `kernel proc` will give you a list of all live processes.

Note that these require a certain version of the Python language and `gdb` to run; an error will be printed at the beginning of your `gdb` sessions if your versions are not compatible.

C.2.4 Disassembling a Program

If you need to know what exact instructions are being run or where exactly in your code you are executing, you should probably disassemble your program. The easiest way to do this inside `gdb` is to use the `disas` command, although you can also use a command like `x/100i $eip` to print the next 100 instructions starting at the address pointed to by your instruction pointer. You can also use the `objdump` tool (separate from `gdb`) to disassemble the entire contents of an executable.

C.3 Advanced Debugging Techniques

Finally, there are some times when the approaches above are just not enough. These tips are for Weenix in particular, although they can easily be adapted for use in debugging many other kernels as well. These techniques will only be relevant when working on the virtual memory system of Weenix, since they are mainly for debugging problems in userland processes.

C.3.1 Debugging a Page Fault

The most obvious thing to do when debugging a page fault is to look at the address the fault is happening on. However, this doesn't tell you the context under which the page fault was generated, such as the stack or the section of code that was running. To find these, the easiest way is to inspect the stored context of the process that generated the pagefault. This will allow you to see the address of the instruction which was running (and perhaps more importantly, the name of its containing function), the stack pointer, and any other registers which might be needed to figure out what the processor was doing. This trick comes in especially handy when you also load in the symbol file for the user-level process, as the next section will show.

C.3.2 Debugging processes from the kernel debugger

It is a little bit like black magic, but you can actually debug user processes from inside the kernel debugger if you load in the symbol file for the user process into the correct location in memory. To find this information, you can use `objdump` (in your ordinary shell) to get some information about the user process.

```
$ objdump --headers --section=".text" user/simple/hello

user/simple/hello:  file format elf32-i386

Sections:
Idx Name          Size      VMA       LMA       File off  Algn
  6 .text         00000064  08048208  08048208  00000208  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
```

The relevant information that this gives us is the starting address of the text section, `0x08048208` (the `VMA` column). So, we can open up Weenix with `gdb` attached, and add in the relevant symbol file for our userland process.

```
Breakpoint 3, bootstrap (arg1=0, arg2=0x0) at main/kmain.c:121
121     {
(gdb) add-symbol-file user/simple/hello 0x08048208
add symbol table from file "user/simple/hello" at
        .text_addr = 0x8048208
(y or n) y
Reading symbols from user/simple/hello...done.
(gdb) b main
Breakpoint 4 at 0x8048208: file ./hello.c, line 12.
(gdb) c
Continuing.
...
Breakpoint 4, main (argc=1, argv=0x8047eec) at ./hello.c:12
12      {
(gdb) list
7
8       #include <unistd.h>
9       #include <fcntl.h>
10
11      int main(int argc, char **argv)
12      {
13              open("/dev/tty0", O_RDONLY, 0);
14              open("/dev/tty0", O_WRONLY, 0);
15
16              write(1, "Hello, world!\n", 14);
```

Now, you should be able to set breakpoints in the userland process.

It is worth noting, however, that this does not prevent you from also putting breakpoints in kernel code. `gdb` is intentionally dumb about how breakpoints work - whenever your instruction pointer reaches the specified address, `gdb` will pause, no matter what symbol files you've added - so since the text of your kernel and the user process are both loaded, you can place breakpoints in either one.

C.3.3 `gdb` and `DYNAMIC`

With dynamic linking enabled, it becomes a step more difficult to debug userland processes from the kernel debugger, but certainly not impossible. The main issue is that there is a bunch more code in the `ld-weenix` shared library that you won't be able to debug unless you tell `gdb` to load the symbol file. The best way we know of to do this is to print out the memory map of the process you're trying to debug and make an educated guess about which region might correspond to `ld-weenix`. (It should be a shared region with execute permissions.) Then,

load the debugging symbols starting at the address which corresponds to the beginning of that region.