Project 3: Writing a Kernel From Scratch

15-410 Operating Systems

February 21, 2005

Contents

1 Introduction

This document will serve as a guide in completing the 15-410 kernel project. The goal of this document is to supply enough information to complete the project without getting bogged down in implementation details. Information contained in lecture notes, or in the Intel documentation will be repeated here only sparingly, and these sources will often be referenced, so keep them handy. Good luck!

1.1 Overview

This project will require the design and implementation of a Unix-like kernel. The 410 kernel will support multiple virtual memory address spaces via paging, preemptive multitasking, and a small set of important system calls. Also, the kernel will supply device drivers for the keyboard, the console, and the timer.

1.2 Goals

- Acquiring a deep understanding of the operation of a Unix-like kernel through the design and implementation of one.
- Gaining experience reading technical specifications such as the Intel documentation.
- Debugging kernel code. Virtual memory, interrupts, and concurrency concerns add complexity to the debugging process.
- Working with a partner. Learning how to program as a team (Pair Programming, division of labor, etc.). Using source control.

1.3 Technology Disclaimer

Because of the availability, low cost, and widespread use of the x86 architecture, it was chosen as the platform for this sequence of projects. As its creator, Intel Corporation has provided much of the documentation used in the development of these projects. In its literature Intel uses and defines terms like interrupt, fault, etc.. On top of this the x86 architecture will be the only platform used in these projects.

The goal of this project set is certainly not to teach the idiosyncrasies of the x86 architecture (or Intel's documentation). That said, it will be necessary to become accustomed to the x86 way of doing things, and the Intel nomenclature, for the purposes of completing this project set. Just keep in mind that the x86 way of doing things is not the only way of doing things. It is the price to be paid for learning the principles of operating systems on a real world system instead of a simulated architecture.

1.4 Important Dates

- Monday, February 21st: Project 3 begins.
- Wednesday, March 2nd: Checkpoint 1 due (in-cluster demo).
- Friday, March 18th: Checkpoint 2 target date (this may be adjusted later by a day or two).
- Friday, April 8th: Project 3 due.

1.5 Groups

The kernel project is a group assignment. You should already be in a group of two from the previous project. If you are not in a group, or you are having other group difficulties, send email to staff-410@cs.cmu.edu. If you experienced partner difficulties during the previous project, it will be very important to *resolve* them in the first week of the kernel project. In order to do well on this project, you will need to work *more* closely and productively with your partner than was true for the thread library project. If things are on a downward trend, that's unlikely to happen.

In order for you to do well in this class, it will be important for you to read and understand your partner's code. We strongly suggest that you schedule time for reading and discussing each other's code at least twice weekly.

1.6 Grading

The primary criteria for grading are correctness, performance, design, and style.

A correct kernel implements the provided specification. Correctness also includes robustness. A robust kernel does not crash (no matter what instructions are executed by user code), handles interesting corner cases correctly, and recovers gracefully from errors.

For the purposes of this class, performance refers mainly to how preemptible your kernel is (see Section 4.1.6). Also, you should avoid standard pitfalls related to putting threads to sleep (see Section 5.3). Preemptibility and sleep/wakeup will represent approximately 10% of the project grade.

We may fuss about code which takes *much* longer than it needs to. For example, if an *easy* O(1) algorithm exists, don't use an O(n) algorithm which requires the same amount of code–this will typically indicate carelessness. On the other hand, we are *not* requiring you to use splay trees or even hash tables where a linked list is appropriate or at least defensible. See Section 9.1.1.

A well designed kernel is organized in an intuitive and straightforward way. Functionality is separated between different source files. Global variables are used when appropriate (they are more appropriate in OS kernels than in most programs), but not to excess (basically, consider the correct scope). Appropriate data structures are used when needed. When practical, data structure implementations are hidden behind an appropriate interface (see Section 9.1.1 below).

A kernel with good style is readable. Some noted deviations from what may be generally considered to be good style will be penalized. Also, poorly commented, hard-to-read code will be

penalized, as will a project that does not follow the prescribed build process. Overall, readability, structure, and maintainability issues will form approximately 10% of the project grade.

Please note that it is the considered opinion of the course staff that inline assembly code (`asm()`), even more than preprocessor macros, has enough defects that its use must always be supported by a conscious argument. For example, once you leave CMU you will probably write code which must run on multiple hardware platforms. This requirement plus inline assembly language immediately results in an `#ifdef` explosion. Also, unless you use the "long form" of `asm()`, which correctly declares to the compiler the C-language effects of your assembly code, you leave yourself open to the substantial risk that a different compiler version, different compilation flags, or even minor changes to your C code may have disastrous interference effects. It is almost always better to make a procedure call to assembly code in a `.S` file.

Your score will be based on a mixture of test suite results and code comments made by a member of the course staff. In addition, we intend to schedule a 30-minute interview/de-briefing session with each group. In order to avoid disappointing grade surprises on the kernel project, you may wish to consult the "Grades" section of the syllabus. While the exact grading criteria and cut-offs differ by semester, turning in a kernel which passes all tests we provide does *not* imply an "A."

While we do not require you to implement copy-on-write or zero-fill-on-demand, many students choose to implement one or the other because it seems more professional. While we concur with the sentiment, please don't let a slick COW implementation (or another exotic option such as a splay-tree scheduler) stop you from getting context switch to be clean and pure and being able to cleanly put threads to sleep and re-awaken them.

In order for us to run the test suite your kernel must *not* drop into the debugger as it is booting, or any other time in the course of a test, as that will result in the test failing. In addition, the shell must work and you must properly implement the `halt()` system call for the tests to run properly.

1.7 Interactions between Project 3 and Project 4

It is likely that groups will not be permitted to do Project 4 unless they complete Project 3 satisfactorily. Detailed go/no-go criteria will be made available to you near the end of Project 3, but your mental model should be that you will need to pass 80% of a test suite which we will provide you with.

Also, Project 4 will probably center on enhancing your Project 3 kernel. This means you will probably need to revise or re-architect some part of your solution to Project 3. It is probably wise to plan ahead for this by writing clean, modular code which you will be able to understand after you turn it in.

1.8 Hand-in

The hand-in directories will be created as the due date nears. More specific instructions will be provided at that time. Subject to later instructions, plan to hand in all source files, header files,

and Makefiles that you write. Plan to keep to yourself disk image files, editor-generated backup files, log files, etc.

When handed in, your kernel must be runnable! This means that it *must*, upon being built and booted, start running `idle`, `init`, and `shell` without user intervention. In particular, it must **not** drop into the simics debugger. When we run the test suite, there will not be a human present to continue execution. Thus, the test harness will declare your kernel to have failed the entire suite.

Also, your kernel should not generate reams of `lprintf()` debugging messages while running. Ideally you should adjust the setting of your trace facility (see Section 9.1) so that it generates *no* messages, but in any case the normal loading, execution, and exiting of a program should not generate more than 20 lines of `kernel.log` output.

2 Hardware Primitives

2.1 Pivilege Levels

The x86 architecture supports four pivilege levels, PL0 through PL3. Lower privilege numbers indicate greater privilege. The kernel will run at PL0. User code will run at PL3.

2.2 Segmentation

A segment is simply a region of the address space. Two notable properties can be associated with a segment: the pivilege level, and whether the segment contains code, stack, or data. Segments can be defined to span the entire address space.

The 410 kernel will use segmentation as little as possible. The x86 architecture requires some use of segmentation, however. Installing interrupt handlers, and managing context switch requires some understanding of segmentation.

In the 410 kernel, there will be four segments. These four segments will each span the entire address space. Two of them will require that the privilege level be set to PL0 to be accessed, and two will require that the privilege level be set to PL3 or lower to be accessed. For each pair of segments, one will be code and one will be data.

2.3 Special Registers

This project requires an understanding of some of the x86 processor data structures. This section will cover some important structures that the kernel must manipulate in order to function properly.

2.3.1 The Segment Selector Registers

There are six segment selector registers: `%cs`, `%ss`, `%ds`, `%es`, `%fs`, and `%gs`. A segment selector is really an index into one of two processor data structures called the Global Descriptor Table (GDT) and Local Descriptor Table (LDT). These tables are where the segments are actually defined.

The provided startup code sets up segment descriptors in the GDT, but it is the resposibility of the kernel to have the correct values in the segment selector registers on entering and leaving the kernel. The code segment selector for the currently running thread is stored in `%cs`. The stack segment selector for the currently running thread is stored in `%ss`. It is possible to specify up to four data segment selectors. They are `%ds` through `%gs`. The code segment selector is used to access instructions. The stack segment selector is used in stack related operations (i.e., `PUSH`, `POP`, etc.). The data segment selectors are used in all other operations that access memory.

On entering the `kernel_main()` function, the kernel and user segments have already been installed into the GDT. When a user thread is started, user level code, stack, and data segment selectors need to be specified and loaded into the segment selector registers. When a user thread takes an interrupt, the code and stack segment selector registers will be saved automatically. The data segment selector registers and the general purpose registers will not be saved automatically, however.

For more information on the GDT and segmentation please review the relevant lecture notes and consult your textbook, sections 2.1, 2.4, and 3.2 of `intel-sys.pdf`, and the segmentation handout on the course web site.

2.3.2 The EFLAGS Register

The `EFLAGS` register controls some important processor state. It will be necessary to provide the correct value for the `EFLAGS` register when starting the first user thread, so it is important to understand its format. The `EFLAGS` register is discussed in section 2.3 of `intel-sys.pdf`. `410kern/lib/inc/x86/eflags.h` contains useful definitions. The bootstrap process sets `EFLAGS` to an appropriate value, available to you via the `get_eflags()` macro from `410kern/lib/inc/x86/proc_reg.h`, for your kernel execution. Before entering user mode you will need to arrange for bit 1 ("reserved") to be 1, and, after studying what they do, should arrange for the `IF` and `IOPL_KERNEL` bits to be set appropriately. The first method you think of for doing this may not be the right method.

2.3.3 Control Registers

- Control Register Zero (`%cr0`): This control register contains the most powerful system flags. The 410 kernel will only be concerned with bit 31, which activates paging when set, and deactivates it when unset. Paging is discussed below. Do not modify the state of any of the other bits.
- Control Register One (`%cr1`): This control register is reserved and should not be touched.
- Control Register Two (`%cr2`): When there is a page fault, `%cr2` will contain the address that caused the fault. This value will be needed by the page fault handler.
- Control Register Three (`%cr3`): This control register is sometimes known as the Page Directory Base Register (PDBR). It holds the physical address of the current page directory in its top 20 bits. Bits 3 and 4 control some aspects of caching and should both be unset.

The %cr3 register will need to be updated when switching address spaces. Writing to the %cr3 register invalidates entries for all pages in the TLB not marked global.

- Control Register Four (%cr4): This control register contains a number of extension flags that can be safely ignored by the 410 kernel. Bit 7 is the Page Global Enable (PGE) flag. This flag should be set for reasons discussed below.

2.3.4 The Kernel Stack Pointer

In the x86 architecture, the stacks for user level code and kernel level code are separate. When an interrupt occurs that transitions the current privilege level of the processor to kernel mode, the stack pointer is set to the top of the kernel stack. A small amount of context information is then pushed onto the stack to allow the previously running thread to resume once the interrupt handler has finished.

The value of the stack pointer when we enter kernel mode is defined by the currently running task. Tasks are a hardware "process" mechanism provided by the x86 architecture. The 410 kernel will not use tasks. It is faster to manipulate the process abstraction in software. It is necessary, however, to define at least one task. This takes place in the bootstrapping code, before execution of the kernel_main() function begins. The provided function set_esp0(), defined in 410kern/lib/x86/seg.c, will specify the beginning value for the kernel stack pointer the next time a user-to-kernel transition occurs.

2.3.5 C interface

There are inline assembly macros defined in 410kern/lib/inc/x86/proc_reg.h, that can be used to read and write many of the processor's registers.

2.4 Paging

The x86 architecture uses a two-level paging scheme with four-kilobyte pages. It is also possible to use larger page sizes, though this is outside the scope of this project. The top level of the paging structure is called the page directory, while the second level consists of objects called page tables. The formats of page directory entries and page table entries are very similar. However, their fields have slightly different meanings. Here is the format of both a page directory entry and a page table entry.

Entries in both tables use the top twenty bits to specify an address. A page directory entry specifies the physical memory address of a page table in the top twenty bits. A page table entry specifies the number of a physical frame in the top twenty bits. Both page tables and physical frames must be page aligned. An object is page aligned if the bottom twelve bits of the lowest address of the object are zero.

The bottom twelve bits in a page directory or page table entry are flags.

- Bit 0: This is the present flag. It has the same meaning in both page directories and page tables. If the flag is unset, then an attempt to read, write, or execute data stored at an

address within that page (or a page that would be referenced by the not present page table) will cause a page fault to be generated. On installing a new page table into a page directory, or framing a virtual page, the present bit should be set.

- Bit 1: This is the read/write flag. If the flag is set, then the page is writable. If the flag is unset then the page is read-only, and attempts to write will cause a page fault. This flag has different meanings in page table and page directory entries. See the table on page 136 of `intel-sys.pdf` for details.

- Bit 2: This is the user/supervisor flag. If the flag is set, then the page is user accessible. This flag has different meanings in page table and page directory entries. See the table on page 136 of `intel-sys.pdf` for details.

- Bit 3: This is the page-level write through flag. If it is set, write-through caching is enabled for that page or page table, otherwise write-back caching is used. This flag should be left unset.

- Bit 4: This is the page-level disable caching flag. If the flag is set, then caching of the associated page or page table is disabled. This flag should be left unset.

- Bit 5: This is the accessed flag. It is set by the hardware when the page pointed to by a page table entry is accessed. The accessed bit is set in a page directory entry when any of the pages in the page table it references are accessed. This flag may be ignored by the 410 kernel.

- Bit 6: This is the dirty flag. It is valid only in page table entries. This flag is set by the hardware when the page referenced by the page table entry is written to. This flag can be used to implement demand paging. However, this flag may be ignored by the 410 kernel.

- Bit 7: This is the page size flag in a page directory entry, and the page attribute index flag in a page table entry. Because the 410 kernel uses four kilobyte pages all of the same type, both of these flags should be unset.

- Bit 8: This is the global flag in a page table entry. This flag has no meaning in a page directory entry. If the global flag is set in a page table entry, then the virtual-to-physical mapping will not be flushed from the TLB automatically when writing `%cr3`. This flag should be used to prevent the kernel mappings from being flushed on context switches. To use this bit, the page global enable flag in `%cr4` must be set.

- Bits 9, 10, 11: These bits are left available for use by software. They can be used to implement demand paging. The 410 kernel may ignore these bits.

2.5 The Layout of Physical Memory

Although there are many ways to partition physical memory, the 410 kernel will use the following model. The bottom 16MB of physical memory (from address 0x00000000 to address 0x00ffffff,

i.e., just under `USER_MEM_START` as defined by `410kern/lib/inc/x86/seg.h`), is reserved for the kernel. This kernel memory should appear as the bottom 16MB of each task's virtual address space (that is, the virtual-to-physical mapping will be the identity function for the first 16 megabytes; this is known as "direct mapping", or "V=R" in the IBM mainframe world).

Note that user code should not be able to read from or write to kernel memory, even though it is resident at the bottom of each user task's address space. In other words, from the point of view of user code, memory between 0x00000000 and 0x00ffffff should be just as invalid as any other memory not part of the text, data, bss, automatic stack, or `new_pages()`-allocated regions.

The remainder of physical memory, i.e., from 0x01000000 up, should be used for frames. In `410kern/lib/inc/x86/seg.h` is a prototype for a function `int machine_phys_frames(void)`, provided by `410kernel/lib/x86/seg.c`, which will return to you the number of `PAGE_SIZE`-sized frames supported by the simics virtual machine you will be running on (`PAGE_SIZE` and the appropriate `PAGE_SHIFT` are located in `410kern/lib/inc/page.h`). This frame count will include both kernel frames and user memory frames.

Please note that the memory-allocation functions discussed below (e.g., `malloc()`) manage *only* kernel virtual pages. You are responsible for defining and implementing an allocator appropriate for the task of managing free physical frames.

3 The Boot Process

The boot process is somewhat complicated, and it is not necessary to fully understand it in order to complete this project. To learn more about the boot process, please read about the GRUB boot loader (`http://www.gnu.org/software/grub/`). This is the boot loader that will be used to load the 410 kernel. The 410 kernel complies with the Multiboot specification as defined at `http://www.mcc.ac.uk/grub/multiboot_toc.html`. After the boot loader finishes loading the kernel into memory, it invokes the function `multiboot_main()` in `410kern/lib/multiboot/base_multiboot_main.c`. The support code in `410kern/lib/multiboot` ensures that the 410 kernel follows the Multiboot specification, initializes processor data structures with default values, and calls the 410 `kernel_main()` function.

4 Device Drivers and Interrupt Handlers

Your work on Project 1 provided you with most of the interrupt-handling knowledge necessary for the kernel project. In this section we will briefly cover ways in which Project 3 demands more or different treatment.

4.1 Interrupts, Faults, and Exceptions

The Interrupt Descriptor Table (IDT) contains entries for hardware device interrupts (covered in Project 1), software interrupts (which you invoked in Project 2), and exception handlers. Exceptions are conditions in the processor which are usually unintended and must be addressed. Page faults, divide-by-zero, and segmentation faults are all types of exceptions.

4.1.1 Hardware Interrupts

You will support the same hardware devices you did in Project 1, though your device drivers will be somewhat more complicated.

4.1.2 Software Interrupts

Hardware interrupts are not the only type of interrupt. Programs can issue software interrupts as well. These interrupts are often used as a way to transfer execution to the kernel in a controlled manner, for example during a system call. To perform a software interrupt a user application will execute a special instruction, `INT n`, which will cause the processor to execute the `n`'th handler in the IDT.

In this project, software interrupts will always cause a privilege level change, and you will need to understand what the CPU hardware places on the stack during a privilege level change (see page 152–153 of `intel-sys.pdf`).

4.1.3 Faults and Exceptions

Please read section 5.3 of `intel-sys.pdf` on Exception Classifications. Note that entries exist in the IDT for faults and exceptions. The 410 kernel should handle the following exceptions: Division Error, Device Not Present, Invalid Opcode, Alignment Check, General Protection Fault, and Page Fault. On each of these exceptions, the kernel should report the virtual address of the instruction that caused the exception, along with any other relevant information (i.e., for page faults which will kill a thread, the program counter, address which generated the fault and the reason the memory access was invalid).

If the kernel decides to kill a thread due to an exception, this must be done cleanly. In particular, any kernel resources related to the thread must be reclaimed. In general, your kernel should treat the exception as if the thread had invoked `exit(-2)`, including, if necessary, the standard mechanisms related to task exit.

4.1.4 Interrupt Handler Flavors

As mentioned previously, an x86 processor uses the IDT to find the address of the proper interrupt handler when an interrupt is issued. To install interrupt, fault, and exception handlers, entries must be installed in the IDT.

An IDT entry can be one of three different types: a task gate, an interrupt gate, or a trap gate. Task gates make use of the processor's hardware task switching functionality, and so are inappropriate for the 410 kernel. Interrupt gates and trap gates differ in that an interrupt gate causes interrupts to be disabled before the handler begins execution. You should think about which kind of gate is appropriate for system calls, and also which kind is appropriate for your hardware device drivers. Some reasonable designs require a mixture of interrupt gates and trap gates. One question which might guide your thinking is, "What happens if a timer interrupt interrupts my keyboard interrupt handler?" It is probably a good idea for your documentation to explain which gate flavors you used for what, and why.

The format of the trap gate is on page 151 of `intel-sys.pdf`. Note that regardless of the type of the gate, the descriptor is 64 bits long. To find out the base address of the IDT, the instruction SIDT can be used. A C wrapper around this instruction is defined in the support code in `410kern/lib/x86/seg.c`. The prototype can be found in `410kern/lib/inc/x86/seg.h`.

The purpose of some of the fields of a trap gate are not obvious. The DPL is the privilege level required to execute the handler. The offset is the virtual address of the handler. The Segment Selector should be set to the segment selector for the target code segment. This is `KERNEL_CS_SEGSEL` defined in `410kern/lib/inc/x86/seg.h`.

4.1.5 Writing an Interrupt Handler

As mentioned above, when the processor receives an interrupt it uses the IDT to start executing the interrupt handler. Before the interrupt handler executes, however, the processor pushes some information onto the stack so that it can resume its previous task when the handler has completed. The exact contents and order of this information is presented on pages 152–153 of `intel-sys.pdf`.

You will probably wish to save (& later restore) additional information on the stack, such as general-purpose registers (`PUSHA` and `POPA` may be useful; see pages 624 and 576 of `intel-isr.pdf`)) and segment registers.

4.1.6 Interrupts and Preemption

The 410 kernel has a number of critical sections. It may be necessary to disable interrupts to protect these critical sections. Interrupts can be disabled by the macro `disable_interrupts()` defined in `410kern/lib/inc/x86/proc_reg.h`, or by the `CLI` instruction. Interrupts can be enabled by the macro `enable_interrupts()` defined in the same file, or by the `STI` instruction.

Our infrastructure for measuring your kernel's preemptibility during grading requires that the `halt()` system call to be implemented (in terms of `SIM_halt()`). Please don't overlook this!

Your 410 kernel should be as preemptible as possible. This means that, ideally, no matter what code sequence is running, whether in kernel mode or user mode, when an interrupt arrives it can be handled and can cause an immediate context switch if appropriate. In other words, if an interrupt signals the completion of an event that some thread is waiting for, it should be possible for your kernel to suspend the interrupted thread and resume the waiting thread so that returning from the interrupt activates the waiting thread rather than the interrupted thread.

We will probably assign you a higher grade on Project 3 if your kernel *always* switches to the waiting thread instead of resuming the interrupted thread (this is not necessarily the right thing to do for performance reasons, but it will help you design for preemptibility and debug your context switching).

To do this, you will need to strive to arrange that as much work as possible is performed within the context of a single thread's kernel execution environment without dependencies on other threads. When race conditions with other processes are unavoidable, try to structure your code so that task/thread switching is disabled for multiple short periods of time rather than for one long period of time.

Please avoid any temptation to "fix" preemptibility issues by delegating work to special "kernel tasks," especially if they would result in serialization of important system calls. As a heuristic, you should make sure that multiple invocations of `fork()` and `new_pages()` can be running (and making progress) "simultaneously" (interleaved by clock interrupts).

A portion of your grade will depend on how preemptible your kernel is.

4.2 Device Drivers

Through the system call interface you will expose the functionality provided by a timer driver, a keyboard driver, and a console driver oddly similar to those you created for Project 1. Since you and your partner both implemented these drivers, this provides you with an excellent opportunity to read, review, and discuss each other's code before beginning to write new code together. Please take advantage of this opportunity!

4.2.1 Floating-Point Unit

Your processor comes equipped with a floating-point co-processor capable of amazing feats of approximation at high speed. However, for historical reasons, the x86 floating-point hardware is baroque. We will not require you to manage the user-visible state of the floating-point unit.

The bootstrapping code we provide will initialize the floating-point system so that any attempt to execute floating-point instructions will result in a "device not present" exception (see the Intel documentation for the exception number). You should do something reasonable if this occurs, i.e., kill the offending user thread (optional challenge: support floating-point).

Please note that since the floating-point hardware is not being managed correctly *you should not use floating-point variables or code in your kernel code or user-space test programs*.

5 Context Switching and Scheduling

5.1 Context Switching

Context switching is historically a conceptually difficult part of this project. Writing a few assembly language functions is usually required to achieve it.

In a context switch, the general purpose registers and segment selector registers of one thread are saved, halting its execution, and the general purpose registers and segment selector registers of another thread are loaded, resuming its execution. Also, the address of the page directory for the thread being switched to is loaded into `%cr3`.

Note that it is possible to context-switch from one thread running in kernel mode to another thread which is runnable and also in kernel mode. In other words, context switching neither requires nor forbids a transition from kernel mode to user mode.

We suggest you structure your kernel so that there is only one piece of code which implements context switching. Our suggestion is based on reading student kernels and observing that multiple context-switch code paths typically indicate multiple conflicting partial understandings of context switch. This often means that each one contains a bug. As discussed in the "Yield" lecture, if a single function performs a context switch, then the point of execution for every non-runnable thread is inside that function and the instruction pointer need not be explicitly saved. We also suggest that you set things up so there is only "one way back to user space" (rather than having the scheduler or context switcher invoke distinct code paths to "clean up" after `fork()`, `thread_fork()`, or `exec()`). If you can't manage one path, you should be able to keep the count down to two.

It is very unwise to attempt to solve a scheduling or context-switching problem by having kernel code invoke the `INT` instruction to get from one place to another. This approach is very expensive and structurally counterproductive.

When you are designing your context switch infrastructure, it will probably be helpful to enumerate the situations in which it might be invoked (hint: *not* just by the timer interrupt handler).

Before a thread runs for the first time, meaningful context will need to be placed on its kernel stack. A useful tool to set a thread running for the first time is the `IRET` instruction. It is capable of changing the code and stack segments, stack pointer, instruction pointer, and `EFLAGS` register all in one step. Please see page 153 of `intel-sys.pdf` for a diagram of what the `IRET` instruction expects on the stack.

5.2 Scheduling

A simple round robin-scheduler is sufficient for the 410 kernel. The running time of the scheduler should not depend on the number of threads currently in the various queues in the kernel. In particular, there are system calls that alter the order in which threads run. These calls should not cause the scheduler to run in anything other than constant expected time (but see Section 9.1.1 on "Encapsulation" below).

You should avoid a fixed limit on the number of tasks or threads. In particular, if we were to run your kernel on a machine with more memory (and/or a larger `USER_MEM_START` value), it should be able to support more tasks and threads. Also, your kernel should respond gracefully to running out of memory. System calls which would require more memory to execute should receive error return codes. In the other direction, it is considered legitimate for a Unix kernel to kill a process any time it is unable to grow its stack (optional challenge: can you do better than

that?).

5.3 Scheduling, Sleeping, and Synchronization

While designing your kernel, you may find yourself tempted to employ various "tricky" techniques to avoid actually putting a thread to sleep. Approaches to avoid:

Saved by the bell Instead of actually putting a thread to sleep, busy-wait until the clock interrupt arrives and invokes your context-switch code for you

To run or not to run? Construct a deceptive "scheduling queue" containing not only runnable threads but also threads asleep for fixed time periods and threads blocked indefinitely, thus requiring the thread scheduler to hunt among decoys for the occasional genuinely runnable thread.

Yield loop Instead of arranging to be awakened when the world is ready for you, repeatedly "sleep for a while" by putting yourself at the end of the run queue. Please note that, as we discussed in class, this approach is *guaranteed* to never sleep the right amount of time (too short for N-1 iterations and then too long the last time).

Not now, maybe later? Allow kernel code to be interrupted by devices, but forbid actual context switching in kernel mode.

Of course, you may *temporarily* employ "quick hacks" during the course of development, but please be aware that each "trick" on the list above maps directly to having avoided understanding an important conceptual issue, and this will influence your grade accordingly.

In general, adhere to these principles:

- A thread should not run in kernel mode when it is hopeless (it should stop running when it's time to stop running),
- A sleeping thread should not run before it can (probably) do productive work, and
- A sleeping thread should begin running, or at least be marked runnable, *as soon as* it can once again do productive work.

6 System Calls

6.1 The System Call Interface

The system call interface is the part of the kernel most exposed to user code. User code will make requests of the kernel by issuing a software interrupt using the `INT` instruction. Therefore, you will need to install one or more IDT entries to handle system calls.

The system call boundary protocol (calling convention) will be the same for Project 3 as it was for Project 2. Interrupt numbers are defined in `410user/lib/inc/syscall_int.h`. Please

make sure your user-space stub library is organized into files according to the guidelines specified in the "Pebbles Kernel Specification" document. However, inside the kernel, there is no need for each system call to contribute its own C source file and matching header file. Please lay out your system calls according to some sensible standard (something between "two files per system call" and "one monster syscall.c").

6.2 Validation

Your kernel must verify all arguments passed to system calls, and should return an integer error code less than zero if any arguments are invalid. The kernel *may not* kill a user thread that passes bad (or inconvenient) arguments to a system call, and it *absolutely may not* crash.

The kernel must verify, using its virtual memory housekeeping information, that every pointer is valid before it is used. For example, arguments to `exec()` are passed as a null terminated array of C-style null terminated strings. Each byte of each string must be checked to make sure that it lies in a valid region of memory. The kernel should also ensure that it isn't "tricked" into performing illegal memory operations for the user as a result of bad system call arguments.

6.3 Specific System Calls

You will implement the system calls described in the *Pebbles Kernel Specification* document, which has been updated as a result of comments received during Project 2.

Though this should go without saying, kernels missing system calls are unlikely to meet with strong approval from the course staff. However, as part of our tireless quest to calibrate the difficulty of the projects, this semester you are **not required** to implement `getchar()` and `task_exit()` for Project 3. You will want to think carefully about the issues associated with these system calls, however, as there is some chance you will be asked to implement them in Project 4.

While implementing `readline()`, please try to ensure that you do *not* context switch to the invoking thread every time a keyboard interrupt makes a new character available (unless that character would cause the `readline()` operation to complete and un-block the thread). We realize that this is the opposite of what we asked you to do for Project 1. To see why this makes sense, observe that if your kernel paged out to disk then a thread's user-space memory buffer could be paged out. Paging it in as each character in a long line typed by a slow person arrived would be wasted effort. Of course, since the shell depends on it, it will be more important for you to have a working `readline()` than an ideal one.

Please don't build a "terminal irony" into your `exit()` system call: it is *not acceptable* for `exit()` to "fail" because your kernel is out of memory...

7 Building and Loading User Programs

7.1 Building User Programs

User programs to be run on the 410 kernel should conform to the following requirements. They should be ELF formatted binaries such that the only sections that must be loaded by the kernel are the .text, .rodata, .data, and .bss sections (C++ programs, which have additional sections for constructors which run before main() and destructors which run after main(), are unlikely to work).

Programs may be linked against the 410-provided user-space library, and *must not* be linked against the standard C library provided on the host system. They should be linked statically, with the .text section beginning at the lowest address in the user address space. The entry point for all user programs should be the `_main()` function found in `410user/tests/crt0.c`.

7.2 Loading User Programs

The 410 kernel must read program data from a file, and load the data into a task's address space. Due to the absence of a file system, user programs will be loaded from large arrays compiled into a "RAM disk" image included in the kernel executable binary.

We have provided you with a utility, `410user/exec2obj`, which takes a list of file names of Pebbles executables and builds a single file, `user_apps.c`, containing one large character array per executable. It also contains a table of contents the format of which is described in `410kern/lib/inc/exec2obj.h`. When a program is executed, your loader will extract the ELF header and various parts of the executable from the relevant character array and build a runnable memory image of the program.

Later in the semester, there may be an opportunity to write a file system for the 410 kernel. To facilitate an easy switch from `exec2obj` to a file system, please use the `getbytes()` skeleton found in `kern/loader.c`: it provides a crude abstraction which can be implemented on top of either `user_apps.c` or a real file system.

Support code has also been provided in `kern/loader.c` to extract the important information from an ELF-formatted binary. `elf_check_header()` will verify that a specified file is an ELF binary, and `elf_load_helper()` will fill in the fields of a `struct se` ("simplified ELF") for you. Once you have been told the desired memory layout for an executable file, you are responsible for using `getbytes()` to transfer each executable file section to an appropriately organized memory region. You should zero out areas, if any, between the end of one region and the start of the next. The bss region should begin immediately after the end of the read/write data region, and the heap should begin on a page boundary.

Note: the .text and .rodata (read-only data) sections of the executable must be loaded into memory which the task's threads cannot modify.

8 The Programming Environment

8.1 Kernel Programming

The support libraries for the kernel include a simple C library, a list based dynamic memory allocator, functions for initializing the processor data structures with default values, and functions for manipulating processor data structures.

8.1.1 A Simple C Library

This is simply a list of the most common library functions that are provided. For details on using these functions please see the appropriate `man` pages. Other functions are provided that are not listed here. Please see the appropriate header files for a full listing of the provided functions.

Some functions typically found in a C I/O library are provided by `libstdio.a`. The header file for these functions is `410kern/lib/inc/stdio.h`.

- `int putchar(int c)`
- `int puts(const char *str)`
- `int printf(const char *format, ...)`
- `int sprintf(char *dest, const char *format, ...)`
- `int snprintf(char *dest, int size, const char *formant, ...)`
- `int sscanf(const char *str, const char *format, ...)`
- `void lprintf_kern( const char *format, ...)`

Some functions typically found in a C standard library are provided by `410kern/lib/libstdlib.a`. The header files for these functions, in `410kern/lib/inc`, are `stdlib.h`, `assert.h`, `malloc.h`, and `ctype.h`.

- `int atoi(const char *str)`
- `long atol(const char *str)`
- `long strtol(const char *in, const char **out, int base)`
- `unsigned long strtoul(const char *in, const char **out, int base)`
- `void *malloc(size_t size)`
- `void *calloc(size_t nelt, size_t eltsize)`
- `void *realloc(void *buf, size_t new_size)`

- `void free(void *buf)`
- `void smemalign(size_t alignment, size_t size)`
- `void sfree(void *buf, size_t size)`
- `void panic(const char *format, ...)`
- `void assert(int expression)`

The functions `smemalign()` and `sfree()` manage aligned blocks of memory. That is, if `alignment` is 8, the block of memory will be aligned on an 8-byte boundary. A block of memory allocated with smemalign *must* be freed with `sfree()`, which requires the `size` parameter. Therefore, you must keep track of the size of the block of memory you allocated. This interface is useful for allocating things like page tables, which must be aligned on a page boundary. By volunteering to remember the size, you free the storage allocator from scattering block headers or footers throughout memory, which would preclude it from allocating consecutive pages. `sfree(void* p, int size)` frees a block of memory. This block *must* have been allocated by `smemalign()` and it must be of the specified size. Note that these memory allocation facilities operate *only* on memory inside the kernel virtual address range. Of course, functions with similar names appear in user-space libraries; those functions *never* operate on kernel virtual memory.

Also, note that what we actually provide you with, as in Project 2, are non-thread-safe (non-reentrant) versions of the memory-management primitives (`malloc()` and friends), with underlined names (e.g., `_malloc()`). Once your kernel supports preemptive context switching, you will need to provide thread-safe wrapper routines based on whatever locking and synchronization primitives you design and implement. But it probably makes sense to get started with simple "pass-through" wrappers.

Some functions typically found in a C string library are provided by `410kern/lib/libstring.a`. The header file for these functions is `410kern/lib/inc/string.h`.

- `int strlen(const char *s)`
- `char *strcpy(char *dest, char *src)`
- `char *strncpy(char *dest, char *src, int n)`
- `char *strdup(const char *s)`
- `char *strcat(char *dest, const char *src)`
- `char *strncat(char *dest, const char *src, int n)`
- `int strcmp(const char *a, const char *b)`
- `int strncmp(const char *a, const char *b, int n)`
- `void *memmove(void *to, const void *from, unsigned int n)`

- `void *memset(void *to, int ch, unsigned int n)`
- `void *memcpy(void *to, const void *from, unsigned int n)`

8.1.2 Processor Utility Functions

These functions, declared in `410kern/lib/inc/x86/proc_reg.h`, access and modify processor registers and data structures. Descriptions of these functions can be found elsewhere in this document.

- `void disable_interrupts()`
- `void enable_interrupts()`
- `void set_cr3(void *)`
- `void set_cr3_nodebug(void *)`
- `void set_esp0(void *)`
- `void *get_esp0()`
- `void *sidt()`

8.1.3 Build Infrastructure

The provided `Makefile` takes care of many of the details of compiling and linking the kernel. It is important, however, to understand how it works. Once you unpack the Project 3 tarball, you should read `README` and `config.mk` and fill in the latter according to the directions.

It is also important that you pay attention to the directory structure. In particular, files in `410kern` and `410user` should be treated as read-only. While you might occasionally need to temporarily modify or instrument library routines for debugging purposes, the update process will overwrite any such changes. When we grade your kernel we will use the "supported" versions of those files, so changes you make *will* be wiped out.

The `Makefile` is configured to allow you to choose when updates take effect in order to provide you with debugging flexibility. However, this means that it will be your responsibility to incorporate support-code changes early enough that you can debug any issues before the project deadline. Basically, if `make` starts beeping, don't just ignore it!

8.2 User Programming

User programs which run on your kernel will have access to the same C library which was available in Project 2. However, the console output functions will not work until your kernel implements the `print()` system call.

8.3 Simulator versus Reality

While most executions of your kernel will be in the `simics` environment, be sure to consider how your kernel would behave if it were executed on real hardware. In particular, `MAGIC_BREAK`, `lprintf_kern()`, and `SIM_halt()` all have no effect if your code isn't running inside `simics`. If your kernel would fail spectacularly in such a situation, your grader will probably notice and be less impressed than you might wish.

While this is entirely optional, if you wish to try booting your kernel in a non-`simics` environment, you have several options:

- A commercial emulation environment such as `VMware` or `Virtual PC`
- Other PC hardware simulators (`Bochs`, `QEMU`, ...)
- Real hardware. If you don't have access to a rebootable PC with a PS/2 keyboard, you can try Dr. Eckhardt's office hours, as he has an easily-rebooted PC. To try it on your own, merely use the Unix `dd` command to transfer your `bootfd.img` file onto a blank, freshly-formatted floppy, check to make sure that the floppy drive is in your BIOS's boot order, and try a reboot.

9 Hints on Implementing a Kernel

In this section you will find a variety of suggestions and exhortations. We certainly do not expect each kernel to follow every suggestion contained herein, but you will probably find at least one of them attractive enough to implement it, at which point we expect you will judge it to have been a worthwhile investment of your time.

9.1 Code Organization

- You may wish to invest in the creation of a trace facility. Instead of lprintf() calls scattered at random through your code, you may wish to set up an infrastructure which allows you to enable and disable tracing of a whole component at once (e.g., the scheduler) and/or allow you to adjust a setting to increase or decrease the granularity of message logging.[1]
- Some eventualities are genuinely fatal in the sense that there is no way to continue operation of the kernel. If, for example, you happened to notice that one thread had overflowed its kernel stack onto the kernel stack of another thread, there would be no way to recover a correct execution state for that thread, nor to free up its resources. In such a situation the kernel is broken, and your job is no longer to arrange things like `return(-1)`, but instead to stop execution as fast as possible before wiping out data which could be used to find and fix the bug in question. You will need to use your judgement to classify situations in to

[1] While it is not rare for a kernel to log messages using impolite linguistic constructs, it *is* rare for this to improve the kernel's reception with course staff.

recoverable ones, which you should detect and recover from, and unrecoverable situations (such as data structure consistency failures), for which you should *not* write half-hearted sort-of-cleanup code.

These considerations may suggest that you make use of the `assert()` macro facility provided by `410user/lib/inc/assert.h` and `410kern/lib/inc/assert.h`.

- Avoid common coding mistakes. Be aware that `gcc` will not warn about possible unwanted assignments in `if`, and `while` statements. Also, note the difference between `!foo->bar`, and `!(foo->bar)`. Practicing Pair Programming can help avoid these kinds of mistakes.

9.1.1 Encapsulation

Instead of typing linked-list traversal code 100 times throughout your kernel, thus firmly and eternally committing yourselves to a linear-time data structure, you should attempt to encapsulate. Don't think of a linked list of threads; think of sets or groups of threads: live, runnable, etc.

Likewise, don't write a 2,000-line page fault handler. Instead of ignoring the semantic properties shared by pages within a region, use those properties to your advantage. Write smaller page-fault handlers which encapsulate the knowledge necessary to handle *some* page faults. You will probably find that your code is smaller, cleaner, and easier to debug.

Encapsulation can allow you to defer tricky code. Instead of implementing the "best" data structure for a given situation, you may temporarily hide a lower-quality data structure behind an interface designed to accomodate the better data structure. Once your kernel is stable, you can go back and "upgrade" your data structures. While we will not greet a chock-full-of-linked-lists kernel or a wall-of-small-arrays kernel with cries of joy, and data structure design is an important part of this exercise, achieving a complete, solid implementation is critical.

9.1.2 Synchronization

If you find yourself needing something sort of like a condition variable, *don't* throw away the modes of thought you learned in Project 2. Instead, use what you learned as an inspiration to design and implement an appropriate similar abstraction inside your kernel.

Likewise, you have multiple options for ensuring that a given sequence of kernel code is not interrupted by a conflicting sequence. One approach would be to litter your code with `disable_interrupts()`, but there are at least two other approaches. Regardless of what you settle on, you probably want to consider which "flavor" of mutex you need... you might need something like a counting mutex, or you might need to have "regular" and "special" mutexes.

One way to improve your synchronization design is to pretend you're designing your kernel to run on a multi-processor machine. This will make it clear that interrupt-disabling genuinely addresses only a small fraction of the synchronization problems a kernel faces.

9.1.3 Method tables

You *can* practice modularity and interface-based design in a language without objects. In C this is typically done via structures containing function-pointer fields. Here is a brief pseudo-code summary of one basic approach (other approaches are valid too!):

```
struct device_ops {
  void (*putchar)(void *, char);
  int (*getchar)(void *);
};

struct device_ops serial_ops = {
  serial_putchar, serial_getchar
};

struct device_ops pipe_ops = {
  pipe_putchar, pipe_getchar
};

struct device_object {
  struct device_ops *opsp;
  void *instance_data;
};

void putchar(struct device_object *dp, char c)
{
    (dp->opsp->putchar)(dp->instance_data, c);
}

void init(void)
{
  struct device_object *d1, *d2;

  d1 = new_pipe_object();
  d2 = new_serial_object();

  putchar(d1, '1'); /* pipe_putchar(d1->instance_data, '1'); */
  putchar(d2, '2'); /* serial_putchar(d2->instance_data, '2'); */
}
```

9.1.4 Embedded Traversal Fields

Imagine a component is designed around a linked list. It may seem natural to re-invent the Lisp[2] "cons cell":

```
struct listitem {
  struct listitem *next;
  void *item_itself;
}
```

The problem with this approach is that you are likely to call `malloc()` twice as often as you should—once for each item, and once for the list-item structure. Since `malloc()` can be fairly slow, this is not the best idea, even if you are comfortable dealing with odd outcomes (what if you can allocate the list item but not the data item, or the other way around?).

Often a better idea is to embed the traversal structure inside the data item:

```
struct item_itself {
  struct item_itself *next;
  int field1;
  char field2;
}
```

This cuts your `malloc()` load in half. Also, once you understand C well enough, it is possible to build on this approach so you can write code (or macros) which will traverse a list of threads or a list of devices.

Isn't this an encapsulation violation? It depends...if "everybody knows" that your component does traversal one way, that is bad. If only your component's exported methods know the traversal rules, this can be a very useful approach.

9.1.5 Avoiding the malloc()-blob Monster

There is one particular place in your kernel where the cons-cell approach (aka "little malloc() blobs") is particularly inappropriate. Before you commit your kernel to containing thousands of them, it is probably wise to compute their true size cost. This will probably involve recalling some `malloc()` internals from 15-213, computing roughly how many blobs you intend to create, and considering the size of the memory pool they will be drawn from.

9.1.6 List Traversal Macros

You may find yourself wishing for a way for a PCB to be on multiple lists at the same time but not relish the thought of writing several essentially identical list traversal routines. Other languages have generic-package facilities, but C does not. However, it is possible to employ the C preprocessor to automatically generate a family of similar functions. If you wish to pursue this approach, you will find a template available in `vq_challenge/variable_queue.h`.

[2] or, for you young whippersnappers, ML

9.1.7 Accessing User Memory

System calls involve substantial communication of data between user memory and kernel memory. Of course, these transfers must be carefully designed to avoid giving user code the ability to crash the kernel.

You will discover that there are many potential hazards. Instead of checking manually for each hazard every time you read or write some piece of user memory, you may find it useful to encapsulate this decision-making in some utility routines. If you think about how and when these routines are used, you may find that you can piggyback these hazard checks onto another operation, resulting in a pleasant package.

9.2 Task/Thread Initialization

- **Task/thread IDs** - Each thread must have a unique integer thread ID. Since an integer allows for over two billion threads to be created before overflowing its range, sequential numbers may simply be assigned to each thread created without worrying about the possibility of wrap-around – though real operating systems do worry about this. The thread ID must be a small, unique integer, not a pointer.

  It is desirable for the data structure used to map a thread ID to its thread control block to be efficient in space and time. One possible implementation would be a hash table indexed by thread ID. Time pressure may argue for simpler data structures, which may be employed if they are appropriately encapsulated.

- **`thread_fork`** - On a `fork()`, a new task ID is assigned, and kernel state is allocated for the new schedulable entity, which will use the resources of the invoking thread's task. The new thread will have identical contents in all user-visible registers except `%eax`, which will contain zero in the new thread and the new thread's ID in the old thread.

- **`fork()`** - On a `fork()`, a new task ID is assigned, the user context from the running parent is copied to the child, and a deep copy is made of the parent's address space. Since the CPU relies on the page directory/page table infrastructure to access memory via virtual addresses, both the source and destination of a copy must be mapped at the same time. This does not mean that it is necessary to map both entire address spaces at the same time, however. The copy may be done piece-meal, since the address spaces are already naturally divided into pages.

  Any time you insert a "fake" mapping into a task's page table you should realize that this mapping will probably make its way into the TLB and may persist there for an indefinite period of time *after you remove the mapping from the page table*. See Section 2.3.3 for one approach to solving this problem.

- **`exec()`** - On an `exec()`, the stack area for the new program must be initialized. The stack for a new program begins with only one page of memory allocated. It is traditional for the "command line" argument vector to occupy memory above the run-time stack.

9.3 Thread Exit

You will find that there are subtle issues associated with enabling a thread or task to exit cleanly, so it's a good idea to pseudo-code how this will work and what else it will interact with.

As you will discover during your design (or else during your implementation), thread exit will involve freeing a variety of resources. Furthermore, the efficiency of the system is increased if you can free resources quickly so they are available for other threads to use. Optional challenge: how small can you make "zombie threads" in your system? Alternatively, how many of the thread's resources can the thread itself free, as opposed to relying on outside help?

9.4 Kernel Initialization

Please consider going through these steps in the `kernel_main()` function.

- Initialize the IDT entries for each interrupt that must be handled.
- Clear the console. The initialization routines will leave a mess.
- Build a structure to keep track of which physical frames are not currently allocated.
- Build the initial page directory and page tables. Direct map the kernel's virtual memory space.
- Create and load the idle task. For grading purposes, you may assume that the "file system" will contain a (single-threaded) executable called `idle` which you may run when no other thread is runnable. Or you may choose to hand-craft an idle program without reference to an executable file.
- Create and load the `init` task. For grading purposes, assume that the "file system" will contain an executable called `init` which will run the shell (or whatever grading harness we decide to run). During your development, `init` should probably `fork()` a child that `exec()`s the program in `410user/progs/shell.c`. It is traditional for `init` to loop on `wait()` in order to garbage-collect orphaned zombie tasks; it is also traditional for it to react sensibly if the shell exits or is killed.
- Set the first thread running.

N.B. Suggesting that `kernel_main` implements these functions does *not* imply that it must do so via straight-line code with no helper functions.

9.5 Memory Management Operations

You will probably find that the x86 virtual memory hardware is complex enough to suggest multiple ways of implementing critical operations such as address-space copying. Your design should probably assume that invalidating a particular address translation is much cheaper than

invalidating many or all address translations *en masse*, which in turn is dramatacally cheaper than disabling paging. Meanwhile, the complexity of an operation should be balanced against its frequency. Overall, a plan which would disable and enable paging repeatedly in an inner loop probably represents a serious and unnecessary performance problem (the kind your grader dislikes).

10 Debugging

10.1 Requests for Help

Please do not ask for help from the course staff with a message like this:

> I'm getting the default trap handler telling me I have a general protection fault. What's wrong?

or

> I installed my illegal instruction handler and now it's telling me I've executed an illegal instruction. What's wrong?

An important part of this class is developing your debugging skills. In other words, when you complete this class you should be able to debug problems which you previously would not have been able to handle.

Thus, when faced with a problem, you need to invest some time in figuring out a way to characterize it and close in on it so you can observe it in the actual act of destruction. Your reflex when running into a strange new problem should be to start thinking, not to start off by asking for help.

Having said that, if a reasonable amount of time has been spent trying to solve a problem and no progress has been made, do not hesitate to ask a question. But please be prepared with a list of details and an explanation of what you have tried and ruled out so far.

10.2 Debugging Strategy

In general, when confronted by a mysterious problem, you should begin with a "story" of what you *expect* to be happening and measure the system you're debugging to see where its behavior diverges from your expectations.

To do this your story must be fairly detailed. For example, you should have a fairly good mental model of the assembly code generated from a given line of C code. To understand why "a variable has the wrong value" you need to know how the variable is initialized, where its value is stored at various times, and how it moves from one location to another. If you're confused about this, it is probably good for you to spend some time with `gcc -S`.

Once your "story" is fleshed out, you will need to measure the system at increasing levels of detail to determine the point of divergence. You will find yourself spending some time thinking

about how to pin your code down to observe whether or not a particular misbehavior is happening. You may need to write some code to periodically test data-structure consistency, artificially cause a library routine to fail to observe how your main code responds, log actions taken by your code and write a log-analyzer perl script, etc.

Please note that the kernel memory allocator is very similar to the allocator written by 15-213 students in the sense that when the allocator reports an "internal" consistency failure, this is *overwhelmingly* likely to mean that the user of some memory overflowed it and corrupted the allocator's meta-data. In other words, even though the error is *reported* by `lmm_free()`, it is almost certainly not an error *in* `lmm_free()`.

10.3 Kernel Debugging Tools

There are a number of ways to go about finding bugs in kernel code. The most direct way for this project is to use the Simics symbolic debugger. Information about how to use the Simics debugger can be found in the documentation on the course website (`410/simics/doc`), and by issuing the `help` command at the simics prompt.

Also available is the `MAGIC_BREAK` macro defined in `410kern/lib/inc/kerndebug.h`. Placing this macro in code will cause the simulation to stop temporarily so that the debugger may be used.

The function call `lprintf_kern()` may also be used to output debugging messages to the simics console, and to the file kernel.log. The prototype for `lprintf_kern()` can be found in `410kern/lib/inc/stdio.h`.

10.4 User Task Debugging

Symbolic debugging user tasks can be useful in the course of finding bugs in kernel code. The `MAGIC_BREAK` macro is also available to user threads by `#include`ing the `410user/inc/magic_break.h` header file.

The function call `lprintf()` may be used to output debugging messages from user programs. Its prototype is in `410user/lib/inc/stdio.h`.

Symbolic debugging of user programs involves some set-up. Simics can keep track of many different virtual memory spaces and symbol tables by associating the address of the page directory with the file name of the program.

Simics must switch to the appropriate symbol table for the current address space as soon as a new value is placed in `%cr3`. For this to work, you must do three things.

1. When a new program is loaded, register its symbol table with Simics with a call to `SIM_register_user_proc()`, defined in `410kern/lib/inc/kerndebug.h`.

2. When a program exits, please make a call to `SIM_unregister_user_proc()` defined in the same file.

3. Every time you switch user address spaces via `set_cr3()`, the Simics magic-break instruction will be used to tell the Simics debugger to switch symbol tables. If you believe you must change the value of `%cr3` in assembly language, simply copy the relevant instructions from the `set_cr3()` we provide.

If you do not wish to enable debugging of user threads, simply do not register threads with Simics, and use the macro `set_cr3_nodebug()` instead of `set_cr3()`. But we recommend you do this, because it is easy and can significantly improve your debugging experience.

11 Checkpoints

The kernel project is a large project spanning several weeks. Over the course of the project the course staff would like to review the progress being made. For that reason, there are checkpoints that will be strictly enforced. The checkpoints exist so that important feedback can be provided, and so should be taken very seriously.

Don't be complacent about missing "just one" checkpoint. Catching up is *hard*: if you're "merely" a week behind, that means you need to work twice as hard for an entire week in order to catch up...

As each checkpoint arrives, we will create a corresponding directory for you to hand in your code. We will generally not grade checkpoint code submissions, but they have proven to be a valuable reference in past semesters. For Checkpoint One you will place your code in `p3ck1`, etc.

11.1 Checkpoint One

For Checkpoint One, you should have a user task (with one thread) that can call `gettid()`. This sounds simple, but it involves quite a bit of work which you will probably need to plan carefully. Strive to document most of this code *before* you write it.

1. Draft fault handlers – you may end up re-writing these (optional challenge: can you *generate* these extremely repetitive assembly functions with C preprocessor macros?).
2. Draft system call handler for gettid()
3. Draft pcb/tcb structures – think about the task/thread split early, in terms of how thread creation and exit will work.
4. Basic virtual memory system
    - Dummy up a physical-frame allocator (allocate-only, no free)
    - Spec and write page-directory/page-table manipulation routines
    - Spec and write draft "address space manager" (probably centered around a "region list" with a list of per-region-type "methods").

5. Rough first-program loader

    - Set up a task.
    - Set up a thread.
    - Initialize various memory regions.
    - Decide on register values.
    - Craft kernel stack contents.
    - Set various control registers.

    You may find that there are many ways for `IRET` to go awry. If so, it might be useful to see how many of the possibilities can be eliminated if you temporary re-work your code so a `RET` instruction would work instead.

11.2 Checkpoint Two

For Checkpoint Two, it should be possible for two threads (probably belonging to two different tasks, which may be hand-loaded) to context-switch back and forth, and *either* `fork()` or `exec()` (your choice) should work.

Reaching Checkpoint Two will be much easier if your code for Checkpoint One was carefully designed and modular.

11.3 Checkpoint Three

For Checkpoint Three, roughly the third week of the project, you should have approximately half of the system calls and exception handlers done. Note that you should count by lines of code rather than by number of entry points–some are easier than others. For example, the page fault handler, which will require noticeable thought and implementation, is probably 50% of the exception-handler implementation by itself.

11.4 Week Four

While we will probably not collect any further checkpoints, by the fourth week of the project you should *really* have at least started all of the system calls.

11.5 Week Five

You should aim to have all of your code written, by the end of this week. You will have plenty of bugs and rewriting to keep you busy in the last week (really!). Remember to focus your effort on getting the core system calls working solidly–you won't pass the test suite if we can't boot your kernel, launch the shell, and have it run commands.

12 Strategy Suggestions

Before we begin, we'd like to recommend to you the sage words of 15-410 student Anand Thakker.

> Each time you sit down to write code, pause to remind yourself that you love yourself, and that you can demonstrate this love by writing good code for yourself.

Here are further suggestions:

- You will probably end up throwing away and re-writing some non-trivial piece of code. Try to leave time for that.
- As you progess through the project, you should acquire the ability to draw detailed pictures of every part of the system. For example, if you come see the course staff about a problem, we may well ask you to draw a picture of the thread stack at the point where the problem is encountered. We may also ask you to draw pictures of a task's page tables, virtual address space, etc. In general, at several points in the project you will need to draw pictures to get things right, so get in the habit of drawing pictures when you're in trouble.
- Try to schedule *at least* three chunks of time every week, each *at least* two hours long, to meet with your partner. You will need to talk about each other's code, update each other on bug status, assign or re-assign responsibility for code blocks or particular bugs, etc.
- **You should really be familiar with your partner's code. You will need to answer exam questions based on it.**
- Since a code merge takes time at least proportional to the inter-merge time, you should probably merge frequently. If you work independently for five weeks, it will probably take you a week to merge your code, at which point you will have no time to debug it.
- Merging can be much easier if you use branches. For example, one partner can implement `fork()` while the other implements `readline()`. Each can work on an indepenent branch, committing every hour or so, rolling back if necessary, adding test code, etc. Once the `readline()` implementor is done, she can merge the most recent revision on her branch against the trunk. Her partner can then merge from the trunk into his working area. This will probably result in a non-functional kernel, but observe that the first partner's branch, the trunk, and his branch should all be ok and kernels can be checked out and built from all of them.
- If you are paranoid, there is no reason why you shouldn't occasionally snapshot your repository to a safe location in case your source control system goes haywire on you.

13 Plan of Attack

A recommended plan of attack has been established. While you may not choose to do everything in this order, it may provide you with some guidance. Hopefully, this will give some ideas about how to start.

1. Read this handout and gain an understanding of the assignment. First, understand the hardware, and then the operations that need to be implemented. Spend time becoming familiar with all of the ways the kernel could be invoked. What happens on a transition from user mode to kernel mode? What happens on a transition from kernel mode to user mode?

2. Review the kernel-specification document. Generate a list of the underlying modules each system call and exception handler will rely on, and also a list of hazards or challenges for each kernel entry point.

3. Write pseudocode for the system calls as well as the interrupt handlers, paging system, and context switcher. Start by writing down how all of these pieces fit together. Next, increase the level of detail and think about how the pieces break down into functions. Then, write detailed pseudocode.

4. Based on the above step, rough out the Task Control Block and Thread Control Blocks. What should go in each?

5. Write functions for the manipulation of virtual address spaces. Direct map the kernel's virtual memory space. Keep track of free physical frames. Figure out to allocate and deallocate frames outside the kernel virtual space so they can be assigned to tasks (optional challenge: can you do this in a way which doesn't consume more kernel virtual space for management overhead as the size of physical memory grows larger?).

6. Now that there is an initial page directory, it is possible to enable paging. Do so, then write the loader. Create a PCB for the idle task. Load and run the idle task.

7. Implement the `gettid()` system call. Once this is working, the system call interface is functioning correctly. Congratulations! You have reached checkpoint one.

8. Write the timer interrupt handler. For now, simply verify that the IDT entry is installed correctly, and that the interrupt handler is running.

9. Write functions for scheduling and context switching. Load a second task. Have the timer interrupt handler context switch between the first and second task.

10. Implement `fork()` or `exec()`. At this point you will have reached checkpoint two.

11. Implement `exec()` or `fork()`, whichever one you skipped before. You can test how they work together by having the init task spawn a third user task. From this point forward your old code to hand-load a task should be invoked only once or twice, to load `init` and possibly `idle`.

12. Implement the `halt()` system call. Why not? It's easy enough.

13. Take a few moments to get user-space symbolic debugging enabled–it's better to do it now than to decide in the last few days that you really need it (right away!) only to run into some bug getting it to work.

14. Implement the `new_pages()` system call, and test it with the user-mode `malloc()`.

15. Write a page fault handler that frames new pages on legal accesses to the automatic-stack region, and prints debugging information on bad accesses (you are probably not ready to kill threads at this point).

16. Integrate the keyboard interrupt handler and the console driver into the kernel. Install an entry for the keyboard in the IDT.

17. Implement the a rough version of the `readline()` system call. If you haven't confronted thread sleep/wakeup yet, you may want to dodge the issue long enough to get the shell running, but you can't put it off forever. Keep in mind that the keyboard interrupt handler and other `readline()` code may need to change during Project 4 to support the `getchar()` system call.

18. Implement the `print()`, `set_term_color()`, and `set_cursor_pos()` system calls.

19. Implement `wait()`, and `exit()`. Please take care that `exit()` is not grossly inefficient. At this point, the shell should run, and you can argue that you have written an operating system kernel. This is roughly checkpoint three.

20. Fill out the page fault handler. Threads should be killed on bad memory accesses.

21. Implement the `sleep()` system call. Recall that the round-robin scheduler should run in constant time. It is **bad** if your scheduler typically requires run-time proportional to the number of *non-runnable* threads. Hmm...

22. Implement the `yield()`, `deschedule()`, and `make_runnable()` system calls.

23. Implement `thread_fork()`, and test it using your thread library.

24. Go back and fill in any missing system calls. Anywhere you relied on a dummy implementation of some critical feature, replace the internals of the interface with something better.

25. Write many test cases for each system call. Try to break the kernel, since we will and you want to get there first...

26. At this point your `README.dox` should contain an accurate known-bug list.

27. You're done! Celebrate!