Hacking GeekOS

David H. Hovemeyer, Jeffrey K. Hollingsworth, and Iulian Neamtiu

April 12, 2005

Copyright © 2003, 2004, 2005 David H. Hovemeyer, Jeffrey K. Hollingsworth, and Iulian Neamtiu

Legal Notice

This work is licensed under the Creative Commons Attribution-NonCommercial-ShareAlike License. To view a copy of this license, visit <http://creativecommons.org/licenses/by-nc-sa/1.0/> or send a letter to Creative Commons, 559 Nathan Abbott Way, Stanford, California 94305, USA.

Chapter 1

Introduction

GeekOS is an educational operating system kernel. GeekOS tries to combine realism and simplicity. It is a realistic system because it targets a real hardware platform - the x86 PC. It strives for simplicitly in that it contains the bare minimum functionality necessary to provide the services of a modern operating system, such as virtual memory, a filesystem, and interprocess communication.

IMPORTANT

This document and the GeekOS distribution are works in progress. At the time of writing (March 3, 2004) this document is about 80% complete, and the GeekOS code is about 95% complete. We will be filling in the remaining text and code in the near future. In the meantime, if you have any questions about this manual or about GeekOS itself, please send email to daveho@cs.umd.edu <mailto:daveho@cs.umd.edu>.

This document has two purposes. The first purpose is to give an overview of the GeekOS kernel, and to cover the topics needed to read, understand, and modify the kernel source code. The second purpose is to present a series of projects in which you can build important new functionality on top of the GeekOS kernel. These projects are suitable for use in a senior level undergraduate course, or for self-study.

1.1 Intended Audience

This document is for anyone interested in gaining hands-on experience in operating system kernel programming. Most operating system textbooks focus on high level theory and concepts. This document is intended to bridge the gap between those concepts and actual, working kernel code. We will try to give you all of the information and background you need to start hacking. In this way, this document complements your operating system textbook.

1.2 Required background

Before you start hacking on GeekOS, we assume that you have the following skills and knowledge:

- Basic understanding of what an operating system kernel does
- A strong understanding of the C programming language
- Experience programming at the system call level in an operating system such as Linux or Windows
- Experience programming using threads, such as pthreads or Java threads

- Some knowledge of computer architecture and organization
- Familiarity with assembly language for some CPU architecture, and a willingness to learn x86 (a.k.a. Intel IA32) assembly language

Chapter 2

Kernel Hacking 101

GeekOS is an operating system *kernel*. In many respects, it is simply a C program. It has functions, threads, a memory allocator, and so forth. However, unlike a C program executing as a *user mode* process of a host operating system such as Linux or Windows, a kernel operates in *kernel mode*. A program executing in kernel mode has total control over the computer's CPU, memory, and hardware devices.

Writing code to execute in kernel mode presents a few challenges that you will need to be aware of. This chapter presents some important tips and techniques that you will need to use as you modify the GeekOS kernel.

2.1 Kernel Mode Restrictions

The runtime environment of the GeekOS kernel has several important restrictions you will need to be aware of.

2.1.1 Limited Set of Library Functions

Because the operating system is that lowest level of software in the computer, all functionality used by the kernel must be implemented within the kernel. This is different than for user programs, which are generally linked against a set of standard libraries containing often-used functions.

The only standard C library functions available in GeekOS are a subset of the string functions (`strcpy()`, `memcpy()`, etc.) and the `snprintf()` function. Prototypes for these functions are defined in the header `<geekos/string.h>`.

In addition to the standard C functions, the GeekOS kernel also contains functions which are similar to C library functions. The `Print()` function (prototype in `<geekos/screen.h>`) is a subset of the standard C `printf()` function. The `Malloc()` and `Free()` functions (prototypes in `<geekos/malloc.h>`) are equivalent to the `malloc()` and `free()` functions.

2.1.2 Limited Stack

Each thread in the GeekOS kernel has a 4K stack. If a thread overflows its stack, a kernel crash will generally be the result. Therefore, you should be very careful to conserve stack space:

- Do not allocate large data structures on the stack. Use the kernel heap allocator (`Malloc()` and `Free()`) instead.
- Do not use recursion. Avoid deep call stacks.

2.1.3 Limited Memory Protection

When the kernel starts executing there is no memory protection; every memory access made by the kernel is to real (physical) memory. Dereferences of null pointers are not trapped. For this reason, kernel code is more vulnerable to stray pointer references than user code. You will need to check your code very carefully to make sure there are no memory errors, because they are hard to debug.

When you add virtual memory to GeekOS (Chapter 9, "Project 4: Virtual Memory"), the kernel will gain a greater degree of protection for memory errors, but you will still need to be careful.

2.1.4 Asynchronous Interrupts

Many hardware devices use *interrupts* to notify the CPU of important events: the expiration of a timer, the completion of an I/O request, etc. An interrupt is an immediate asynchronous transfer of control to an *interrupt handler*. When the handler completes, control returns to the point where execution was interrupted, and the original code resumes. Note that interrupt handlers may cause a *thread context switch*, meaning that other threads may execute before control returns to the interrupted thread.

The practical implications of interrupts are described in Section 3.2. You will want to read this section carefully.

2.2 Recommended Kernel Hacking Practices

If you observe the precautions described in this document, you will find that programming in kernel mode is fairly straightforward. However, to make your kernel hacking experience more pleasant, we strongly encourage you to adopt the following habits.

2.2.1 Use Assertions

The header `<geekos/kassert.h>` defines the `KASSERT()` macro, which takes a boolean expression. If the expression evaluates to false, the macro prints a message and halts the kernel. Here is an example:

```
void My_Function(struct Thread_Queue *queue)
{
    KASSERT(!Interrupts_Enabled()); /* Interrupts must be disabled */
    KASSERT(queue != 0);            /* queue must not be null */
    ...
}
```

As much as possible, you should rigorously use assertions to check function preconditions, postconditions, and data structure invariants. Assertions have two important benefits. First, a failed assertion immediately and precisely pinpoints a bug in the code, often before the kernel state becomes seriously corrupted. Second, assertions help document the code.

2.2.2 Use Print Statements

The `<geekos/screen.h>` header defines the `Print()` function, which supports much of the functionality of the standard C `printf()` function. Print statements are the most useful general technique for debugging kernel code. As with any debugging, you should adopt the strategy of forming a hypothesis and gathering evidence to support or refute the hypothesis.

2.2.3 Test Early and Often

To a much greater extent than for user programs, kernel code needs to be developed and tested in small pieces. Whenever you reach a stable point in your kernel development, you should commit your work to

version control. We strongly recommend that you use CVS <http://www.cvshome.org/> to store your code.

Chapter 3

Overview of GeekOS

This chapter presents a high level overview of the GeekOS kernel and the subsystems you will use when you add new functionality to GeekOS. Once you have read this chapter, you can refer to Chapter 12, "GeekOS API Reference" for detailed descriptions of functions in the GeekOS kernel.

3.1 Memory

The GeekOS kernel manages all memory in the system. Two types of memory can be allocated.

3.1.1 Page Allocator

All of the memory in the system is divided into chunks called *pages*. In the x86 architecture, the page size is 4K. A page is a unit of memory that can be part of a virtual address space; this characteristic of pages will come into play when you add virtual memory to GeekOS (Chapter 9, "Project 4: Virtual Memory"). For now, you can just think of pages as small fixed size chunks of memory. Pages are allocated and freed using the `Alloc_Page()` and `Free_Page()` functions in the `<geekos/mem.h>` header file.

3.1.2 Heap Allocator

The heap allocator provides allocation of arbitrary-sized chunks of memory. The `Malloc()` function allocates a chunk of memory, and the `Free()` function releases a chunk of memory. The prototypes for these functions are in the `<geekos/malloc.h>` header file.

3.2 Interrupts and Threads

Interrupts and *threads* are the mechanisms used by GeekOS to divide CPU resources between the various tasks the operating system performs. Understanding how interrupts and threads interact is crucial to being able to add new functionality to the kernel.

3.2.1 Interrupts

Interrupts are used to inform the CPU of the occurrence of an important event. The important characteristic of an interrupt is that it causes control to transfer immediately to an *interrupt handler*. Interrupt handlers are simply C functions.

There are several kinds of interrupts:

- *Exceptions* indicate that the currently executing thread performed an illegal action. They are a form of *synchronous* interrupt, because they happen in a predictable manner. Examples include execution of an invalid instruction and attempts to divide by zero. Exceptions generally kill the thread which raised them, because there is no way to recover from the exception.

- *Faults*, like exceptions, are also synchronous. Unlike exceptions, they are generally recoverable; the kernel can do some work to remove the condition that caused the fault, and then allow the faulting thread to continue executing. An example is a *page fault*, which indicates that a page containing a referenced memory location is not currently mapped into the address space. If the kernel can locate the page and map it back into the address space, the faulting thread can continue executing. You will learn more about page faults in Chapter 9, "Project 4: Virtual Memory".

- *Hardware interrupts* are used by external hardware devices to notify the CPU of an event. These interrupts are *asynchronous*, because they are unpredictable. In other words, an asynchronous interrupt can happen at any time. Sometimes the kernel is in a state where it cannot immediately handle an asynchronous interrupts. In this case, the kernel can temporarily *disable interrupts* until it is ready to handle them again. An example of a hardware interrupt is the *timer interrupt*.

- *Software interrupts* are used by *user mode* processes to signal that they require attention from the kernel. The only kind of software interrupt used in GeekOS is the *system call interrupt*, which is used by processes to request a service from the kernel. For example, system calls are used by processes to open files, perform input and output, spawn new processes, etc.

When an interrupt handler has completed, it returns control to the thead which was interrupted at the exact instruction where the interrupt occurred. For the most part, the original thread resumes as if the interrupt never happened.

The occurrence of an interrupt can cause a *thread context switch*. This fact has imporant consequences for code that modifies shared kernel data structures, and will be described in detail in Section 3.2.4.

3.2.2 Threads

Threads allow multiple tasks to share the CPU. In GeekOS, each thread is represented by a `Kernel_Thread` object, defined in `<geekos/kthread.h>`.

Threads are selected for execution by the *scheduler*. At any given time, a single thread is executing. This is the *current thread*, and a pointer to its `Kernel_Thread` object is available in the `g_currentThread` global variable. Threads that are ready to run, but not currently running, are placed in the *run queue*. Threads that are waiting for a specific event to occur are placed on a *wait queue*. Both the run queue and wait queues are instances of the `Thread_Queue` data structure, which is simply a linked list of `Kernel_Thread` objects.

Some threads execute entirely in kernel mode. These threads are called *system threads*. System threads are used to perform demand-driven tasks within the kernel. For example, a system thread called the *reaper thread* is used to free the resources of threads which have exited. The floppy and IDE disk drives each use a system thread to wait for I/O requests, perform them, and communicate the results back to the requesting threads.

In contrast to system threads, *processes* spend most of their time executing in *user mode*. Processes should be familiar to you already; when you run an ordinary program in an operating system like Linux or Windows, the system creates a process to execute the program. Each process consists of a *memory space* reserved for the exclusive use of the running program, as well as other resources like files and semaphores.

In GeekOS, a process is simply a `Kernel_Thread` which has a special data structure attached to it. This data structure is the `User_Context`. It contains all of the memory and other resources allocated to the process. Because processes are just ordinary threads which have the capability of executing in user mode, they are sometimes referred to in GeekOS as *user threads*.

Processes start out executing in user mode. However, interrupts occurring while the process is executing in user mode cause the process to switch back into kernel mode. When the interrupt handler returns, the process resumes executing in user mode.

3.2.3 Thread Synchronization

GeekOS provides a high level mechanism to synchronize threads: *mutexes* and *condition variables*. (The mutex and condition variable implementation in GeekOS is modeled on the pthreads API, so if you have some any pthreads programming, this section should seem very familiar.) Mutexes and condition variables are defined in the `<geekos/synch.h>` header file.

IMPORTANT

Mutexes and condition variables may only be used to synchronize threads. It is not legal to access a mutex or condition variable from an handler for an asynchronous interrupt.

Mutexes are used to guard *critical sections*. A mutex ensures MUTual EXclusion within a critical section guarded by the mutex; only one thread is allowed to hold a mutex at any given time. If a thread tries to acquire a mutex that is already held by another thread, it is suspended until the mutex is available.

Here is an example of a function that atomically adds a node to a list:

```
#include <geekos/synch.h>

struct Mutex lock;
struct Node_List nodeList;

void Add_Node(struct Node *node) {
    Mutex_Lock(&lock);
    Add_To_Back_Of_Node_List(&nodeList, node);
    Mutex_Unlock(&lock);
}
```

Condition variables represent a condition that threads can wait for. Each condition variable is associated with a mutex, which must be held while accessing the condition variable and when inspecting or modifying the program state associated with the condition.

Here is an elaboration of the earlier example that allows threads to wait for a node to become available in the node list:

```
#include <geekos/synch.h>

struct Mutex lock;
struct Condition nodeAvail;
struct Node_List nodeList;

void Add_Node(struct Node *node) {
    Mutex_Lock(&lock);
    Add_To_Back_Of_Node_List(&nodeList, node);
    Cond_Broadcast(&nodeAvail);
    Mutex_Unlock(&lock);
}

struct Node *Wait_For_Node(void) {
    struct Node *node;

    Mutex_Lock(&lock);
    while (Is_Node_List_Empty(&nodeList)) {
        /* Wait for another thread to call Add_Node() */
        Cond_Wait(&nodeAvail, &lock);
    }
    node = Remove_From_Front_Of_Node_List(&nodeList);
    Mutex_Unlock(&lock);
```

```
    return node;
}
```

3.2.4 Interactions between Interrupts and Threads

The GeekOS kernel is *preemptible*. This means that, in general, a *thread context switch* can occur at any time. Choosing which thread to execute at a preemption point is the job of the *scheduler*. In general, the scheduler will choose the task which has the highest *priority* and is ready to execute.

The main cause of asynchronous (involuntary) threads switches is the timer interrupts, which the kernel uses to ensure that no single thread can completely monopolize the CPU. However, other hardware interrupts (such as the floppy disk interrupt) can also cause asynchronous thread switches. Threads often need to modify data structures shared by other threads and/or interrupt handler functions. If a thread switch were to occur in the middle of an operation modifying a shared data structure, the data structure could be left in an inconsistent state, leading to a kernel crash or other unpredictable behavior.

Fortunately, it is easy to temporarily disable preemption by *disabling interrupts*. This is done by calling the `Disable_Interrupts()` function (prototype in `<geekos/int.h>`). After this function is called, the processor ignores all external hardware interrupts. While interrupts are disabled, the current thread is guaranteed to retain control of the CPU: no other threads or interrupt handlers will execute. When the thread is ready to re-enable preemption, it can call `Enable_Interrupts()`.

There are a variety of situations when interrupts should be disabled. Generally, disabling interrupts can be used to make any sequence of instructions *atomic*; this means that the entire sequence of instructions is guaranteed to complete as a unit, without interruption.

The most important specific situation when interrupts should be disabled is when a scheduler data structure is modified. A typical example is putting the current thread on a *wait queue*. Here is an example:

```
/* Wait for an event */
Disable_Interrupts();
while (!eventHasOccurred) {
    Wait(&waitQueue);
}
Enable_Interrupts();
```

In this example, a thread is waiting for the occurrence of an asynchronous event. Until the event occurs, it will suspend itself by waiting on a wait queue. When the event occurs, the interrupt handler for the event will set the `eventHasOccurred` flag and move the thread from the wait queue to the run queue.

Consider what could happen if interrupts were *not* disabled in the example above. First, the interrupt handler for the event could occur between the time `eventHasOccurred` is checked and when the calling thread puts itself on the wait queue. This might result in the thread waiting forever, even though the event it is waiting for has already occurred. A second possibility is that a thread switch might occur while the thread is adding itself to the wait queue. The handler for the interrupt causing the thread switch will place the current thread on the run queue, *while the wait queue is in a modified (inconsistent) state*. Any code accessing the wait queue (such as an interrupt handler or another thread) might cause the system to crash.

Fortunately, (almost) all functions in GeekOS that need to be called with interrupts disabled contain an assertion that will inform you immediately if they are called with interupts enabled. You will see many places in the code that look like this:

```
KASSERT(!Interrupts_Enabled());
```

These statements indicate that interrupts must be disabled in order for the code immediately following to execute. Knowing when interrupts need to be disabled is a bit tricky at first, but you will soon get the hang of it.

One final caveat: regions of code which explicitly disable interrupts cannot be nested. Another way to put this is that it is illegal to call `Disable_Interrupts()` when interrupts are already disabled, and it is illegal to call `Enabled_Interrupts()` when interrupts are already enabled. Consider the following code:

```
void f(void) {
    Disable_Interrupts();
    g();
    ... modify shared data structures ...
    Enable_Interrupts();
}

void g(void) {
    Disable_Interrupts();
    ...
    Enable_Interrupts();
}
```

This code, if allowed to execute, would contain a bug. If the function `g()` returned with interrupts enabled, a context switch in function `f()` could interrupt a modification to a shared data structure, leaving it in an inconsistent state. Fortunately, there is a simple way to write code that will selectively disable interrupts if needed:

```
bool iflag;

iflag = Begin_Int_Atomic();
... interrupts are disabled ...
End_Int_Atomic(iflag);
```

Sections of code that use `Begin_Int_Atomic()` and `End_Int_Atomic()` may be nested safely.

3.3 Devices

The GeekOS kernel contains *device drivers* for several important hardware devices.

3.3.1 Text Screen

The text screen provides support for displaying text. The screen driver in GeekOS emulates a subset of VT100 and ANSI escape codes for cursor movement and setting character attributes. Text screen services are provided in the `<geekos/screen.h>` header file.

The main function you will use in sending output to the screen is the `Print()` function, which supports a subset of the functionality of the standard C `printf()` function. Low level screen output is provided by the `Put_Char()` and `Put_Buf()` functions, which write a single character and a sequence of characters to the screen, respectively.

3.3.2 Keyboard

The keyboard device driver provides a high level interface to the keyboard. It installs an interrupt handler for keyboard events, and translates the low level key scan codes to higher level codes that contain ASCII character codes for pressed keys, as well as modifier information (whether shift, control, and/or alt are pressed). The header for for the keyboard services is `<geekos/keyboard.h>`.

Threads can wait for a key event by calling the `Wait_For_Key()` function. Key codes are returned using the `Keycode` datatype, which is a 16 bit unsigned integer. The low 10 bits of the keycode indicate which physical key was pressed or released. Several flag bits are used:

- `KEY_SPECIAL_FLAG` is set for keys that do not have an ASCII representation. For examples, function keys and cursor keys call into this category. If this flag is *not* set, then the low 8 bits of the key code contain the ASCII code.
- `KEY_KEYPAD_FLAG` is set for keys on the numeric keypad.
- `KEY_SHIFT_FLAG` is set if one of the shift keys is currently pressed. For alphabetic keys, this will cause the ASCII code to be upper case.
- `KEY_CTRL_FLAG` and `KEY_ALT_FLAG` are set to indicate that the control and alt keys are pressed, respectively.
- `KEY_RELEASE_FLAG` is set for key release events. You will probably want to ignore key events that have this flag set.

3.3.3 System Timer

The system timer is used to provide a periodic timeslice interrupt. After the current thread has been interrupted a set number of times by the timer interrupt, the scheduler is invoked to choose a new thread. You will generally not need to use any timer services directly. However, it is worth noting that the system timer is the mechanism used to ensure that all threads have a chance to execute by preventing any single thread from monopolizing the CPU.

3.3.4 Block Devices: Floppy and IDE Disks

Block devices are the abstraction used to represent storage devices: i.e., disks. They are called block devices because they are organized as a sequence of fixed size blocks called *sectors*. Block device services are defined in the `<geekos/blockdev.h>` header file.

Although real block devices often have varying sector sizes, GeekOS makes the simplifying assumption that all block devices have a fixed sector size of 512 bytes. This is the value defined by the `SECTOR_SIZE` macro.

GeekOS supports two kinds of block devices: *floppy drives* and *IDE disk drives*. A naming scheme is used to identify particular drives attached to the system:

- `fd0`: the first floppy drive
- `ide0`: the first IDE disk drive
- `ide1`: the second IDE disk drive

A particular instance of a block device is represented in the kernel by the `Block_Device` data structure. You can retrieve a block device object by passing its name to the `Open_Block_Device()` function. Once you have the block device object, you can read and write particular sectors using the `Block_Read()` and `Block_Write()` functions.

Block devices are used as the underlying storage for *filesystems*. A filesystem builds higher level abstractions (files and directories) on top of the raw block storage offered by block devices. In Chapter 10, "Project 5: A Filesystem", you will implement a filesystem using an underlying block device.

Chapter 4

Overview of the Projects

This chapter gives a brief overview of the projects in which you will add important new functionality to the GeekOS kernel. It also discusses all of the requirements for compiling and running GeekOS.

4.1 Project Descriptions

There are a total of seven projects. For the most part, each project builds on the one before it, so you will need to do them in order. The projects were originally developed as part of a senior level undergraduate operating systems course, so the complete sequence will require a significant amount of effort to complete. Some projects will be more difficult than others; in particular, the virtual memory project (Chapter 9, "Project 4: Virtual Memory") and the filesystem project (Chapter 10, "Project 5: A Filesystem") require you to write a fairly large amount of code (several hundred to a thousand lines of code for each project). The good news is that when you complete the last project, GeekOS will be a functional operating system, capable of running multiple process with full memory protection.

Project 0 (Chapter 5, "Project 0: Getting Started") serves as an introduction to modifying, building, and running GeekOS. You will add a kernel thread to read keys from the keyboard and echo them to the screen.

For Project 1 (Chapter 6, "Project 1: Loading Executable Files"), you become familiar with the structure of an executable file. You are provided with code for loading and running executable files, but you need to first become familiar with the ELF file format, then write code to parse the provided file and pass it to the loader.

In Project 2 (Chapter 7, "Project 2: Adding Processes"), you will add support for *user mode processes*. Rather than using virtual memory to provide separate user address spaces, this project uses *segmentation*, which is simpler to understand.

Project 3 (Chapter 8, "Project 3: Scheduling") improves the GeekOS scheduler and adds support for semaphores to coordinate multiple processes.

Project 4 (Chapter 9, "Project 4: Virtual Memory") replaces the segmentation based user memory protection added in Project 1 with paged virtual memory. Pages of memory can be stored on disk in order to free up RAM when the demand for memory exceeds the amount available.

Project 5 (Chapter 10, "Project 5: A Filesystem") adds a hierarchical read/write filesystem to GeekOS.

Project 6 (Chapter 11, "Project 6: ACLs and Inter-process Communication") adds access control lists (ACLs) to the filesystem, and adds interprocess communication using anonymous half-duplex pipes. Upon the completion of this project, GeekOS will resemble a very simple version of Unix.

4.2 Required Software

Compiling GeekOS requires a number of tools. The good news is that if you are running on a Linux/x86 or FreeBSD/x86 system, or if you are running Cygwin <`http://cygwin.com/`> on Windows, you probably have most or all of the software you need to compile GeekOS already installed. If you don't, it is generally not too difficult to obtain the required software.

The following is a complete list of software needed to compile GeekOS.

- gcc <http://gcc.gnu.org/>, version 2.95.2 or later, targeting any i386/ELF platform, or targeting PECOFF under Cygwin
- Any ANSI-compliant C compiler for the host platform (the computer you are planning to compile GeekOS on); unless you are cross-compiling, this can be the same compiler you use to compile GeekOS
- GNU binutils <http://www.gnu.org/software/binutils/>, targeting any i386/ELF platform, or PECOFF under Cygwin; you probably already have this if you have gcc
- GNU Make <http://www.gnu.org/software/make/>; the GeekOS makefile will not work with other versions of make
- Perl <http://www.perl.org/>, version 5 or later
- egrep <http://www.gnu.org/software/grep/grep.html>
- AWK <http://www.gnu.org/software/gawk/>
- diff3 <http://www.gnu.org/software/diffutils/>
- NASM <http://nasm.sourceforge.net/>; on Linux or FreeBSD systems, this is probably the only required software not already installed

4.3 Setting Up the GeekOS Distribution

Before you get started, you should choose a directory in which to install the GeekOS distribution.

NOTE

This document assumes that you are using a Unix-like system such as Linux, FreeBSD, or MacOS/X, and that you are using the bash <http://www.gnu.org/software/bash/> shell, or a similar shell such as the Bourne (/bin/sh) or Korn (/bin/ksh) shells.

A GeekOS distribution is a file whose name looks like `geekos-`*`version`*`.tar.gz`. For example, let's say you are using GeekOS version 0.2.0, that you have downloaded the GeekOS distribution to the directory */home/fred*, and that you want to install the distribution in the directory */home/fred/software*. In this case, you would install the source distribution using the following commands:

```
$ cd /home/fred/software
$ gunzip -c /home/fred/geekos-0.2.0.tar.gz | tar xvf -
$ GEEKOS_HOME=/home/fred/software/geekos-0.2.0
$ export GEEKOS_HOME
$ PATH=$GEEKOS_HOME/bin:$PATH
$ export PATH
```

4.4 Starting a Project

Once you have installed the GeekOS distribution and all of the software required to compile it, you are ready to extract Project 0. Let's say you have chosen the directory */home/fred/work* to contain the working directory for your project. Then you would run the following command:

```
$ cd /home/fred/work
$ startProject project0
```

This will create a directory */home/fred/work/project0* containing the GeekOS source code, ready for you to start hacking.

IMPORTANT

It is an extremely good idea to use CVS <http://www.cvshome.org/> or a similar version control system to store your project files. Each time you reach a stable point in your project, commit the source files. That way, you will always be able to revert to a stable version if something goes wrong with your code.

4.5 Continuing to a New Project

Each project contains a slightly different version of the base GeekOS system. For example, the buffer cache system is introduced in the fourth project because it is required by the filesystem. Also, code is sometimes removed because it is no longer needed.

To make it easy for you to move your code from one project to the next, the `startProject` script can automatically merge your code from the previous project into the code for the new project. For example, let's say you have completed Project 0 and are ready to move onto Project 1. You would run the following commands:

```
$ cd /home/fred/work
$ startProject project1 project0
```

Although the process of incorporating your old code into the new project is largely automatic, you will sometimes see messages like the following:

```
Warning: Conflicts detected in merge of file src/geekos/main.c
```

When you look at the file in the new project directory that was reported to contain a merge conflict, you will see something like the following:

```
<<<<<<< Your version of src/geekos/main.c
=======
static void Mount_Filesystems(void);
static void Spawn_Init_Process(void);
>>>>>>> Master version of src/geekos/main.c from project1
```

In this case, two function prototypes were added to the source file `src/geekos/main.c`.

In general, you should resolve conflicts by simply deleting the conflict markers (the lines reading "<<<<<<<", "=======", and ">>>>>>>"). This will have the effect of including the code from both your previous project and the new code from the master source for the new project. Sometimes you will need to comment out code from your previous project that will not be needed in the new project.

When you have removed all of the conflict markers, you can start working on getting the project to compile. Once the code compiles, you can start working on adding the new features that the project requires.

4.6 Compiling a Project

Once you have used the `startProject` script to create a working directory for a project, and you have installed all of the software required to compile GeekOS, it should be a simple matter to compile the project. Let's say you are going to compile Project 1. Here are the commands you would use:

```
$ cd /home/fred/work/project1/build
$ make depend
$ make
```

4.7 Running GeekOS Using Bochs

Although GeekOS targets a real hardware platform, it is intended to be run in the Bochs <http://bochs.sourceforge.net/> PC emulator. Bochs runs as an ordinary user process in your host operating system, and is available for most common operating systems. We recommend that you use Bochs version 2.0 or later; as of the time of this writing, the latest version of Bochs is 2.1.1, which works well with GeekOS.

Once you have installed Bochs, you will need to edit the file `build/.bochsrc` in the working directory for your project. In this file, you will see two entries that look like the following:

```
# You will need to edit these lines to reflect your system.
vgaromimage: /software/bochs-2.0.2/share/bochs/VGABIOS-lgpl-latest
romimage: file=/software/bochs-2.0.2/share/bochs/BIOS-bochs-latest, address=0xf0000
```

You should change the lines beginning `vgaromimage` and `romimage` so that they reflect the directory in which you installed Bochs. For example, if you installed Bochs in the directory `/usr/local/bochs-2.1`, you should replace the occurrences of `/software` with `/usr/local/bochs-2.1`.

After editing the `.bochsrc` file, you are ready to start bochs. Add the directory containing the `bochs` executable to your `PATH` environment, and then invoke the `bochs` command from within the `build` directory of the project. You be prompted with a set of options. Choose `Begin simulation`, or simply hit the **Enter** or **Return** key to choose the default.

If GeekOS has been compiled successfully, and Bochs is configured properly, a window should appear emulating the VGA text screen. You are now running GeekOS!

4.8 Troubleshooting

If GeekOS does not run correctly, look at the file `bochs.out` produced in the `build` directory of the project. If errors prevented Bochs from running, or if GeekOS crashed, this file will usually contain diagnostic information which identifies the problem.

4.9 Project Hints

Every project contains placeholders indicating where you need to add code. These placeholders use the `TODO` macro. Here is an example from Project 0:

```
TODO("Start a kernel thread to echo pressed keys");
```

These placeholders are often accompanied by a comment giving you ideas about how to approach the new functionality that needs to be added.

Chapter 5

Project 0: Getting Started

The purpose of this project is to introduce you to working with the GeekOS source code and running GeekOS in the Bochs emulator.

5.1 The Assignment

Add code to the kernel to create a new kernel mode thread. The kernel mode thread should print out "Hello from xxx" where xxx is your name. It should then call the keyboard input routine `Wait_For_Key()` repeatly (and echo each character entered) until the termination character (control-d) is entered.

5.2 Hints

- This project will not require much code to be written. However, it may take some time to get things set up and debugged.
- Execution of the GeekOS kernel begins in the `Main()` function in the source file `src/geekos/main.c`.
- To start a new kernel mode thread you use the `Start_Kernel_Thread()` function. Look at the comments before the definition of this function in `src/geekos/kthread.c`.

Chapter 6

Project 1: Loading Executable Files

6.1 Introduction

In this project you will write code to parse an executable file in ELF format and pass the result of the parsing to a program laoder we provide.

6.2 Required Reading

This project will require you to understand the ELF executable format.

You will need to read the ELF Executable Format <http://www.x86.org/ftp/manuals/tools/elf.pdf> documentation. In this project, you will need to be able to parse the ELF *program headers* in order to find out how to load an executable file's text (code) and data into process memory.

6.3 Project Synopsis

This project will require you to change the `src/geekos/elf.c`. Your only task is to implement the `Parse_ELF_Executable()` function. This involves reading the program headers of an ELF executable to find the file offset, length, and user address for the executable's text and data segments. Based on this information, you should fill in the `Exe_Format` data structure passed as a parameter. This data structure will be used by the loader to determine how to load the executable. The loader is already implemented for you.

6.4 Loading the Executable

GeekOS, like other operating systems, uses files to store executable programs. These files are in ELF format. We have provided you with a simple read-only filesystem for this project, plus a test file that you will load. The test file is `src/user/a.c`, and after compilation and building, GeekOS will see it as `/c/a.exe`. When GeekOS boots up, it reads `/c/a.exe` into memory, calls your parsing code from `Parse_ELF_Executable()` and starts a kernel-mode thread that will run the `a.exe` code.

Loading ELF executables is fairly straightforward. You will need to locate the ELF program headers. These headers will describe the executable's *text* and *data* segments. [1] As you parse the ELF executable, you will fill in the fields of an `Exe_Format` data structure, which is a high level representation of how to load data from the executable file into memory.

[1] Note: in the context of the ELF format, "segment" simply refers to a region of the executable file that is loaded into memory. It has nothing to do with segments used by the CPU.

6.5 Testing Your Project

If you've done everthying correctly, when you start Bochs you should see this output:

```
Hi ! This is the first string
Hi ! This is the second string
Hi ! This is the third (and last) string
If you see this you're happy
```

Chapter 7

Project 2: Adding Processes

7.1 Introduction

In this project you will extend the GeekOS operating system to include user mode processes and system calls.

7.2 Required Reading

This project will make extensive use of systems programming features of the x86 (a.k.a. IA32) CPU architecture. It will also require you to understand the ELF executable format.

The Intel IA32 Architecture Manuals are the definitive reference for programming x86 CPUs. You can find them at the Intel website <http://www.intel.com/>; the order numbers for the set are 253665 (Volume 1, Basic Architecture), 253666 and 253667 (Volumes 2A and 2B, Instruction Set Reference), and 253668 (Volume 3, System Programming Guide). For this project, you will need to understand *segments* and *local descriptor tables*. These are described in Volume 3. You will want to read Volume 3, Chapter 3 carefully to understand how segments work.

You might need to revise some information in ELF Executable Format <http://www.x86.org/ftp/manuals/tools/elf.pdf> documentation, since you'll reuse ELF parsing from Chapter 6, "Project 1: Loading Executable Files".

7.3 Project Synopsis

This project will require you to make changes to several files. In general, look for the calls to the `TODO()` macro. These are places where you will need to add code, and they will generally contain a comment giving you some hints on how to proceed.

In `src/geekos/user.c`, you will implement the functions `Spawn()`, which starts a new user process, and `Switch_To_User_Context()`, which is called by the scheduler before executing a thread in order to switch user address spaces if required.

In `src/geekos/elf.c`, you will implement the `Parse_ELF_Executable()` function. This involves reading the program headers of an ELF executable to find the file offset, length, and user address for the executable's text and data segments. Based on this information, you should fill in the `Exe_Format` data structure passed as a parameter. This data structure will be used by the `Spawn()` function to determine how to load the executable.

In `src/geekos/userseg.c`, you will implement functions which provide support for the high level operations in `src/geekos/user.c`. `Destroy_User_Context()` frees the memory resources used by a user process. `Load_User_Program()` builds the `User_Context` structure for a new process by loading parts of the executable program into memory. The `Copy_From_User()` and `Copy_To_User()` functions copy data between the user address space and the kernel address space. The `Switch_To_Address_Space()` function activates a user address space by loading the processor's LDT register with the LDT of a process.

In `src/geekos/kthread.c`, you will implement functions which take a completed `User_Context` data structure and create a thread which is ready to execute in user mode. The `Setup_User_Thread()` function sets up the initial kernel stack for the process, which specifies the initial contents of the processor registers when the process enters user mode for the first time. `Start_User_Thread()` is a higher level operation which takes a `User_Context` object and uses it to start the new process.

7.4 Using Segmentation

To implement memory protection for user processes, this project uses *segmentation*. With segmentation, each process resides in a single physically contiguous chunk of memory. You can allocate this chunk of memory using the `Malloc()` function.

In order to provide a private address space, you will need to create *segments* for code and data of the process. Both segments will refer to the single memory chunk you have allocated for the process. The segments will reside in the *local descriptor table* (LDT) that you will create for the process. In the LDT, you will define *segment descriptors* that define the process code and data segments. Both of these should be created at the user privilege level: this will have the effect of that the process will only be able to access the memory in the segments you define in the LDT.

Here is a general list of steps you will need to follow in order to set up the segments and LDT for a process:

1. Create an LDT descriptor by calling the routine `Allocate_Segment_Descriptor()`
2. Create an LDT selector by calling the routine `Selector()`
3. Create a text segment descriptor by calling `Init_Code_Segment_Descriptor()`
4. Create a data segment descriptor by calling `Init_Data_Segment_Descriptor()`
5. Create an data segment selector by calling the routine `Selector()`
6. Create an text segment selector by calling the routine `Selector()`

7.5 Loading the Executable

GeekOS, like other operating systems, uses files to store executable programs. These files are in ELF format. We have provided you with a simple read-only filesystem for this project, which contains several programs that you can use to test user mode. To simplify the process of loading an executable, you will simply load the executable into a single buffer in the kernel. To do this, you can use the function `Read_Fully()`, whose prototype is in `<geekos/vfs.h>`. Here is an example of how you would read the executable `/c/shell.exe` into a kernel buffer:

```
int rc;
void *buf;
ulong_t fileLen;

rc = Read_Fully("/c/shell.exe", &buf, &fileLen);
if (rc == 0) {
    /*
     * Read the file successfully!
     * buf points to a Malloc'ed buffer containing the file data,
     * and fileLen contains the length of the file.
     */
    ...
}
```

Loading ELF executables is fairly straightforward. You will need to locate the ELF program headers. These headers will describe the executable's *text* and *data* segments. [1] As you parse the ELF executable, you will fill in the fields of an `Exe_Format` data structure, which is a high level representation of how to load data from the executable file into memory.

7.6 Setting up the Process Memory

Besides loading the executable's text and data segments into memory, you will also need to create two other data structures in the process's memory space.

You will need to create a *stack* for the new process. The stack will reside at the top of the user address space. You can use the `DEFAULT_USER_STACK_SIZE` macro in `src/geekos/userseg.c` as the stack size.

You will also need to create an *argument block* data structure. This data structure creates the `argc` and `argv` arguments that are passed to the `main()` function of the user process. Two functions are defined to help you build the argument block (prototypes in `<geekos/argblock.h>`). `Get_Argument_Block_Size()` takes the *command string* passed to the `Spawn()` function and determines how many command line arguments there will be, and how many bytes are required to store the argument block. The `Format_Argument_Block()` function takes the number of arguments and argument block size from `Get_Argument_Block_Size()` and builds the argument block data structure in the memory location you have allocated for it.

Note that you will need to take the size of the stack and argument block into account when you decide how much memory to allocate for the process.

7.7 User Thread Creation

As described in Section 7.3, you will need to create a thread for the new process. You will need to set up the stack of your new thread to look like it has just been interrupted. This will require pushing the appropriate values onto the stack to match what the hardware would push for an interrupt. The routine `Push()` (`src/geekos/kthread.c`) can be used to push individual values onto the stack. The required values should be pushed onto the stack as follows:

- Stack Data selector (data selector)
- Stack Pointer (end of data memory)
- Eflags (IF bit should be set so interrupts are enabled)
- Text selector (text selector)
- Program Counter (code entry point address)
- Error Code (0)
- Interrupt Number (0)
- General Purpose Registers (esi should contain address of argument block)
- DS register (data selector)
- ES register (data selector)
- FS register (data selector)
- GS register (data selector)

[1] Note: in the context of the ELF format, "segment" simply refers to a region of the executable file that is loaded into memory. It has nothing to do with segments used by the CPU.

7.8 Adding System Calls

System calls are software interrupts made by processes in order to request services from the kernel. In GeekOS, the kernel handlers for system calls are implemented in the file `src/geekos/syscall.c`. Each system call you must implement is described in this file.

Some of the system calls require you to copy data between the kernel address space and the current process's user address space. This task is carried out by the `Copy_From_User()` and `Copy_To_User()` functions whose prototypes are in the header file `include/geekos/user.h`, and whose implementations are in the file `src/geekos/userseg.c`. You will need to implement these functions. With segmentation, this is a fairly easy task; once you have verified that the user buffer is valid (i.e., it lies entirely within memory belonging to the process), a call to `memcpy()` can be used to do the transfer.

7.9 Spawning the Init Process

When GeekOS boots up, it should start the user mode process in the file `/c/shell.exe`. This process is the ancestor of all other user mode processes in the operating system. You will need to add the code to start this process and wait for it to exit. See the function `Spawn_Init_Process()` in `src/geekos/main.c`.

7.10 Testing Your Project

We have provided several user mode programs with which you can test your project. `shell.exe` is a simple command interpreter program. It is available on a PFAT filesystem from within GeekOS at the path `/c/shell.exe`. It should be the first user program loaded by your kernel; for this reason, it is referred to as the *init process*. Once you have successfully implemented `Spawn()` and its support routines, and implemented all of the required system call handlers, GeekOS should boot into a familiar command prompt.

Once the init process loads, you can use it to test some other small user mode programs. `/c/b.exe` prints a message, echoes its command line arguments, and exits. `/c/c.exe` executes an illegal system call. It will be killed by the kernel.

Chapter 8

Project 3: Scheduling

8.1 Introduction

The purpose of this project is to explore scheduling algorithms and learn about inter-process synchronization via semaphores.

8.2 Multilevel Feedback Scheduling

There are many scheduling algorithms. In this project, you will augment the existing GeekOS Round-Robin scheduling algorithm with a multilevel feedback scheduler. In Round-Robin, all threads (really their `Kernel_Thread` structures) sit in a FIFO queue. In a multi-level feedback scheduler, you will use 4 queues instead of 1. Each queue is assigned a priority level. The queues will be numbered 0 through 3, with 0 being the highest priority, and 3 being the lowest. This will require changing `s_runQueue` (in `src/geekos/kthread.c`) from being a struct to being an array of structs; one for each priority level.

A newly created process's `Kernel_Thread` structure will be placed on the ready queue of highest priority (i.e., 0). If the process runs for the full quantum, then it will be placed on the next lowest priority (1, if the process was new). Each time a process completes a full quantum, it will be placed on the ready queue with the next lowest priority until it is at priority 3, at which point it can not go any lower. Hence, CPU intensive processes will be eventually placed on the lowest priority queue. If the process is blocked, the priority level will increase by one level until after blocking three quanta in a row it will be back to priority 0. To schedule a new `Kernel_Thread` to run, look at the head of the highest priority queue. If there is a `Kernel_Thread` there, place it on the run queue. If not, go to the next lowest priority queue, and keep repeating until you find a `Kernel_Thread`. Scheduling always attempts to look at the highest priority queue and work down. This may mean low priority processes are starved.

The choice of which scheduler to use should be made within the function `Get_Next_Runnable()`. Any function that calls the `Get_Next_Runnable()` should be unaware which scheduling algorithm is being used (i.e., do not pass the scheduling type as an argument). It should only be aware that some `Kernel_Thread` has been selected.

You will need to handle the case of the Idle thread specially. It should be placed in the lowest level of scheduling priority and should never be permitted to move out of that level.

Your operating system should be able to switch which scheduling algorithm is being used via a system call. The system call int To make the scheduling policy configurable, you should implement the `Sys_SetSchedulingPolicy()` system call in `src/geekos/syscall.c`. This system call will take two parameters, *`policy`* (passed in `state->ebx`) and *`quantum`* (passed in `state->ecx`). If the value of policy is 0, the system should switch to round robin scheduling, if the policy is 1, the system should switch to multilevel feedback. Other values of this parameter should result in an error code being returned (i.e. a non-zero return value). The value of the quantum parameter should be the number of ticks that a user process may run before getting removed from the processor. To implement the tunable quantum, you should change the constant `MAX_TICKS` in timer.c to be a global variable (whose default value is `MAX_TICKS`) that is set by this system call.

8.3 Semaphores

You will add the following system calls to your kernel (all defined in `src/geekos/syscall.c`):

- `Sys_CreateSemaphore()`
- `Sys_P()`
- `Sys_V()`
- `Sys_DestroySemaphore()`

`Sys_CreateSemaphore()` creates a semaphore. The user address and length of the string containing the semaphore name are stored in `state->ebx` and `state->ecx`, respectively. It will get back a semaphore ID, an integer between 0 and N-1. You should be able to handle at least 20 semaphores whose names may be up to 25 characters long. If there are no semaphores left (i.e., there were 20 semaphores with unique names already given), a negative number can be returned indicating an error. If the name corresponds to a semaphore that already exists, your function should return its ID. Otherwise, it should create a new semaphore, using the value in `state->edx` as the initial count value for the semaphore. In either case, you should add semaphore id (SID) to the list of semaphores the current process can use, as well increment the count of registered users which are permitted to use the semaphore.

The `P()` and `V()` operations acquire and release the semaphore, respectively. `P()` waits until the semaphore's count is greater than 0, decrements the count by 1, and then proceeds. `V()` increments the semaphore count by 1, and should wake up a thread that is waiting to acquire the semaphore, if any. When `Sys_P()` and `Sys_V()` are called, the kernel will check if the process has permission to make this call. It will do so by checking if the process has the SID in its list of SIDs that it can access (which is why you needed to create such a list). If it is there, it will be allowed to execute `P()` or `V()`. If not, the kernel should return back a negative value.

`Sys_DestroySemaphore()` will delete the passed semaphore. It will keep track of how many processes have references to this semaphore, and delete the semaphore from the table when the last process that can access this semaphore calls `Destroy_Semaphore()`. Note that you should ensure that when processes exit, they release their references to any semaphores they have access to, even if they don't explicitly call `Destroy_Semaphore()`.

8.4 Timing

One way to compare scheduling algorithms is to see how long it takes a process to complete from the time of creation to the termination of the process. You will investigate these differences by implementing a system call, `Sys_GetTimeOfDay()`.

`Sys_GetTimeOfDay()` will return the value of the kernel global variable `g_numTicks`. The variable is already implemented in the kernel, so you only need to implement the system call to read it. You can use this system call to determine how much time has elapsed between two events. You can do this by calling `Get_Time_Of_Day()` once at the beginning of the process (in the user code) and once at the end. You can calculate how long the process took to run, as well as when the process first got scheduled (based on ticks). Notice that there is no attempt to remove time spent by other processes. For example, if your process context switches out, then runs a second process, the second process's time during the context switch will be included in the first process's total time. This is known as "wall clock" time. One can also just calculate the time used by the process itself. This time is called process time (or sometimes virtual time). GeekOS currently calculates this time, but you do not need to use this information in this project.

8.5 Evaluating the Scheduling Policies

You should run several tests on the supplied application `workload.exe`, varying the quantum length as well as the two scheduling algorithms. At minimum try running the system with the inputs of:

```
/c/workload.exe rr 1
/c/workload.exe rr 100
/c/workload.exe mlf 1
/c/workload.exe mlf 100
```

You should investigate and be able to explain why the results occurred. The exercise is meant to let you consider the effects of quantum length and scheduling algorithms on the run of several processes.

Chapter 9

Project 4: Virtual Memory

9.1 Introduction

The purpose of this project is to add paging to the GeekOS kernel. This will require many small, but difficult changes to your project. More than any previous project, it will be important to implement one thing, test it and then move to the next one.

9.2 Changing the Project to Use Page Tables

The first step is to modify your project to use page tables and segmentation rather than just segments to provide memory protection. To enable using page tables, every region of memory your access (both kernel and data segment) must have an entry in a page table. The way this will work is that there will be a single page table for all kernel only threads, and a page table for each user process. In addition, the page tables for user mode processes will also contain entries to address the kernel mode memory. The memory layout for this is shown in Figure 9.1.

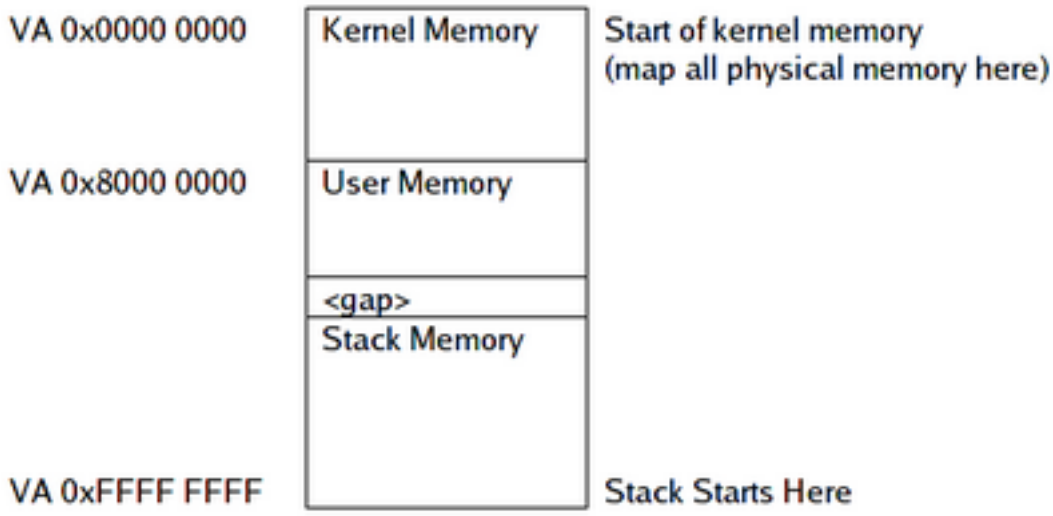

Figure 9.1: Virtual memory layout in GeekOS.

The kernel memory should be a one to one mapping of all of the physical memory in the processor (this limits the physical memory of the processor to 2GB, but this is not a critical limit for this project). The page table entries for this memory should be marked so that this memory is only accessible from kernel mode

(i.e. the `userMode` bit in the page directory and page table should be 0). To make this change, you should start by creating a page directory and page table entries for the kernel threads by writing a function that initializes the page tables and enables paging mode in the processor. You will do this in the `Init_VM()` function in `src/geekos/paging.c`.

To set up page tables, you will need to allocate a page directory (using `Alloc_Page()`) and then allocate page tables for the entire region that will be mapped into this memory context. You will need to fill out the appropriate fields in the page tables and page directories, which are represented by the `pte_t` and `pde_t` datatypes defined in `<geekos/paging.h>`. Finally, to enable paging for the first time, you will need to call an assembly routine, `Enable_Paging()`, that will take the base address of your page directory as a parameter and then load the passed page directory address into register `cr3`, and then set the paging bit in `cr0` (the MSB, bit 31).

The next step is to modify your user processes to all use pages in the user region. This is a two step process. First, you need to allocate a page directory for this user space. You should copy all of the mappings from the kernel mode page directory for those memory regions in the low range of memory. Next you need to allocate page table entries for the user processes text and data regions. Do not allocate extra space for the stack here. Finally, you should allocate space for one page of stack memory at the end of the virtual address range (i.e. the last entry in the last page table). For the user space page mappings, make sure to enable the `userMode` bits in both the page directory and page table entries.

You will also need to change some aspects of how the code from Project 1 sets things up. You should change the code and data segments for user processes so that the base address is be 0x80000000, and the limit is 0x80000000. This will allow the user space process to think that its virtual location 0 is the 2GB point in the page layout and will greatly simplify your kernel compared to traditional paged systems. You will also need to add code to `Switch_To_Address_Space()` to switch the PTBR register (`cr3`) as part of a context switch; where you load the LDT of the user context, you should also load the address of the page directory for the process, which is the `pageDir` field in the `User_Context` structure. [1]

When you are allocating pages of memory to use as part of a user address space, you should use a new function, `Alloc_Pageable_Page()` (prototype in `<geekos/mem.h>`. The primary difference is that any page allocated by this routine should have a special flag `PAGE_PAGEABLE` set in the flags field of its entry in the corresponding `Page` data structure. Having this flag set marks the page as being eligible to be stolen and paged out to disk by the kernel when a page of memory is needed elsewhere, but no free pages are available. Note that you should *not* allocate page tables or page directories using this function.

9.3 Handling Page Faults

One of the key features of using paging is to have the operating system handle page faults. To do this you will need to write a page fault interrupt handler. The first thing the page fault handler will need to do is to determine the address of the page fault; you can find out this address by calling the `Get_Page_Fault_Address()` function (prototype in `<geekos/paging.h>`. Also, the `errorCode` field of the `Interrupt_State` data structure passed to the page fault interrupt handler contains information about the faulting access. This information is defined in the `faultcode_t` data type defined in `<geekos/paging.h>`. Once the fault address and fault code have been obtained, the page fault handler will need to determine an appropriate action to take. Possible reasons for a page fault, and the action to take are shown in Figure 9.2.

9.4 Paging Out Pages

At some point, your operating system will run out of page frames to assign to processes. In this case, you will need to pick a page to evict from memory and write it to the backing store (paging file). You should

[1] You may choose to allocate the user code and data segment descriptors in the GDT (Global Descriptor Table) rather than having a separate LDT for each process. If you decide to use this approach, then the `User_Context` will not need to contain an LDT or selector fields. Instead, you should define user mode code and data segments in the GDT using the `Allocate_Segment_Descriptor()` function, *before any user process is created*, and create selectors for these segments to use for the code and data segment registers for each newly created process. Each user process can use the same user code and data segments; because each process uses a separate virtual address space, there is no way that a process can access another process's memory.

Cause	Indication	Action
Stack growing to new page	Fault is within one page of the current stack limit	Allocate a new page and continue.
Fault for paged out page	Bits in page table indicate page is on disk	Read page from paging device (sector indicated in PTE) and continue.
Fault for invalid address	None of the other conditions apply	Terminate user process

Figure 9.2: Actions to be taken when a page fault occurs.

implement a version of pseudo-LRU. Use the reference bit in the page tables to keep track of how frequently pages are accessed. To do this, add a `clock` field to the `Page` structure in `<geekos/mem.h>`. You should update the clock on every page fault.

You will also need to manage the use of the paging file. The paging file consists of a group of consecutive 512 bytes disk blocks. Calling the routine `Get_Paging_Device()` (prototype in `<geekos/vfs.h>`) will return a `Paging_Device()` object; this consists of the block device the paging file is on, the start sector (disk block number), and the number of sectors (disk blocks) in the paging file. Each page will consume 8 consecutive disk blocks. To read/write the paging device, use the functions `Block_Read()` and `Block_Write()`.

9.5 Page Ins

When a page is paged out to disk, the kernel stores the index returned by `Find_Space_On_Paging_File()` in the `pageBaseAddr` field of the page table entry (`pte_t`), and also stores the value `KINFO_PAGE_ON_DISK` in the entry's `kernelInfo` field. In your page fault handler, when you find a non-present page that is marked as being on disk, you can use the value stored in `pageBaseAddr` to find the data for the page in the paging file.

9.6 Copying Data Between Kernel and User Memory

Because the GeekOS kernel is preemptible and user memory pages can be stolen at any time, some subtle issues arise when copying data between the kernel and user memory spaces. Specifically, the kernel must *never* read or write data on a user memory page if that page has the `PAGE_PAGEABLE` bit set at any time that a thread switch could occur. The reason is simple; if a thread switch did occur, another process could run and steal the page. When control returns to the original thread, it would be reading or writing the wrong data, causing serious memory corruption.

There are two general approaches to dealing with this problem. One is that interrupts (and thus preemption) should be disabled while touching user memory. This approach is not a complete solution, because it is not legal to do I/O (i.e., `Block_Read()` and `Block_Write()`) while interrupts are disabled.

The second approach is to use *page locking*. Before touching a user memory page, the kernel will atomically clear the `PAGE_PAGEABLE` flag for the page; this is referred to as *locking* the page. Once a page is locked, the kernel can then freely modify the page, safe in the knowledge that the page will not be stolen by another process. When it is done reading or writing the page, it can *unlock* the page by clearing the `PAGE_PAGEABLE` flag. Note that page flags should only be modified while interrupts are disabled.

9.7 Implementation

In order to implementing virtual memory and paging, you will need to implement several functions.

9.7.1 Functions in `src/geekos/paging.c`

- `Init_VM()` (defined in) will set up the initial kernel page directory and page tables, and install a page fault handler function.
- `Init_Paging()` (defined in `src/geekos/paging.c`) should initialize any data structures you need to manage the paging file. As mentioned earlier, the `Get_Paging_Device()` function specifies what device the paging file is located on, and the range of disk blocks it occupies.
- `Find_Space_On_Paging_File()` should find a free page-sized chunk of disk space in the paging file. It should return an index identifying the chunk, or -1 if no space is available in the paging file.
- `Free_Space_On_Paging_File()` will free a chunk of space in the paging file previously allocated by `Find_Space_On_Paging_File()`.
- `Write_To_Paging_File()` writes the data stored in a page of memory to the paging file.
- `Read_From_Paging_File()` reads the data for a page stored in the paging file into memory.

9.7.2 Functions in `src/geekos/uservm.c`

- `Destroy_User_Context()` frees all of the memory and other resources (semaphores, files) used by a process.
- `Load_User_Program()` loads an executable file into memory, creating a complete, ready-to-execute user address space.
- `Copy_From_User()` copies data from a user buffer into a kernel buffer.
- `Copy_To_User()` copies data from a kernel buffer into a user buffer.
- `Switch_To_Address_Space()` switches to a user address space by loading its page directory and (if necessary) its LDT.

9.8 Extra Credit

The implementation of virtual memory in GeekOS is a very simple one. There are many ways that it can be extended and improved.

9.8.1 Improving `Find_Page_To_Page_Out()`

When a page of pageable memory is required and no pages are available, the kernel uses the `Find_Page_To_Page_Out()` function (in `src/geekos/mem.c`) to select a page to page out. While interrupts are disabled (meaning no other threads or interrupt handlers can run), this function traverses the array of all `Page` data structures, in order to find the one with the oldest clock field.

How can you make this function work more efficiently?

9.8.2 Adding a User Heap

Currently, the GeekOS kernel only creates a code segment, data segment, and stack for user processes. However, it does not create a user heap, meaning that user processes cannot allocate memory dynamically (using a function like `malloc()`).

In order to add support for user heaps to GeekOS, you will need to do several things:

- The kernel will need to inform the `_Entry()` function in the C library (`src/libc/entry.c`) of the start address and maximum size of the heap area. You may also wish to allow user processes to request that the heap be extended (similar to the `brk()` system call in Unix and Linux). In any case, `_Entry()` will need to initialize the heap area.
- The C library will need to implement functions to allocate and free memory. You can copy the file `src/geekos/bget.c` into the C library, and use the functions `bpool()`, `bget()`, and `brel()` to initialize the heap, allocate memory, and free memory, respectively. To add new source files to the C library, put them in the `src/libc` directory and add them to the definition of the `LIBC_C_SRCS` macro in `build/Makefile`.
- The page fault handler will need to dynamically allocate pages for memory accesses that occur in the heap area, in much the same way as it allows the stack to grow dynamically.

Chapter 10

Project 5: A Filesystem

10.1 Introduction

The purpose of this project is to add a new filesystem to GeekOS.

10.2 GOSFS: GeekOS FileSystem

The main part of this project is to develop a new filesystem for the GeekOS. This filesystem will reside on the second IDE disk drive in the Bochs emulator. This will allow you to continue to use your existing pfat drive to load user programs while you test your filesystem.

GOSFS will provide a filesystem that includes multiple directories, access control (via user ids), and long file name support. The access control will be added as part of the next project.

10.3 The Virtual Filesystem Layer (VFS)

In this project, you will work extensively with the virtual filesystem layer, commonly referred to as the VFS. This part of the kernel allows multiple filesystem implementations to coexist. A high level view of the VFS, as well as the C library and kernel subsystems it communicates with, is shown in Figure 10.1.

The VFS layer works by dispatching requests for filesystem operations (such as opening, reading, or writing a file) to the appropriate filesystem implementation. The VFS works by defining several abstract datatypes representing mounted filesystem instances, open files, and open directories. Each of these datatypes contains a *virtual function table* which contains pointers to functions in the filesystem that the object belongs to. For example, `File` objects created by the GOSFS filesystem have a virtual function table whose `Read()` entry points to the function `GOSFS_Read()`. [1]

To give you an idea of how VFS works, here is what happens when a user process reads data from an open file in a GOSFS filesystem.

1. The process calls the `Read()` in the C library, which generates a software interrupts to notify the kernel that a system call is requested. It passes a file descriptor identifying the open file, the address of the buffer to store the data read from the file, and the number of bytes requested.

2. The kernel calls the `Sys_Read()` system call handler function. This function will look at the process's open file list (in `User_Context`) to find the `File` object representing the open file. It will also need to allocate a kernel buffer to temporarily hold the data read from the file. It will then call the kernel VFS function `Read()`.

3. The VFS `Read()` will find the address of the `Read` function in the file's virtual function table. Because the file is part of a GOSFS filesystem, this will resolve to the `GOSFS_Read()` function.

[1] If you are familiar with virtual function tables in C++ and Java, this is exactly the same idea.

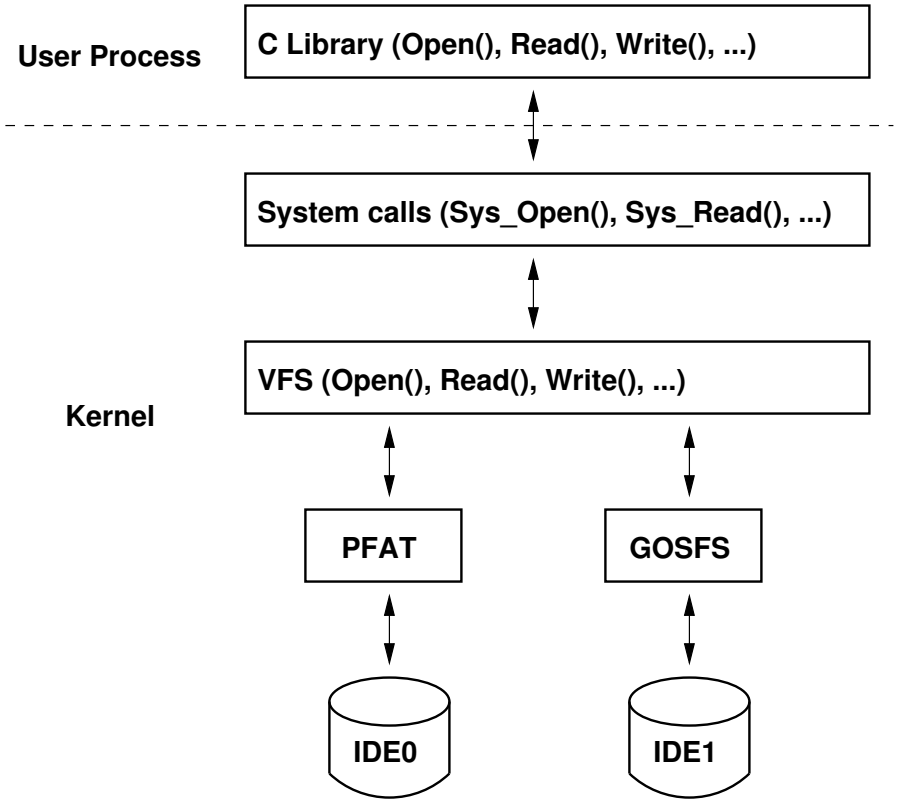

Figure 10.1: Overview of the Virtual Filesystem (VFS)

4. `GOSFS_Read()` will do whatever work is necessary to read the requested data from the file, at the current file position, copying the data into the kernel buffer created by `Sys_Read()`.

5. When control returns to `Sys_Read()`, it will copy the data read from the kernel buffer to the user buffer (using the `Copy_To_User()` function), and destroy the kernel buffer.

6. Finally, `Sys_Read()` will return, and the process will resume execution.

10.4 VFS Data Types and Operations

This section describes the high level VFS data types and operations, which are defined in `<geekos/vfs.h>`. You should read and understand the definitions in this header, and you may also want to look at the VFS function implementations in `src/geekos/vfs.c`.

Every operation described in this section is implemented by function in `src/geekos/gosfs.c`. For example, the `Read()` file operation is implemented by the function `GOSFS_Read()`.

10.4.1 Filesystem

The `Filesystem` data structure represents a filesystem type. There are two operations associated with `Filesystem`:

- The `Format()` operation formats a block device. Formatting writes filesystem metadata to the device so that it can be mounted.

- The `Mount()` operation mounts a block device containing on a particular path prefix in the overall filesystem namespace. It must check to see that the required filesystem metadata is present, and then initialize any internal data structures needed by the filesystem in order to serve requests such as opening files, reading and writing file data, etc.

10.4.2 Mount_Point

The `Mount_Point` data structure represents a mounted filesystem instance. In GeekOS, filesystems are mounted on a *path prefix*, which indicates what part of the overall filesystem namespace the mounted filesystem will be part of. For example, the PFAT filesystem containing the user executables is usually mounted on the `/c` prefix. Each mounted filesystem uses a block device as its underlying storage.

`Mount_Point` objects support several operations:

- The `Open()` operation opens a file in the mounted filesystem.
- The `Create_Directory()` operation creates a directory in the mounted filesystem.
- The `Open_Directory()` operation opens a directory in the mounted filesystem, so that its entries can be read using the `Read_Entry()` operation.
- The `Stat()` retrieves the files metadata, such as size and file permissions, for a named file in the mounted filesystem.
- The `Sync()` flushes all file data and filesystem metadata stored in memory but not written to the disk. This operation is needed because filesystems ususally keep data in memory buffers, and write modified data out to the disk later. Following a `Sync()` operation, all data in the filesystem is guaranteed to be written to the disk.

For all `Mount_Point` operations that take paths, the path prefix used to mount the filesystem is removed before the operation is called. So, if a GOSFS filesystem is mounted on prefix "`/d`", and the file "`/d/stuff/foo.txt`" is passed to the high level VFS function `Open()`, the path passed to `GOSFS_Open()` will be "`/stuff/foo.txt`".

10.4.3 File

The `File` data type represents a file or directory opened by a process. `File` objects are created by the `Open()` and `Open_Directory()` `Mount_Point` operations.

Regular `File` objects (i.e., those representing files, not directories) maintain a current *file position*. All reads and writes are performed relative to the current file position.

`File` objects support the following operations:

- The `FStat()` operation, like `Mount_Point`'s `Stat()` operation, retrieves file metadata, such as size and file permissions.
- The `Read()` operation reads data at the current file position. Directories do not support this operation.
- The `Write()` operation writes data at the current file position. Directories do not support this operation.
- The `Seek()` operation changes the current file position. Directories do not support this operation.
- The `Close()` operation closes the file or directory.
- The `Read_Entry()` operation reads the next directory entry from an open directory. This operation is not supported for regular files.

10.5 Requirements

Each user space process will have an open file table that keeps track of which files that process can currently read and write. Any user process should be able to have 10 files open at once: this constant is defined as the `USER_MAX_FILES` macro in `<geekos/user.h>`.

The header file `<geekos/gosfs.h>` specifies constants and data structures for you to use in your filesystem implementation. All disk allocations will be in units of 4KB (i.e. 8 physical disk blocks); this value is specified by the `GOSFS_FS_BLOCK_SIZE` macro. You should maintain a free disk block list via a bit vector. A library of bitset functions is provided (prototypes in `<geekos/bitset.h>`) that will allow you to set and clear bits in blocks of memory, and also to find free bits representing free filesystem blocks. Your filesystem should support long filenames up to 127 characters, as specified by the `GOSFS_FILENAME_MAX` constant. The total file name (i.e. a full path) will be no more than 1023 characters (as specified by the `VFS_MAX_PATH_LEN` in the header `<geekos/fileio.h>`).

You will use a version of indexed allocation to represent the blocks of your filesystem. The first 8 4KB blocks are direct blocks, the next 1024 blocks are indirect blocks. These values are specified by the `GOSFS_NUM_DIRECT_BLOCKS` and `GOSFS_NUM_INDIRECT_BLOCKS` macros, respectively. The filesystem should provide a way to implement the next 1 million blocks as double indirect blocks, but you are not required to implement that interface.

10.6 GOSFS_Dir_Entry

The `GOSFS_Dir_Entry` data structure represents a single directory entry as it is stored on disk. It contains several fields:

- The `size` field stores the size of the file or directory referred to by the entry.
- The `flags` field stores several boolean flags. The `GOSFS_DIRENTRY_USED` flag indicates that the directory entry is used. If this is not set, it means the directory entry is available. The `GOSFS_DIRENTRY_ISDIRECTORY` flag indicates that the directory entry refers to a subdirectory. The `GOSFS_DIRENTRY_SETUID` flag means that the directory entry refers to a file which is a *setuid executable*: if the file is executed, the new process created will have the same user id as the file's owner. (You will not need to do anything with this bit until the next project.)
- The `filename` field stores the filename, including room for a nul terminator character.
- The `blockList` field stores the block numbers of the direct, indirect, and doubly indirect blocks representing the allocated storage for the file or directory.
- Finally, the `acl` field stores the list of Access Control List entries for the file or directory. The first entry specifies the entry for the file's owner.

10.7 The Buffer Cache

In your filesystem implementation, you will frequently need to read data from the filesystem into memory, and sometimes you will need to write data from memory back to the filesystem. Most operating system kernels use a *buffer cache* for this purpose. The idea is that the buffer cache contains memory buffers which correspond to particular filesystem blocks. To read a block, the kernel requests a buffer containing that block from the buffer cache. To write a block, the data in the buffer is modified, and the buffer is marked as *dirty*. At some point in the future, the data in the dirty buffer will be written back to the disk.

For this project, we have provided you will a buffer cache implementation, which you can use by including the `<geekos/bufcache.h>` header file. The `Buffer_Cache` data structure represents a buffer cache for a particular filesystem instance. You can create a new buffer cache by passing the block device to the `Create_FS_Buffer_Cache()` function. A buffer containing the data for a single filesystem block is represented by the `FS_Buffer` data type. You can request a buffer for a particular block by calling the

`Get_FS_Buffer()` function. The actual memory containing the data for the block is pointed to by the `data` field of `FS_Buffer`. If you modify the data in a buffer, you must call the `Modify_FS_Buffer()` function to let the buffer cache know that the buffer is now dirty. When your code has finished accessing or modifying a buffer, it should be released using the `Release_FS_Buffer` function.

Buffers are accessed in a *transactional* manner. Only one thread can use a buffer at a time. If one thread is using a buffer and a second requests the same buffer (by calling `Get_FS_Buffer()`), the second thread will be suspended until the first thread releases the buffer (by calling `Release_FS_Buffer()`). Note that you need to be careful never to call `Get_FS_Buffer()` twice in a row, since that will cause a self-deadlock.

The GeekOS buffer cache writes dirty buffers to disk lazily. If you want to immediately write the contents of a buffer back to the disk, you can call the `Sync_FS_Buffer()` function. It is generally a good idea to write back modified filesystem buffers when they contain *filesystem metadata* such as directories or disk allocation bitmaps; by writing these blocks eagerly, the filesystem is less likely to be seriously corrupted if the operating system crashes or if the computer is turned off unexpectedly. You can flush all of the dirty buffers in a buffer cache by calling the `Sync_FS_Buffer_Cache()`; this is useful for implementing the `Sync()` operation for your mounted filesystems.

The `Destroy_FS_Buffer_Cache()` function destroys a buffer cache, first flushing all dirty buffers to disk. This may be useful in your implementation of `GOSFS_Format()`.

10.8 Getting Started

Implementing a filesystem is complex. This section offers some suggestions on how to approach the project.

First, implement the `GOSFS_Format()` function. You may want to create a temporary `FS_Buffer_Cache` object to use to perform I/O.

Next, implement the `GOSFS_Mount()` function. This function is passed a `Mount_Point` object with a pointer to the block device containing the filesystem image. You will probably want to create an auxiliary data structure to store in the mount point; you can use this data structure to store a pointer to your buffer cache object, and any other information you will need when performing `Mount_Point` operations such as opening a file. You can store a pointer to the auxiliary data structure in the `fsData` field of the `Mount_Point` object.

You can test formatting and mounting a GOSFS filesystem by running the following commands from the GeekOS shell prompt:

```
$ format ide1 gosfs
$ mount ide1 /d gosfs
```

Once you can format and mount a GOSFS filesystem, you can start implementing `Mount_Point` functions. `GOSFS_Open()` is a logical place to start. When this function is called with the `O_CREATE` bit set in the mode flags, you should create a new file. As with `GOSFS_Mount()`, you will probably want to store an auxiliary data structure in the `File` object when it is created; you can use the `fsData` field for this purpose. You can use the `touch.exe` program to test file creation. You will also need to implement `GOSFS_Create_Directory()`, which you can test using the `mkdir.exe` program.

Once files and directories can be created, you can start working on `File` operations. A good place to start would be the `GOSFS_Close()` function. Eventually you will need to implement more complex functions like `GOSFS_Read()`, `GOSFS_Read_Entry()`, and `GOSFS_Write()`.

10.9 Issues

This section discusses some implementation issues you should think about as you work on the project.

10.9.1 Concurrency

Mount points, files, and directories can be accessed by multiple processes. Therefore, you will need to establish a locking discipline for your filesystem data structures to ensure that they can be safely accessed by multiple threads and processes. For this project, you can use a simple approach, such as having a single mutex protect an entire filesystem instance.

10.9.2 File Sharing

The `File` data structure defined in `<geekos/vfs.h>` represents a file or directory that has been opened by a process. If two processes open the same file at the same time, they will have separate `File` objects. This is necessary because both processes must have separate file positions; if one process performs a read or a seek operation, it should not affect the other process. However, both objects refer to the same file in the filesystem; changes made by one process should be seen by the other. This means that you will probably want to arrange for both VFS `File` objects to have a pointer to a common, shared data structure representing the "real" file. This arrangement is shown in Figure 10.2.

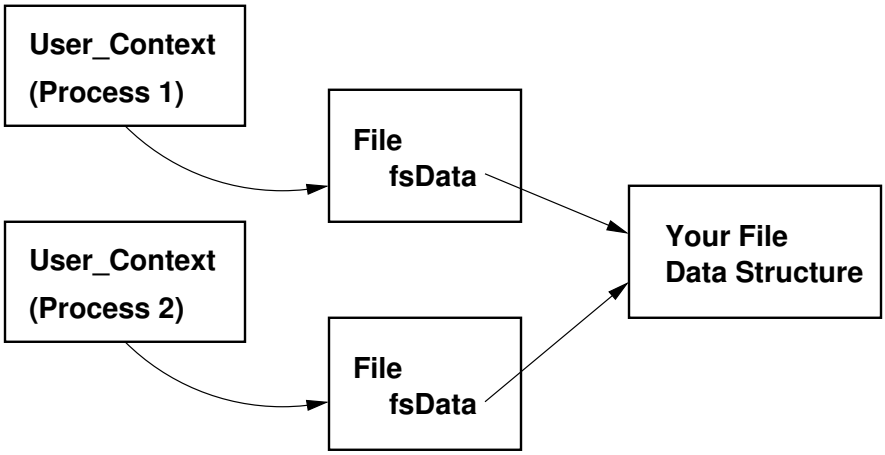

Figure 10.2: Multiple `File` objects can refer to a common data structure

Because there can be multiple references to your internal file data structure, you will probably want to keep a reference count, so you can clean up properly when there are no more references.

Chapter 11

Project 6: ACLs and Inter-process Communication

IMPORTANT

This chapter has not yet been written. It will be finished in the near future. Please send email to daveho@cs.umd.edu <mailto:daveho@cs.umd.edu> for more information.

Chapter 12

GeekOS API Reference

This chapter documents commonly used functions in the GeekOS kernel. Note that this chapter is likely to be incomplete. You can always refer directly to the GeekOS source code if you need more information on a particular function.

12.1 Thread functions

This section describes functions used to manage threads.

12.1.1 Start_Kernel_Thread()

struct Kernel_Thread * **Start_Kernel_Thread** (*startFunc*, *arg*, *priority*, *detached*);
Thread_Start_Func *startFunc*;
ulong_t *arg*;
int *priority*;
bool *detached*;

Starts a new kernel thread. *startFunc* is a function which will be called as the body of the new thread. The start function will execute with interrupts enabled. It should return void and take an unsigned long parameter. *arg* is an arbitrary value passed to the thread's start function. It can be used to convey extra information, or a pointer to a data structure to be used by the thread. *priority* is the priority of the new thread. `PRIORITY_NORMAL` should be used for ordinary kernel threads. `PRIORITY_USER` should be used for user processes. *detached* indicates whether the parent thread will wait for the child thread to exit. If true, then the parent must call the `Join()` function to wait for the child. If false, then the parent must not call `Join()`.

12.1.2 Exit()

void **Exit** (*exitCode*);
int *exitCode*;

Causes the current thread to exit with the exit code given by *exitCode*. If the current thread has a parent and is not a detached thread, the exit code will be returned by the `Join()` call made by the parent. Interrupts must be enabled.

12.1.3 Join()

int **Join** (*kthread*);
struct Kernel_Thread **kthread*;

Wait for a child thread *kthread* to exit. The child must have been created with its *detached* parameter set to true. Once the child has exited, returns the exit code of the child thread. Interrupts must be enabled.

12.1.4 Wait()

void **Wait** (*waitQueue*);
struct Thread_Queue **waitQueue*;

Puts the current thread on given wait queue. The thread will resume executing at a later point when another thread or interrupt handler calls `Wake_Up()` or `Wake_Up_One()` on the same wait queue. Interrupts must be disabled.

12.1.5 Wake_Up()

void **Wake_Up** (*waitQueue*);
struct Thread_Queue **waitQueue*;

Wakes up all threads on given wait queue by moving them to the run queue. Interrupts must be disabled.

12.1.6 Wake_Up_One()

void **Wake_Up_One** (*waitQueue*);
struct Thread_Queue **waitQueue*;

Wakes up the highest priority thread in given wait queue by moving it to the run queue. Interrupts must be disabled.