Project 3: Writing a Kernel From Scratch

15-410 Operating Systems

October 4, 2003

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

- Friday, October 3: Project 3 begins
- Friday, October 10: Checkpoint 1 due (standard turn-in procedure).
- Monday, October 20: Checkpoint 2 due (we will probably handle this via 5-minute mini-interviews).
- Friday, November 14: 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.

1.6 Grading

The primary criteria for grading are correctness, 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 processes), handles interesting corner cases correctly, and recovers gracefully from errors. 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. 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.

1.7 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 generate. Plan to keep to yourself disk image files, editor-generated backup files, log files, etc.

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 processes 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 processes 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 another processor data structure called the Global Descriptor Table(GDT). The GDT is where the segments are actually defined. The provided startup code sets up the 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 process is stored in `%cs`. The stack segment selector for the currently running process 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 process is started, user level code, stack, and data segment selectors need to be specified and loaded into the segment selector registers. When a user process 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 sections 2.1, 2.4, and 3.2 of `intel-sys.pdf` and the segmentation handout on the 15-410 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 process, so it is important to understand its format. The EFLAGS register is discussed in section 2.3 of intel-sys.pdf. 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 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 should arrance for the IF and IOPL_USER bits to be set. 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 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 process 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 process 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() will modify the initial kernel stack pointer. This function is defined in lib/x86/seg.c.

2.3.5 C interface

There are inline assembly macros defined in `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. The top level of the paging structure is called the page directory, while the second level consists of objects called page tables. The format of the entries in the page directory and page tables are very similar, however, their fields have slightly different meaning. 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 virtual 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 preset 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 only valid 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 0x0 to address 0xffffff, i.e., just under `USER_MEM_START` as defined by `lib/inc/x86/seg.h`), is reserved for the kernel. This kernel memory should appear as the bottom 16MB of each process'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 processes should not be able to read from or write to kernel memory, even though it is resident at the bottom of each user process' address space. In other words, like the "black hole" between the top of the heap and the bottom of the stack, there should also be a "black hole" from 0 to 0xffffff.

The rest of physical memory, i.e. from 0x1000000 up, should be used for frames. In `lib/inc/x86/seg.h` is a prototype for a function `int machine_phys_frames(void)`, provided by `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 `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 `lib/multiboot/baes_multiboot_main.c`. The support code in `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.

A memory map of the system is stored in a list of range descriptors that is pointed to by the information handed to the kernel by the multiboot initialization functions. The relevant structures are defined in `lib/inc/multiboot.h`. This information should be used when setting up the list of free physical frames, and in determining the amount of memory availible in the system.

4 Device Drivers and Interrupt Handlers

4.1 The Interrupt Descriptor Table

The Interrupt Descriptor Table(IDT) is a processor data structure that the processor reads when an interrupt is received. Interrupts are delivered to the processor by the Programmable Interrupt Controller(PIC) described below. The table contains a descriptor for each interrupt. A descriptor contains information about what should happen when an interrupt is received. Importantly, it contains the address of the handler for that interrupt. An interrupt handler is a function that performs a set of actions appropriate for the kind of interrupt that was issued. Once the processor locates the appropriate entry in the IDT, it saves information such that after the interrupt handler returns, it can resume what it was doing before the interrupt was issued. The PIC delivers interrupts to the processor over Interrupt Request(IRQ) lines. Note that IRQ numbers do not necessarily map to matching indeces into the IDT. PICs have the ability to map their IRQ lines to any entries in the IDT. For more information about interrupts, the IDT, and the PIC, please see chapter 5 of `intel-sys.pdf`.

4.2 Interrupts, Faults, and Exceptions

4.2.1 Hardware Interrupts

When a packet arrives at a network interface, the user presses a key on the keyboard or moves the mouse, or any other type of event occurs at a hardware device, that device needs a way of getting the attention of the kernel. One way is for the kernel to keep asking the device if it has something new to report (either periodically or continuously). This is called polled I/O. This is wasteful of CPU time. Modern kernels take advantage of hardware interrupts.

When a device wants to raise a hardware interrupt, it communicates this desire to one of two PICs by asserting some control signals on interrupt request lines. The PICs are responsible for serializing the interrupts (taking possibly concurrent interrupts and ordering them), and then communicating the interrupts to the processor through special control lines. The PICs tell the processor that a hardware interrupt has occurred, as well as which request line the interrupt occurred on so the processor knows how to handle the interrupt. There are some conventions as to what devices are associated with what interrupt request lines.

The PIC chip used in older IBM compatible PCs only has 8 IRQ lines. This proves to be limiting, so a second PIC is daisy chained off of the first one in more recent machines. When an interrupt is triggered on the second PIC, it in turn triggers an interrupt on the first PIC (on IRQ 2). The interrupt is then communicated to the processor.

4.2.2 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`), which will cause the processor to execute the nth handler in the IDT. In addition to hardware and software interrupt handlers, the IDT also contains information about exception handlers. Exceptions are conditions in the processor that are usually unintended and need to be addressed. Page faults, divide-by-zero, and segmentation faults are all types of exceptions.

4.2.3 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 relevent information(ie. the faulting address on a Page Fault if the user process will be killed by the kernel).

4.2.4 Configuring Interrupts

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 need to 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 disable interrupts on entering the interrupt handler, and so are also inappropriate as the 410 kernel is preemptable. The 410 kernel uses trap gates for all of its interrupts.

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 `lib/x86/seg.c`. The prototype can be found in `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 `lib/inc/x86/seg.h`.

4.2.5 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 page 153 of `intel-sys.pdf`.

This information is indeed enough to resume executing whatever code was running when the interrupt first arrived. However, in order to service the interrupt we need to execute some code; this code will clobber the values in the general purpose registers. When execution in normal code resumes, that code will expect to see the same values in the registers, as if no interrupt had ever occurred. So the first thing an interrupt handler must do is save all the general purpose registers, plus `%ebp`.

The easiest way to save these registers is to just save them on the stack. This is easily done with the `PUSHA` and `POPA` instructions (more information on these instructions can be found on pages 624 and 576 of `intel-isr.pdf`). To be sure that the registers are saved before anything else occurs, assembly wrappers for the interrupt handlers should be written. All that must be done in assembly is saving the registers, calling the C handler, then restoring the registers. To return from an interrupt, use the `IRET` instruction. It uses the information initially saved on the stack by the interrupt to return to the code that was executing when the interrupt occurred.

A final note about writing assembly: comment assembly code profusely. One comment per instruction is not a bad rule of thumb. Keep your assembly in separate files, and to export a symbol (like `asm_timer_wrapper`) so you can refer to it elsewhere, use the assembler directive `.globl`, like this:

```
.globl asm_timer_wrapper
```

4.2.6 Disabling Interrupts

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 `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.

The simulator is programmed to log how long interrupts are disabled. Note that the C macros referenced above issue the Simics magic instruction. For convenience, macros have also been written to replace the `CLI` and `STI` instructions. These macros are defined in `lib/inc/x86/cli_sti.asm.h`. This file *must* be `#include`d in each assembly file in which the `CLI` or `STI` instructions appear.

The log file `ints.log` is stored when `SIM_halt()` is called. This function should be used to implement the `halt()` system call. If interrupts are disabled for an especially long time, a note of it is made in the log file.

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 process is waiting for, it should be possible for your kernel to suspend the interrupted process and resume the waiting process so that returning from the interrupt activates the waiting process rather than the interrupted process.

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

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

4.3 Device Drivers

4.3.1 Communicating with Devices

There are two ways to communicate with a device in the x86 architecture. The first is to send bytes to an I/O port. The second is through memory-mapped I/O.

Most devices in the x86 architecture are accessed through I/O ports. These ports are controlled by special system hardware that has access to the data, address, and control lines on the processor. By using special hardware instructions to read and write from these ports, I/O ports can be used without infringing upon the normal address space of the kernel or user applications. This is because these special instructions tell the hardware that this memory reference is actually an I/O port, not a traditional memory location. For more information on I/O ports, consult chapter 10 of intel-arch.pdf. Both the timer and the keyboard use I/O ports. For convenience, an assortment of C wrapper functions is provided for reading and writing I/O ports. These are located in `lib/inc/x86/pio.h`.

There are also some devices which are accessed by reading and writing special addresses in traditional memory (memory-mapped I/O). This part of memory is part of the regular address space and therefore needs to be carefully managed. The console uses memory-mapped I/O (along with some I/O ports for things like cursor position).

4.3.2 Console Device Driver Specification

The contents of the console are controlled by a region of main memory (memory mapped I/O). Each character on the console is represented in this region by a byte pair. The first byte in this pair is simply the character itself. The second byte controls the foreground and background colors used to draw the character. These byte pairs are stored in row major order. For the console, there should be 25 rows of 80 characters each. The location of video memory, as well as the color codes used for the second byte of each pair, are defined in `lib/inc/video_defines.h`.

Writing a character to the console is as simple as writing a byte pair to video memory. To write the character 'M' as character seven of line four, you would do something like this:

```
*(CONSOLE_MEM_BASE+4*CONSOLE_WIDTH+7)='M';
*(CONSOLE_MEM_BASE+4*CONSOLE_WIDTH+7+1=console_color;
```

Where `CONSOLE_MEM_BASE` is the defined location of the console memory, `CONSOLE_WIDTH` is the width of the console in characters(80), and `console_color` is a variable containing the current foreground and background color.

It is also necessary to manipulate the cursor. The cursor is controlled by the Cathode Ray Tube Controller(CRTC). Communication with the CRTC is accomplished with a special pair of registers. The CRTC is one of those devices that is accessed through I/O ports. The two special registers are an index register and a data register. The index register tells the CRTC what function is to be performed, such as setting the cursor position. The data register then accepts a data value associated with the operation. The data register is only one byte long, and the offset of the cursor(which is measured in single bytes) is a two byte quantity, so setting the cursor position is done in two steps. The commands to send to the CRTC, as well as the location of the CRTC I/O ports, are defined in `lib/inc/video_defines.h`. To hide the cursor completely, simply set the cursor to an offset greater than the size of the console.

Now that communication with the console and CRTC is possible, a device driver for the console can be written. Here is the API for the driver:

- `char putbyte( char ch )` - Writes a single character to the console at the location of the cursor. Scrolls if necessary. Handles the special characters; a newline, a carriage return, or a backspace.

- `void putbytes( const char *s, int len )` - Prints a string `s` of length `len` starting at the current cursor position. If `len` is not a positive integer or `s` is NULL, the function has no effect.

- `void set_term_color( int color )` - Changes the foreground and background color of future characters printed on the console. If `color` is invalid, the function has no effect.

- `void get_term_color( int *color )` - Writes the current foreground and background color at the address specified by `color`.

- `void set_cursor( int row, int col )` - Sets the position of the cursor to the position `(row, col)`. If the cursor is hidden, a call to `set_cursor()` must not show the cursor.

- `void get_cursor( int *row, int *col )` - Writes the current position of the cursor into the arguments `row` and `col`.

- `void hide_cursor()` - Hides the cursor.

- `void show_cursor()` - Shows the cursor.
- `void clear_console()` - Clears the console.
- `void draw_char( int row, int col, int ch, int color )` - Writes the character `ch` to the console at position `(row,col)` with color `color`. Does not scroll or otherwise modify the console.

4.3.3 Timer Device Driver Specification

The timer device driver is important as it is used to trigger the 410 kernel's process scheduler. Timer interrupts must be handled quickly, or the timer will generate the next timer interrupt before the PIC has been reset, and that interrupt will be lost.

Communicating with the timer is done through I/O ports. These I/O ports are defined in timer.h. Also defined in `lib/inc/timer_defines.h` is the internal rate of the PC timer, 1193182 Hz. Fortunately, we can configure the timer give us interrupts at a fraction of that rate. For convenience, you should configure the timer to generate interrupts every 10 milliseconds.

To initialize the timer, first set its mode by sending `TIMER_SQUARE_WAVE` defined in `timer_defines.h` to `TIMER_MODE_IO_PORT` defined in the same file. The timer will then expect you to send it the number of timer cycles between interrupts. This rate is a two byte quantity, so first send it the least significant byte, then the most significant byte. These bytes should be sent to `TIMER_PERIOD_IO_PORT` defined in `timer_defines.h`.

When the timer interrupt occurs the processor consults the IDT to find out where the timer handler is. The index into the IDT for the timer is `TIMER_IDT_ENTRY`, defined in `timer_defines.h`. You will need to complete this entry for your timer handler to execute properly.

Aside from incrementing your tick counter, your timer interrupt handler should save and restore the general purpose registers. You also need to tell the PIC that you have processed the most recent interrupt that the PIC delivered. This is done by sending an `INT_CTL_DONE` to one of the PIC's I/O ports, `INT_CTL_REG`. These are defined in `lib/inc/interrupts.h`.

Note: You will be testing this on an instruction set simulator. Even though you are simulating an older processor on a relatively fast machine, Simics does not make an effort to exactly correlate the simulation to real wall clock time. Your timer may appear a little wobbly in the simulator, but it should be roughly accurate. If you have it set up properly, the timer will be exactly correct on real hardware.

4.3.4 Keyboard Device Driver Specification

Like the timer, the keyboard is also interrupt driven. However we are not only interested in the fact that a keyboard interrupt happened; each keyboard interrupt also has data that comes along with it. The information retrieved from the keyboard also needs to be processed in order to turn it into a stream of intelligible characters suitable for delivery to application processes. At a high level, the keyboard device driver provides a buffer that contains characters returned by the `readchar()` function.

One would think that reading keys from the keyboard would be as simple as receiving a simple character stream. Unfortunately it is not that easy. For one thing, both key presses and key releases are reported by the keyboard via interrupts. The other complicating factor is that the data reported by the keyboard is in a special format called scan codes. These codes need to be converted into normal ASCII characters. The support code provides a function that converts a scan code into an ASCII character. This function (called `process_scancode()`) is described in `lib/inc/keyhelp.h`.

The keyboard device driver has a very simple interface - it is just one function, `readchar()`.

- `char readchar()` - Returns the next character in the keyboard buffer. This function does not block if there are no characters in the keyboard buffer. `readchar()` returns the character in the buffer, or -1 if the keyboard buffer is currently empty.

4.3.5 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 process (optional challenge: support floating-point).

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

It is suggested that a context switch always take place at the same location. If a single function performs a context switch, then the point of execution for every process not currently running is inside that function. The instruction pointer need not be explicitly saved.

Before a process runs for the first time, meaningful context will need to be placed on its kernel stack. A useful tool to set a process 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.

Please note that in kernel mode, context switches may not be deferred until a later time.

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 processes currently in the various process queues in the kernel. In particular, there are system calls that alter the order in which processes run. These calls should not cause the scheduler to run in anything other than constant expected time (but see Section 9.1.1, “Encapsulation” below).

You should avoid a fixed limit on the number of processes. In particular, if we were to run your kernel on a machine with more memory, it should be able to support more processes. 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?).

6 System Calls

6.1 The System Call Interface

The system call interface is the part of the kernel most exposed to user processes. User processes 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 P3 as it was for P2. Interrupt numbers are defined in `lib/inc/syscall_int.h`.

6.2 System Call Specifications

6.2.1 Overview

The 410 kernel supports the basic process creation and control operations: `fork()`, `exec()`, `wait()`, and `exit()`. It also supports `sleep()` and `yield()` functions, terminal I/O functions, and a primitive threading facility.

6.2.2 Validation

Your 410 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 process that passes bad 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.

6.2.3 The System Calls

Implement the following system calls in your kernel. Though this should go without saying, kernels missing system calls are unlikely to meet with strong approval from the course staff. Except for `minclone()`, the prototypes listed here denote the interface exposed to user processes.

- `int fork()` - Creates a new child process as a copy of the calling process. On success, the process ID of the child process is returned to the parent, and zero is returned to the child. Note that `fork()` makes a deep copy of the parent's address space and gives it to the child.

- `int exec(char *execname, char **argvec)` - Replaces the currently running program in the calling process with the program stored in the file named `execname`. The argument `argvec` points to a vector of arguments to pass to the new program as its argument list. The new program receives these arguments as arguments to its `main()` function. The argument vector should be passed to the `exec()` system call formatted as a C style argument vector, but this should be validated. The argument vector should be passed to the new program's `main()` function as a C style argument vector, as the second argument. The first argument to the new program's `main()` function should be the length of the argument vector. Be sure to be familiar with how C style argument vectors are formatted. Reasonable limits may be placed on the number of arguments that a user program may pass to `exec()`, and the length of each argument. Be sure to do as much validation as possible before deallocating the old program's resources. On success, there is no return from this call in the calling program. If something goes wrong, an integer error code less than zero should be returned.

- `void exit(int status)` - Terminates execution of the calling process immediately, and saves the integer `status` for possible later collection by the parent through a call to `wait()`. In the case that the process' resources are not being shared, all resources used by the calling process are reclaimed, except for the saved exit status. When a process with children or a member of a thread group calls `exit()` or is aborted, the children or other threads should continue to run normally. Orphans may either be reparented to the `init` user process, which contains a while loop around `wait()`, or may be told that they no longer have a parent. Unix-like kernels use the former semantic, and also save the equivalent of the PCB until `wait()` is called on the exited process.

- `int wait(int *status_ptr)` - Collects the process ID and exit status returned by a child process of the calling process. If the calling process has no children, an integer error code less than zero is returned. Otherwise, if there are no exited children waiting to be collected, the calling process blocks until a child exits. The process ID of the child process is returned on success, and its exit status is copied to the integer referenced by `status_ptr`.

- `int yield(int pid)` - Defers off execution of the calling process to a time determined by the scheduler, in favor of the process with process ID `pid`. If `pid` is -1, the scheduler may determine which process to run next. The only processes whose scheduling should be affected by `yield()` are the calling process, and the process that is `yield()`ed to. If the process with process ID `pid` is not runnable, or doesn't exist, then an integer error code less than zero is returned. Zero is returned on success.

- `int deschedule(int *reject)` - Atomically checks the integer pointed to by `reject`. If the integer is non-zero, the call returns immediately with return value zero. If the integer pointed to by `reject` is zero, then the calling process will not be scheduled to run by the scheduler until a call to `make_runnable()` on the calling process. An integer error code less than zero is returned if reject is not a valid pointer.

- `int make_runnable(int pid)` - Makes the `deschedule()`d process with process ID `pid` runnable by the scheduler. On success, zero is returned. If `pid` is not the process ID of a process that called `deschedule()`, then an integer error code less than zero is returned.

- `int minclone()` - Except for return values, creates an exact copy of the calling process. The PID of the created process is returned to the calling process. Zero is returned to the created process. If something goes wrong, an error code less than zero is returned to the calling process, and no new process is created. The calling process and the created process share an address space. Subsequent calls by the calling or created processes to the `fork()` or `exec()` system calls may be refused.

- `int getpid()` - Returns the process ID of the calling process.

- `void *brk(void *addr)` - Sets the break value of the calling process. The initial break value of a process is the address immediately above the last address used for program instructions and data. This call has the effect of allocating or deallocating enough memory to cover only up to the specified address, rounded up to an integer multiple of the page size. If `addr` is less than the break value when the program began execution, or if `addr` is within four pages of the stack, the break point is not changed. The return value of `brk()` is always the break value after the call has been performed, even if the break value does not change.

- `int sleep(int ticks)` - Deschedules the calling process until at least `ticks` timer interrupts have occurred after the call. Returns immediately if `ticks` is zero. Returns an integer error code less than zero if `ticks` is negative. Returns zero otherwise.

- `char getchar()` - Returns a single character from the character input stream. If the input stream is empty the process is descheduled until a character is available. If some other process is descheduled on a `readline()` or `getchar()`, then the calling process must block and wait its turn to access the input stream. Characters processed by the `getchar()` system call should not be echoed to the console.

- `int readline(int len, char *buf)` - Reads the next line from the console and copies it into the buffer pointed to by `buf`. If there is no line of input currently available, the calling process is descheduled until one is. If some other process is descheduled on a `readline()` or a `getchar()`, then the calling process must block and wait its turn to access the input stream. The length of the buffer is indicated by `len`. If the length of the line exceeds the length of the buffer, only `len`-1 characters should be copied into `buf`. `readline()` should

NULL terminate `buf`. Characters not placed in the buffer should remain available for other calls to `readline()` and `getchar()`. The available line should not be copied into `buf` until there is a newline character available. If the line is smaller than the buffer, then the complete line including the newline character is copied into the buffer. Characters that will be consumed by a `readline()` should be echoed to the console immediately. If there is no outstanding call to `readline()` no characters should be echoed. Echoed user input may be interleaved with output due to calls to `print()`. The readline system call returns the size of the line not including the NULL terminating character on success. An integer error code less than zero is returned if `buf` is not a valid memory address, if `buf` falls in the text section of the process, or if `len` is unreasonably large.

N.B.[1]: The calling process should *not* be scheduled every time a character is processed by the keyboard interrupt handler.

- `int print(int len, char *buf)` - Prints the string pointed to by `buf` to the console. Only `len` characters of this string should be printed. The calling process should block until all characters have been printed to the console. Output of two concurrent `print()`s should not be intermixed. If `len` is larger than some reasonable maximum or if `buf` is not a valid memory address, an integer error code less than zero should be returned.

- `int set_term_color(int color)` - Sets the terminal print color for any future output to the console. If `color` does not specify a valid color, an integer error code less than zero should be returned. Zero is returned on success.

- `int set_cursor_pos(int row, int col)` - Sets the cursor to the location `(row, col)`. If the location is not valid, an integer error code less than zero is returned. Zero is returned on success.

- `int get_cursor_pos(int *row, int *col)` - Writes the current location of the cursor to the addresses provided as arguments. If the arguments are not valid addresses, then an error code less than zero is returned. Zero is returned on success.

- `int ls(int size, char *buf)` - Fills in the user-specified buffer with the names of executable files stored in the system's RAM disk "file system." If there is enough room in the buffer for all of the (null-terminated) file names *and* an additional null byte after the last filename's terminating null, the system call will return the number of filenames successfully copied. Otherwise, an error code less than zero is returned and the contents of the buffer are undefined. For the curious among you, this system call is (very) loosely modeled on the System V `getdents()` call.

- `void halt()` - Ends the simulation by calling the support function `SIM_halt()`.

7 Building and Loading User Programs

[1]N.B. stands for "Nota Bene," Latin for "note well," or "pay attention."

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 `user/user_tests/crt0.c`.

7.2 Loading User Programs

The 410 kernel must read program data from a file, and load the data into a process' address space. Due to the absence of a file system, user programs will be loaded from large arrays compiled directly into the kernel. A utility, `exec2obj`, has been provided; it takes as an argument a list of files, and it creates a .c file containing one char array, named after the file, consisting of the files' data. The file is called `user_apps.c`. It also contains a table of contents the format of which is described in `inc/exec2obj.h`.

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 `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 `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 process 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 `lib/libstdio.a`. The header file for these functions is `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 `lib/libstdlib.a`. The header files for these functions, in `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.

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

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. To build the kernel, list all object files that need to be created under `OBJS` in `kernel.mk`. To build under AFS, build with `make afs`. To build on a computer with an internet connection, but not connected to AFS, build with `make web`. To build on a standalone computer, build with `make offline`.

8.2 User Programming

The same C library is provided for user programs. However, console output functions will not work until the `print()` system call is implemented. Also, a pseudo-random number generator is provided as a user library.

9 Hints on Implementing a Kernel

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.

- 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 process had overflowed its kernel stack onto the kernel stack of another process, there is no way to recover a correct execution state for that process, 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 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.

  You may find the C preprocessor symbols `__FILE__` and `__LINE__` (and maybe even the newfangled `__FUNCTION__`) useful to you in this regard. Note that those symbols begin and end with *two* underline characters.

- 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 processes; think of sets or groups of processes: 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.

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.

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 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'); */
}
```

9.1.3 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;
}
```

[2]or, for you young whippersnappers, ML

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 processes 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.

9.2 Process Initialization

- **Process IDs** - Each process must have a unique integer process ID. Since an integer allows for over two billion processes to be created before overflowing its range, sequential numbers may simply be assigned to each process created without worrying about the possibility of wrap-around – though real operating systems do worry about this. The process ID must be a small, unique integer, not a pointer. The data structure used to map a process ID to its process control block should not be inefficient in space or time. This probably means that a hash table indexed by process ID should be used to store the mapping from process IDs to PCBs.
- **`fork()`** - On a `fork()`, a new process 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. Sine the CPU can only access memory by virtual addresses via page directories, both the source and destination of a copy must be mapped at the same time. Both address spaces need not be mapped at the same time, however. The copy may be done piece-meal, since the address spaces are already naturally divided into pages.
- **`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. To complete the initialization of the stack for a new program, the argument list must be copied above the stack.

9.3 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 process. For grading purposes, you may assume that the "file system" will contain an executable called `idle` which you may run when no other process

is runnable. Or you may choose to hand-craft an idle process without reference to an executable file.

- Create and load the `init` process. 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 `user/410_tests/shell.c`. It is traditional for `init` to loop on `wait()` in order to garbage-collect orphaned zombie processes; it is also traditional for it to react sensibly if the shell exits or is killed.

- Set the first process 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.

9.4 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 wouuld 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 Debugging

10.1 Kernel Debugging

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, and by issuing the `help` command at the simics prompt.

Also available is the `MAGIC_BREAK` macro defined in 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 `lib/inc/stdio.h`.

Also, please note that the kernel memory allocator is very similar to the allocator written by 15-213 students. If 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`.

10.2 User Process Debugging

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

The function call `lprintf()` may be used to output debugging messages from user programs. Its prototype is in `user/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 `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 processes, simply do not register processes with Simics, and use the macro `set_cr3_nodebug()` instead of `set_cr3()`.

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 three checkpoints that will be strictly enforced. The checkpoints exist so that important feedback can be provided, and so should be taken very seriously.

11.1 Checkpoint One

After the first week of the project, pseudocode functions should be written for each system call, and for the page fault handler. Also, the pseudocode functions should be commented so that doxygen will produce HTML documentation. A draft version of the Process Control Block structure should be written and documented in the same way as the pseudocode functions. Source and header files containing the system call pseudocode and documentation, and a header file containing the PCB and documentation are to be handed in.

11.2 Checkpoint Two

After the second week of the project, it should be possible to load and run the idle program provided in `user/410_tests/idle.c`. Also, virtual memory should be enabled, and the keyboard, console and timer drivers should be integrated into the kernel.

11.3 Checkpoint Three

After the third week of the project, it should be possible to call the `get_pid()` and `brk()` system calls from user processes. Also, it should be possible to load and run two user processes such that the timer interrupt handler invokes the scheduler to context switch between the two processes at regular intervals. A working page fault handler should be written. This probably means that the majority of functions for manipulating virtual memory should be implemented.

12 Plan of Attack

A recommended plan of attack has been established. 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. 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.

3. Based on the above step, construct the Process Control Block(PCB). What should go in a PCB? At this point, checkpoint one has been completed.

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

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 processes (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. Write a page fault handler that frames pages on legal accesses, and prints debugging information on bad accesses.

7. 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 process. Load and run the idle process. Verify that timer interrupts are still occurring, and that the timer interrupt handler is sill being run.

8. Write functions for scheduling and context switching. Load a second process. Have the timer interrupt handler context switch between the first and second processes.

9. Integrate the interrupt handler for the keyboard, and the console driver into the kernel. Install an entry for it in the IDT. Keep in mind that the keyboard interrupt handler may need to change later to support the `getchar()` and `readline()` system calls. At this point checkpoint two has been completed.

10. Implement the `getpid()` system call. Once this is working, the system call interface is functioning correctly.

11. Implement the `brk()` system call, and test it with the user process `malloc()`.

12. Implement the `halt()` system call.

13. Fill out the page fault handler. Stack and Heap growth should now be supported. Processes should be killed on bad memory accesses.

14. Implement `fork()`. Keep in mind the hints listed above.

15. Implement `exec()`. Test `fork()` and `exec()` by having an init process spawn a third user process.

16. Implement `wait()`, and `exit()`. Please take care that `exit()` is not grossly inefficient.

17. Implement the `readline()` system call.

18. Implement the `print()`, `set_term_color()`, and `set_cursor_pos()` system calls. At this point, the shell should run.

19. Implement `minclone()`, and test it using the thread library.
20. Implement the `yield()`, `deschedule()`, and `make_runnable()` system calls.
21. Implement the `sleep()` system call. Recall that the scheduler must run in constant time.
22. Implement the `getchar()` system call.
23. Write many test cases for each system call. Try to break the kernel.
24. You’re done! Celebrate!