GeekOS: An Instructional Operating System for Real Hardware

David Hovemeyer
Department of Computer Science
University of Maryland
daveho@cs.umd.edu

December 10, 2001

Introduction

This paper describes GeekOS, a tiny operating system kernel which runs on real hardware (x86-based PCs). GeekOS has been used as the basis for projects in an undergraduate operating systems class at the University of Maryland. This paper discusses the motivation for the creation of GeekOS, describes its design and implementation, and relates experiences using it as an instructional operating system.

Most educators teaching operating systems use projects to help students understand the issues and concepts. There are two basic approaches to developing operating system projects:

1. execute on bare hardware, directly exposing the hardware devices
2. execute within the user mode of a 'host' operating system, simulating the hardware-level details

In this paper, I will present opinions arguing in favor of executing on bare hardware, and (hopefully) back them up with real experience gained from GeekOS. One of the most important factors in making the real hardware approach feasible is the availability of a high quality PC hardware emulator[Boc01]. In this paper I hope to present evidence that emulators offer many of the benefits of simulation while preserving the benefits of coding for the bare metal.

Because its intended audience is educators and students, GeekOS is free software, and may be downloaded from the GeekOS web site[Gee01]. This paper describes version 0.0.1a, released on November 27, 2001.

The outline of the paper is as follows. First, I will go into more detail about the motivation for GeekOS. Next, I will describe the design and implementation of GeekOS. I will then discuss experience gained from using GeekOS as an instructional operating system. Finally, I'll describe ideas for future work, and present some conclusions.

1 Motivation

This section describes the motivations for creating GeekOS. In particular, it explains why I started a new instructional operating system from scratch.

The first motivation was the desire to create a template for booting a program on a bare x86 PC. Booting a program written a high-level language, even something as simple as 'Hello world', is an important first step in writing a new OS kernel. By putting together a simple, well-documented example of how a kernel is bootstrapped, I thought I might assist other aspiring kernel hackers. Due to the simplicity of GeekOS, it might also be useful for embedded systems developers.

The second motivation was the desire to update the projects for the undergraduate operating system course at the University of Maryland. The projects were originally developed in MS-DOS, using a 16 bit real mode compiler and assembler. While hosting the projects from within MS-DOS allows direct access to hardware and memory (a desirable property), it also leaves the host operating system vulnerable to being crashed by the projects. Also, programming in MS-DOS exposes the dreaded Intel segment/offset address model. Finally, students accustomed to editing and compiling from in a 'real' operating system such as Unix or Windows NT are unlikely to find working within MS-DOS to be a pleasant experience.

GeekOS is a logical replacement for the original 16 bit projects because it remains true to their spirit of programming close to the bare metal. Because x86 PCs are cheap[1] and readily available, and because high quality free development tools exist for them, they are a natural choice for the host platform. In addition, an excellent free PC hardware emulator, Bochs[Boc01], allows OS development to be done within a 'virtual' PC run as an ordinary user process under a variety of popular operating systems. The availability of a high quality emulator was a deciding factor in the viability of writing an instructional OS for real hardware, since it offers many of the advantages of simulation (debugging, isolation, etc.) while preserving all of the benefits of coding for real hardware. The main disadvantage of emulation is that it requires a fast CPU to get acceptable emulated performance. With PCs being both extremely fast and cheap, this is not really an issue. A lab of low-end PCs[2] running Windows or Linux would be more than adequate to allow students to develop their projects using GeekOS and Bochs. In addition, because Bochs is an emulator, it runs on non-x86 CPUs. Therefore, Unix workstations with suitable cross-development tools are also a possibility.

2 Alternative Instructional OS Kernels

In this section I'll explain why I found existing instructional operating systems unsatisfactory.

2.1 Nachos

One of the most widely-used instructional operating systems, Nachos [CPA93], runs as a user-level process under Unix, and executes user programs via an instruction-level simulator

[1] I recently purchased a 90 MHz Pentium with 16 MB of RAM and a 500 MB hard drive for $13.

[2] My main development system for GeekOS is a 333 MHz Intel Celeron running Linux.

for a MIPS processor. The operating system code is written in C++ (which is compiled and executed for the host Unix system), and the OS kernel executes 'beside' the CPU simulator. The main advantage of this approach is that the operating system may be debugged using the host system's debugger, and a crash of the operating system only affects a single user process. Another advantage to this approach is that it is friendly to timesharing; many students may run their projects on a single computer. Finally, the operating system is insulated from low-level hardware details; it may treat the system call interface of the host operating system as its 'hardware'.

I find the Nachos approach to be unsatisfactory for two reasons. The main reason, which is partly aesthetic, is that Nachos destroys one of the elegant properties that, in my opinion, makes operating systems so fascinating. In a 'real' operating system, the operating system runs on the same CPU and hardware as the user programs. In Nachos, it does not, a fact which may cause real confusion when students try to understand how a real operating system separates the kernel from user processes. Another (lesser) objection is that Nachos abstracts away chip and device level programming; this could be seen as an advantage, but I believe that operating systems are a nice way to introduce students to the low-level organization of real hardware. Finally, because of the unusual architecture of Nachos, it would be difficult to port to run on bare hardware (where the OS kernel and user processes run on the same CPU).

2.2 Minix

Minix[Min01, TW97] is another widely-used instructional operating system. Unlike Nachos, it runs on real hardware. However, the goals of Minix are quite different from those of GeekOS. Minix is a complete operating system which includes virtual memory, a filesystem, device drivers, TCP/IP networking, and a full suite of user-mode programs. As such, it comprises many tens of thousands of lines of source code. While well written and extensively documented, it is nonetheless large and complex. This creates two problems from the standpoint of teaching about operating systems. First, it is somewhat intimidating to dive into such a large source base. Second, Minix already implements all important OS features, making it somewhat difficult to design significant student projects. GeekOS, being at most a rudimentary OS implementation, suffers neither of these problems.

3 Design and Implementation

This section discusses my design goals for GeekOS, and describes the current design and implementation in some detail.

3.1 Design Goals

Three main goals influenced the design of GeekOS: simplicity, realism, and understandability.

As an instructional operating system kernel, keeping GeekOS simple was of paramount importance. To this end, I tried to limit its features to those that I considered most fundamental. The following list summarizes GeekOS's features:

- interrupt handling
- heap memory allocator
- time-sliced kernel threads with static-priority scheduling
- mutexes and condition variables for synchronization of kernel threads
- user mode with segmentation-based memory protection and a simple system call interface
- device drivers for keyboard and VGA text mode display

Absent from this list are paged virtual memory, storage device drivers, and filesystem. I used the x86's segmentation mechanism to implement memory protection for user mode tasks. Note that the user mode segments use a flat 32 bit memory model, avoiding the contortions required by real mode segment/offset addressing. To work around the lack of disk storage or filesystem, GeekOS includes a mechanism for compiling user programs as data objects linked directly into the kernel. This technique could also be used to implement a RAM-based filesystem.

3.2 Low-level Details

This section explores the design and implementation of GeekOS in more detail, and describes some of the important data structures and functions.

3.2.1 Loading and Runtime Environment

GeekOS is booted from a floppy. The bootsector loads a 16 bit setup program and the kernel image from the floppy into memory, and then jumps to the setup program. The setup program initializes the hardware sufficiently to enter 32 bit protected mode, sets up the stack of the initial kernel thread, and jumps to the `Main()` function, which is the kernel's entry point.

The `Main()` function calls the initialization functions for the various kernel subsystems, and then starts user programs which are specified in a special data structure linked into the kernel.

3.2.2 Memory

Because GeekOS does not use paging, all kernel code runs using physical addresses. Figure 1 shows how GeekOS and its data structures occupy physical memory.

The bootsector places the kernel at address 10000h (64K). The setup code and data is placed at address 90000h; however, once the kernel's `Main()` function is entered, the setup program is not used again, and its memory is reclaimed.

After filling its `.bss` segment with zeroes and initializing the VGA text display, the kernel builds the *page list* data structure. This is an array of `Page` structs, one for each page of physical memory, and is allocated immediately after the kernel image. The `Page` struct is defined as follows:

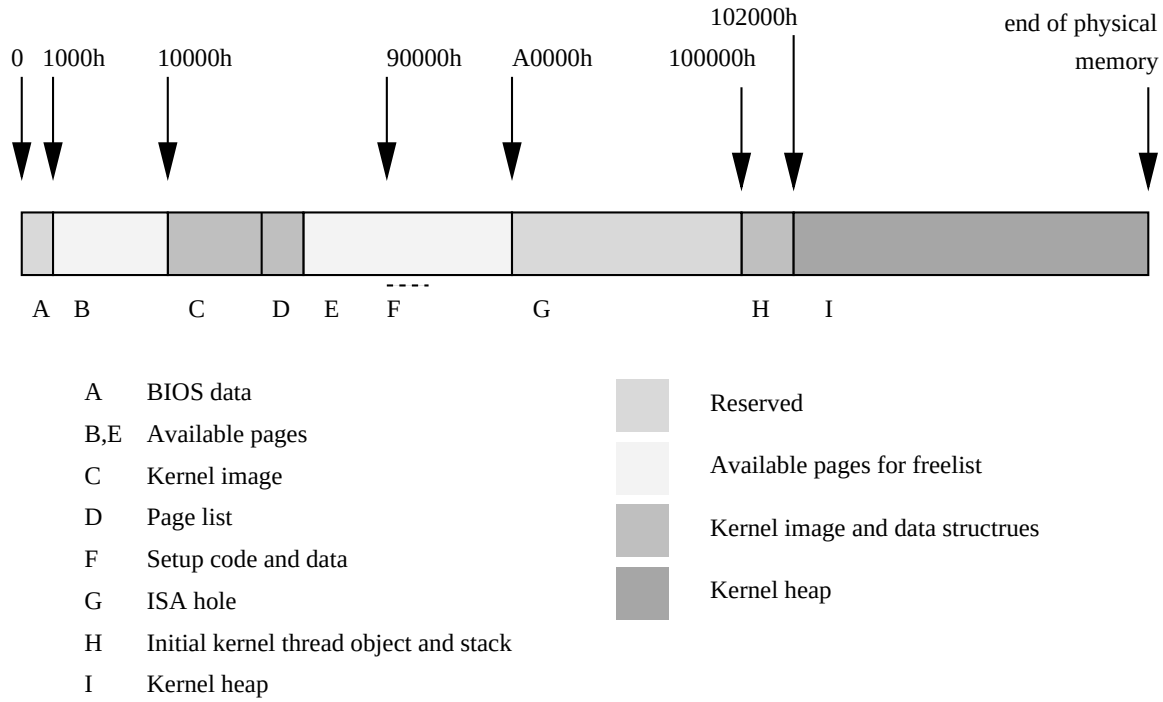

Figure 1: GeekOS memory map (not to scale).

```
#define PAGE_KERN      0x0001   // page used by kernel code or data
#define PAGE_HW        0x0002   // page used by hardware (e.g., ISA hole)
#define PAGE_ALLOCATED 0x0004   // page is allocated
#define PAGE_UNUSED    0x0008   // page is unused
#define PAGE_HEAP      0x0010   // page is in kernel heap

struct Page {
    unsigned flags;
    unsigned long nextFree;
};
```

The `flags` field indicates the current state of the page. The `nextFree` field is used for the *page freelist*, which is a list of pages available to the page allocator. In the future, other fields will be added to the `Page` struct to handle the association of pages with virtual memory objects.

The page allocator, accessible through the `Alloc_Page()` and `Free_Page()` functions, allocates pages of physical memory. Its pool of available pages consists of all pages below A0000h except for page 0 (which contains BIOS data) and pages used by the kernel image and the page list. The page allocator is currently only used for creation of `Kernel_Thread` objects and their stacks. When support for virtual memory is added to GeekOS, the page allocator will be used as a source of physical pages to the virtual memory subsystem.

From A0000h to 100000h is the ISA hole, which is reserved for PC hardware devices such

as VGA memory. Currently the only part of this region used by GeekOS is the VGA text screen, located at address B8000h.

Immediately following the ISA hole are two pages of memory for the initial kernel thread and its stack. All remaining memory is used by the kernel heap allocator. The heap allocator, accessible through functions `Malloc()` and `Free()`, provides general purpose dynamic allocation for objects of any size. I used a public domain library called BGET[BGE01] to implement the heap allocator, since it was not dependent on any library and thus was easy to incorporate into GeekOS. Currently, the heap allocator is only used for the allocation of memory for user programs. When virtual memory is added to GeekOS, the kernel heap will grow dynamically as needed rather than occupying a fixed region. It will also be implemented on top of the page allocator, rather than being disjoint from it.

3.2.3 Kernel Threads

Kernel threads are the abstraction for a schedulable entity in GeekOS. I chose a threaded model for GeekOS because it is simple to understand, and should be familiar to students who have used user-level threads. A kernel thread may execute entirely in kernel mode, or it may have a *user context*. The `Kernel_Thread` struct is defined as follows:

```
struct Kernel_Thread {
    unsigned long esp;
    volatile unsigned long numTicks;
    int priority;
    DEFINE_LINK( Thread_Queue, Kernel_Thread );
    void* stackPage;
    struct User_Context* userContext;
    struct Kernel_Thread* owner;
    int refCount;
    Boolean alive;
    struct Mutex joinLock;
    struct Condition joinCond;
};
```

The `esp` field is used to save the thread's stack pointer when the thread is suspended. Section 3.2.4 will describe context saving and restoring in more detail. The `stackPage` field points to the kernel thread's stack page. The `numTicks` and `priority` fields are used by the scheduler to implement timer based preemption and priority-based scheduling, respectively. The `DEFINE_LINK` macro defines previous and next fields used when a kernel thread is on a *thread queue*. The `userContext` field, if non-null, points to the thread's user context, which is essentially a combined code and data segment allowing the thread to execute a user mode program. Section 3.2.6 will describe user contexts in more detail. The remaining fields are used in the implementation of the `Join()` function, which allows a parent thread to wait for a child thread to exit.

Figure 2 shows the prototypes for the functions which create and destroy kernel threads.

```
typedef void (*Thread_Start_Func)( unsigned long arg );

struct Kernel_Thread* Start_Kernel_Thread(Thread_Start_Func startFunc,
   unsigned long arg, int priority, Boolean detached);
struct Kernel_Thread* Start_User_Thread(struct User_Context* userContext,
   unsigned long entryAddr, Boolean detached);
void Exit( void );
```

Figure 2: Thread creation and destruction functions.

```
void Schedule( void );
void Yield( void );
void Wait( struct Thread_Queue* waitQueue );
void Wake_Up( struct Thread_Queue* waitQueue );
```

Figure 3: Voluntary thread scheduling functions.

Kernel threads may be created in two ways. Threads that execute exclusively in kernel mode are created by the `Start_Kernel_Thread()` function, which takes a pointer to a start function which implements the body of the thread. Threads executing user mode programs are created with the `Start_User_Thread()` function, which takes a pointer to a `User_Context` and the address of the code entry point within the user context memory.

Kernel threads are destroyed when they voluntarily call the `Exit()` function. Because a thread may be suspended while executing a critical section guarded by a mutex, GeekOS does not provide a way for one thread to kill another, because there is currently no mechanism for ensuring that mutex locks are released.

3.2.4 Interrupts and Scheduling

In GeekOS, a thread context switch may happen for two reasons.

First, a thread may voluntarily give up the CPU by calling the `Yield()` or `Wait()` functions. In each of these functions, the current thread is placed on a thread queue and a new thread is selected by calling the `Schedule()` function. The `Yield()` function places the calling thread on the *run queue*, from which it may be rescheduled. The `Wait()` function places the calling thread on a *wait queue*, where it will remain until another thread or interrupt handler places it back on the run queue using the `Wake_Up()` function. The voluntary thread scheduling functions are shown in Figure 3.

Second, a thread may be preempted at any point where interrupts are enabled. The timer interrupt handler increments the current thread's `numTicks` field, and if it has exceeded its time slice, it is suspended and a new thread is chosen.

```
struct Mutex;
struct Condition;

void Mutex_Lock( struct Mutex* mutex );
void Mutex_Unlock( struct Mutex* mutex );
void Cond_Wait( struct Condition* cond, struct Mutex* mutex );
void Cond_Signal( struct Condition* cond );
void Cond_Broadcast( struct Condition* cond );
```

Figure 4: Thread synchronization functions.

In an early version of GeekOS (which did not support timer based preemption), I used an explicit data structure to store the suspended thread's saved context for the voluntary context switches. ('Context' refers to the thread's current register contents and program counter.) A separate mechanism was used to save and restore the executing thread's context on the stack when an interrupt occurred. This created two mechanisms for performing essentially the same task. Therefore, I reimplemented the voluntarily thread scheduling functions to save and restore the thread's context on the stack using the same format as the interrupt handling mechanism[3]. The `Switch_To_Thread()` function manipulates the stack to make it appear that an interrupt has occurred in the thread being suspended. The `Restore_Thread()` function works the same way as the interrupt return code, and in fact returns control to the restored thread using the `iret` instruction. By making the voluntary thread scheduling functions compatible with the interrupt handling code, the interrupt return code can switch to a new thread by simply saving the old thread's stack pointer into the `Kernel_Thread` object's `esp` field, switching to the new thread's stack by loading its `esp` field, and then executing a normal interrupt return.

By unifying interrupt handling and thread context switching, it was trivial to allow interrupt handlers to notify the interrupt return code that a new thread should be chosen. A global boolean variable, `g_needReschedule`, is checked upon each return from an interrupt handler. If set, the current thread is suspended and a new thread is chosen.

The current scheduler (the `Get_Next_Runnable()` function) uses a static priority scheme to determine the next thread to run. Threads of equal priority will share the CPU in round-robin fashion. A useful student project would be to implement an adaptive dynamic scheduler to ensure that low priority threads are not starved by higher priority threads.

3.2.5 Thread Synchronization

In order to synchronize kernel threads, GeekOS provides *mutexes* and *condition variables*, which are modeled on those provided by POSIX threads. Mutexes are used to mediate exclusive access to data structures shared by multiple threads. Condition variables allow threads to wait for a condition to be satisfied, or to signal that a condition has been satisfed.

[3]I read about this technique on the `alt.os.development` newsgroup, so I can't take credit for it.

Prototypes of the thread synchronization functions are shown in Figure 4. These functions are implemented using the lower-level `Wait()` and `Wake_Up()` functions.

3.2.6 User Mode

Ultimately, the usefulness of any OS kernel lies in its ability to run other programs. It should do so in a manner that protects the kernel from user programs and protects user programs from each other. Therefore, a user mode for GeekOS was essential in order for it to be realistic.

In order to get something running quickly, I used the x86 segmentation hardware to create a restricted environment for user mode programs. User code may access and modify its own data and make system calls, but all other OS resources are off limits. Any attempt by a user program to access code or data outside its own sandbox results in a general protection fault which kills the offending thread.

A user process is created by attaching a *user context* to a kernel thread. The `User_Context` struct is defined as follows:

```
struct User_Context {
    struct Segment_Descriptor ldt[2];
    struct Segment_Descriptor* ldtDescriptor;
    void* memory;
    unsigned long size;
    int refCount;
    unsigned short ldtSelector;
    unsigned short csSelector;
    unsigned short dsSelector;
};
```

The `ldt` field is a local descriptor table (LDT) in which the user context's code and data segments are defined. The `ldtDescriptor` field points to the LDT's descriptor in the global descriptor table (GDT). Each user context has its own LDT. By switching LDTs each time a new user context is activated, GeekOS prevents user programs from accessing or modifying memory other than their own. The `memory` and `size` fields define the memory containing the context's code and data. User programs use a single chunk of contiguous memory for code, data, and stack. Because all loads, stores, and instruction fetches are done from the user data and code segments, the user program thinks it is running in a flat memory space beginning at address 0 and ending at address `size`.

User programs communicate with the kernel through a system call interface. A system call is made by loading the system call number into the `eax` register, loading parameters into other registers, and invoking software interrupt 144. The system call interrupt handler verifies that a valid system call has been requested, enables interrupts, calls the handler function, disables interrupts, and returns the result in the `eax` register. Copyin and copyout functions are provided for moving data between user space and kernel space. Several demonstration system calls, along with their user-mode wrappers, are provided as a model for students who need to implement their own system calls.

GeekOS does not currently have storage device drivers or a filesystem. Therefore, it provides a way to link user programs directly into the kernel as data objects. User executables are first linked at base address 0. From each executable, a flat binary image is produced. A perl script then converts the images into C data structures (called `User_Programs`). These data structures are collected into a table from which the kernel can look them up by name, and then launch them with the `Start_User_Program()` function.

While the current user mode system was simple to implement and is easy to understand, it has a number of shortcomings. First, because the user memory is fixed size, it would be difficult to implement a user mode heap (using something like the Unix `sbrk()` system call). We could play games such as placing the user program's stack below its heap, but this is not an elegant solution. Another problem is that because `gcc` (the compiler used to compile GeekOS) does not support segmented memory, it would be difficult to implement shared libraries or shared memory. Finally, there is no way to trap null pointer dereferences using segmentation.

I hope to add support for paged virtual memory to the next version of GeekOS. Some thought will need to be given to how virtual memory can be implemented in a way that preserves GeekOS's simplicity, which I consider to be an essential property in its role as an instructional OS kernel.

3.2.7 GeekOS Source Code

The source code to GeekOS currently consists of about 3,500 lines of C[4] and about 1,000 lines of assembly. Of these totals, about 1,600 lines are comments or blank.

I made a conscious effort to write GeekOS in a way that would allow the source code to be understood easily. In addition to liberally commenting the source code, I took pains to avoid the use of cryptic identifiers or idioms[5]. From comments I've received from visitors to the GeekOS web site, I think I've suceeded in making the code readable.

4 GeekOS as an Instructional Operating System

GeekOS was used as the basis for student projects in the Fall 2001 undergraduate operating systems course (CMSC 412) at the University of Maryland. Several possibilities were suggested to the students, who were allowed to choose among them:

- implement a slab memory allocator[Bon94]
- implement multi-level and round-robin scheduling algorithms
- implement adaptive mutexes
- implement `fork()` and `exec()` system calls

[4]This figure does not include the kernel heap memory allocator.

[5]Operating systems hackers have a well-deserved reputation for coding in an intentionally obscure style, as though the mark of a good programmer was the incomprehensibility of his or her code.

- implement IPC using unidirectional pipes

In addition, some students choose their own projects, which included implementing paged virtual memory, implementing an IDE device driver, and implementing an ethernet device driver.

In order to gauge the effectiveness of GeekOS as an instructional operating system, I distributed a survey to the students. The survey asked them to describe their experiences working with GeekOS and the Bochs emulator. In particular, I wanted to answer three main questions. First, was working at the level of bare hardware useful, or was it an unnecessary source of complications? Second, was working with Bochs straightforward, or was it too complex or slow? Third, did GeekOS itself help reinforce the concepts taught in the class?

On the whole, the students' responses were positive. Out of 25 completed surveys, only one student felt that GeekOS was not easy to understand (14 found it 'somewhat' easy to understand, and 10 responded with an unqualified 'yes'). When asked whether GeekOS helped them understand the concepts taught in the course, 15 students answered 'somewhat', and 10 answered 'yes'. Most of the students (18) responded 'yes' when asked whether they found it interesting or useful that their projects were capable of running on bare hardware, while 5 answered 'somewhat' and 2 answered 'no'. I conclude that GeekOS was thus reasonably effective as an instructional OS kernel, and that the decision to work at the hardware level was justified.

Before the students started working with GeekOS, I was concerned that the performance of Bochs and the cross-development environment set up on the academic computing cluster might not be adequate, since the cluster is heavily loaded. I was therefore relieved to find out that most students did their projects on their own machines (largely using Windows, with a few students using Linux). The three students who did develop their projects on the cluster found that the performance was adequate. From these responses I conclude that using Bochs in an academic setting is entirely feasible, even when some students are using a time-shared system.

I was pleasantly surprised to find that most students found Bochs to be easy (11) or somewhat easy (11) to work with. Only 3 did not find Bochs to be easy to work with. This is confirmation that Bochs is well-designed and reliable, which has been my experience in my own work with it. However, most students (20) did not use the built-in debugger.

Finally, the survey asked whether the students thought GeekOS was a good basis for the course projects. 19 answered 'yes', 5 answered 'somewhat', and 1 answered 'no'. The survey also solicited general comments on GeekOS in its role as an instructional tool. Two students said that they would have liked more pointers on where to go for hardware-level programming information. Two students felt that GeekOS was not well integrated into the course cirriculum, and would have preferred to use it earlier in the semester. One student would have liked to have comprehensive reference documentation on GeekOS internals (since currently the only such documentation is comments in the source code).

From the student feedback, I think GeekOS was largely effective in its role as an instructional OS. However, there are many ways in which it could be improved. I think detailed reference documentation with pointers to hardware programming information would be extremely useful, both for students and for teaching assistants. In addition, in future semesters

I think it will be possible to integrate GeekOS more smoothly into the cirriculum for the course, using the experience gained this semester.

5 Future Work

To make GeekOS a more effective educational tool, comprehensive reference documentation needs to be written. Because x86 PCs (as emulated by Bochs) are quite easy to program, and because GeekOS is a fairly simple piece of software, I think good documentation could be put together pretty easily.

One of the most difficult problems encountered in hardware-level programming is the difficulty of getting good diagnostic information when errors occur. Bochs includes a built-in assembly level debugger. However, since GeekOS is written in C, stepping through the assembly code emitted by the compiler is not especially helpful. It would be very nice to add support for source-level debugging to the Bochs debugger. I think this would greatly reduce the burden on students to understand complex hardware issues, and this help fulfill the promise of hardware emulation as a pedagogical vehicle.

In addition to improving the suitability of GeekOS as an instructional tool, I plan to continue its development as an OS kernel. The most urgent feature needed by GeekOS is support for paged virtual memory. This would allow much more flexible control over the memory layout of user mode processes. For example, dynamic expansion of the heap and user stack(s) would be much simpler to implement with paging than with segmentation. In addition, support for shared memory regions would become possible. However, paging is significantly more complex than segmentation, so finding a way to support paging without greatly increasing the complexity of GeekOS will be an interesting challenge.

Device drivers such as disk and ethernet could be useful, since they would make it easier for students to create filesystems and network stacks as projects. However, these are also interesting as projects in their own right, so incorporating them into the base system might not be desirable.

GeekOS currently does not have any mechanism for interprocess communication or resource naming. While I have some ideas on the drawing board, I think they are too complex and specific to incorporate into the base system. Since these are also good possibilities for student projects, I think leaving them out is preferable.

I think it is likely that GeekOS will split into two branches. The 'classic' system will just add paging to the current code base. The 'experimental' system will incorporate new drivers and subsystems, and possibly ports to other architectures, and thus be less suited for instructional use. Ultimately, I would like to develop the experimental version of GeekOS to the point where it can self-host its own development; obviously this goal is relatively far off.

6 Conclusions

I think the experience with GeekOS has shown that programming for bare hardware is an effective way to teach operating system principles. I have received several comments, from students as well as visitors to the GeekOS web site, that GeekOS has helped them understand

OS programming, and in this way it has satisified my original goals. I think with further refinement it can become even better as an educational tool.

Acknowledgements

First, I would like thank Bobby Bhattacharjee for founding the University of Maryland's OS reading group, which spurred my interest in operating systems and lead to the creation of GeekOS. He also decided to use GeekOS for the undergraduate operating systems course (CMSC 412), which resulted in this paper. I would like to thank the students of CMSC 412 for their feedback on GeekOS. The members of the `alt.os.development` community, in addition to providing many good ideas, answered several questions I had during the development of GeekOS.

References

[BGE01] The BGET Memory Allocator, http://www.fourmilab.ch/bget, 2001.

[Boc01] Bochs, the Cross-Platform IA-32 Emulator, http://bochs.sourceforge.net, 2001.

[Bon94] Jeff Bonwick. The Slab Allocator: An Object-Caching Kernel Memory Allocator. In *Proceedings of the USENIX Summer Technical Conference*, pages 87–98, 1994.

[CPA93] Wayne A. Christopher, Steven J. Proctor, and Thomas E. Anderson. The Nachos Instructional Operating System. Technical Report CSD-93-739, University of California, Berkeley, Computer Science Division, 1993.

[Gee01] GeekOS web site, http://geekos.sourceforge.net, 2001.

[Min01] Minix Information Sheet, http://www.cs.vu.nl/~ast/minix.html, 2001.

[TW97] Andrew S. Tanenbaum and Albert S. Woodhull. *Operating Systems: Design and Implementation*. Prentice-Hall, second edition, 1997.