ECE391: Computer Systems Engineering **Spring 2012**
Machine Problem 3 **Checkpoint 1: Monday 12 March**
Checkpoint 2: Monday 2 April
Checkpoint 3: Monday 16 April
Due: Monday 30 April

Illinix 391

Contents

1 Introduction

Read the whole document before you begin, or you may miss points on some requirements (*e.g.*, the bug log).

In this machine problem, you will work in teams to develop the core of an operating system. We will provide you with code that boots you into protected mode, sets up the GDT, LDT, and initial TSS, and maps a read-only file system image into physical memory for you. You must set up the interrupt descriptor table (IDT), initialize a few devices, write the system call interface along with ten system calls, and provide support for two tasks, one of which you will start from a shell image in the file system, and the second of which will be started from the shell by making system calls. You will need to provide basic paging support for the tasks, including one 4 MB kernel page and one 4 MB application page for each task (a minimal number of entries). In the final stage of the project, you will extend your OS to support multiple tasks, dynamic memory allocation and scheduling.

The goal for the assignment is to provide you with hands-on experience in developing the software used to interface between devices and applications, i.e., operating systems. We have deliberately simplified many of the interfaces to reduce the level of effort necessary to complete the project, but we hope that you will leave the class with the skills necessary to extend the implementation that you develop here along whatever direction you choose by incrementally improving various aspects of your system.

2 Using the Group Repository

This semester, instead of creating your own `svn` repositories, we have had EWS provide them.

One person in each group should check-in an initial copy of the MP3 code: `svn import -m "<comment>" /ece391/mp3 https://subversion.ews.illinois.edu/svn/sp12-ece391/_projects/<name>`
For this command, `<comment>` is an arbitrary comment you want associated with the creation of the new project in your repository. `<name>` is the name of your group. A new directory will be created under svnroot to hold your sources. For example, your command might look like this:

```
# svn import -m "import project"
https://subversion.ews.illinois.edu/svn/sp12-ece391/_projects/Alpha
```
NOTE: the underscore at the start of projects directory name does not always copy-paste correctly

Each member can now check out a version of your sources with:

```
svn co https://subversion.ews.illinois.edu/svn/sp12-ece391/_projects/<name> <client>
```

Initially, it will ask for `user`'s or `root`'s password, so just press enter. When it asks for your username and password, use your NETID and AD password. The checkout command will create an svn checkout directory called `<client>`.

Inside a newly created working directory. You can add/delete files to/from the repository with `svn add [file_list]` and `svn delete [file_list]` respectively. Remember to `svn update` each time you sit down to work on the project and `svn commit` when you are done. Doing so will ensure that all members are working on the most current version of the sources. As a final note, it is bad practice to commit broken sources. You should make sure that your sources compile correctly before committing them to the repository.

3 Getting Started: Booting and GRUB

For this project, you will make use of GRUB (GRand Unified Bootloader) to boot your OS image file. GRUB implements the Multiboot specification, the details of which can be found online at `http://www.gnu.org/software/grub/manual/multiboot/multiboot.html`. You will need to read through at least the chunk of this documentation entitled "Boot information format" to understand the information that GRUB provides to your operating system upon bootup. Various boot parameters are stored in a `multiboot_info_t` data structure, whose address is passed in via EBX to the entry point of the OS image. The `multiboot.h` file contains this and other structure definitions that you should use when accessing the boot information.

GRUB drops you into protected mode, but with paging turned off and all the descriptor tables (GDT, LDT, IDT, TSS) in an undefined state. The first thing to do is set up a GDT, LDT, and TSS. The given code in `boot.S` does this for you, and then calls `entry()` in `kernel.c`, passing the address of the `multiboot_info_t` to it. GRUB has loaded the file system image at some place in physical memory, as a module; there is a section in the `multiboot_info_t` structure which describes how to access the module information, including physical addresses and size information. You will need to keep the base address of this file system module around, since many operations will need to interact with the file system. You may need to extract other boot information as well, such as the physical memory map provided to you by the BIOS (and also passed in as part of the boot information structure).

To get started, read the INSTALL file given in the MP distribution for instructions on booting the OS image file. Once you have the skeleton OS booting, you are ready to begin designing and implementing features. Necessary references for this project are the x86 ISA manuals, which can be found in the "Tools, References, and Links" section of the class website. *Volume 3: System Programming* details things like segmentation, virtual memory, interrupts and exceptions, and task support in all of their gory detail. The other volumes provide invaluable resources such as x86 instruction specifications. You will need to reference these manuals early and often; familiarize yourself with their contents before beginning. To debug your MP, you will take advantage of QEMU, which by now, you should be comfortable with.

One final note: GRUB depends on seeing the Multiboot header within the first 8KB of the OS image file. This header is located in the `boot.S` file of your sources, and `boot.o` is passed as the first object file to the linker to ensure that it falls within the first 8KB of the OS image. Don't move `boot.o` to a different position in the link order given in the Makefile, or GRUB may not be able to boot your OS image.

4 The Pieces

The materials provided to you will launch your machine into protected mode, set up the GDT and LDT as well as a TSS. A file system image with a shell, a few utilities, a single directory entry, and a real-time clock device file will also be mapped into physical memory for you. We will also provide you with the tools necessary to develop your own test applications, including the compilation infrastructure used to produce the shell and utilities and the file system image itself (the `createfs`). For simplicity, we will stick to text-mode graphics, but your OS will in the end support operation of a user-level animation package from MP1. Some basic `printf` support is also provided to aid in debugging, but eventually you will need to write your own output support for the terminal.

5 What to Hand in

5.1 Checkpoint 1: Processor Initialization

For the checkpoint, you must have the following accomplished:

5.1.1 Create group repository

You must have your code in a shared group repository, and each group member will need to demonstrate that they can read and change the source code in the repository.

5.1.2 Load the GDT

You learned about the global descriptor table in both lecture and discussion. Linux creates four segments in this table: Kernel Code Segment, Kernel Data Segment, User Code Segment, and User Data Segment. In `x86_desc.S`, starting at line 38, we have created a global descriptor table for you.

Write code that makes an emulated Intel IA-32 processor use this global descriptor table. We have marked a location in the code (`boot.S` line 27) at which you will need to place this initialization code for the GDT to ensure that you follow the correct boot sequence. You will need to look through the ISA Reference Manual for information about how to write this code (http://courses.ece.uiuc.edu/ece391/references/IA32-ref-manual-vol-3.pdf).

5.1.3 Initialize the IDT

Your IDT must contain entries for exceptions, a few interrupts, and system calls. Consult the x86 ISA manuals for the descriptor formats. The exception handler(s) should use the printing support to report errors when an exception occurs in the kernel, and should squash any user-level program that produces an exception, returning control to the shell (the shell should not cause an exception in a working OS)—see System Calls for further details. You will also need to handle interrupts for the keyboard and the real-time clock. Finally, you will need to use entry `0x80` for system calls, as described below.

5.1.4 Initialize the devices

Adapt the initialization code from Linux to initialize the PIC, the keyboard, and the real-time clock. Set up a general-purpose infrastructure similar to what is done in the Linux kernel. You need to handle the keyboard and RTC interrupts, but you also need to make sure that these devices are initialized before taking interrupts. We suggest that you first mask out all interrupts on the PIC, then initialize the PIC, initialize the devices, and, as part of each device's initialization, enable its associated interrupt on the PIC. The handler addresses should be installed dynamically/ indirectly via a data structure used by the default handlers (as in Linux).

For the checkpoint, your OS must execute the `test_interrupts` handler (provided in `lib.c`) when any interrupt occurs. This handler will change characters on the terminal. This simple test will determine if you have the IDT entries set up correctly, the PIC enabled, and the devices initialized and able to generate interrupts. A good way to test this is to initialize the keyboard controller (enabling its interrupt support) and then press keys to generate interrupts.

5.1.5 Initialize paging

As preparation for the next steps in the MP you must have paging enabled and working. You will be creating a page directory and a page table with valid page directory entries and page table entries. More information about this process appears in Appendix C and in the Intel ISA manual linked from the class web page.

5.1.6 Checkpoint 1 handin

For this handin, we expect you to have written your own "blue screen" of death for each of the exceptions. At a minimum, this screen must identify the exception taken. You may also want to read about how exceptions are handled in the Intel ISA and print more useful information, such as the memory address reference that caused a page fault. This information will be of use to you later for debugging. We expect you to be able to boot your operating system with paging turned on and to enter a `halt loop` or a `while (1) loop`. Then we expect you to boot your operating system and explicitly dereference `NULL` to demonstrate your "blue screen" identifying the resulting page fault. We also expect you to be able to press a key and demonstrate that your operating system reaches the `test_interrupts` function in an interrupt handler, but you do **not** need to write a full interrupt handler, merely show that you can receive interrupts.

5.1.7 Work plan

Although we are not explicitly requiring that you tell us how you plan to split up the work for this project, we do expect that you will want to do so to allow independent progress by all team members. A suggested split of the work (after getting the devices initialized) is to have each person work on one of the subgoals of Checkpoint 2 with one person concentrating on how these functions will eventually integrate with system calls in Checkpoint 3, which connects all previous pieces using the system calls as glue. Checkpoint 4 also requires the group to work together on terminal drivers, RTC interrupts, and everything learned in Checkpoint 3.

Setting up a clean testing interface will also help with partitioning the work, since group members can finish and test parts of the project without other members having finished the other parts (yet).

5.2 Checkpoint 2: Device Drivers

For the second due date, the following functionality must be implemented:

5.2.1 Create a terminal driver

When printable characters are typed at the keyboard, they should be displayed to the screen. You will need to keep track of the screen location for this purpose. You need not support scrolling (wraparound is ok), but you may want to interpret CTRL-L or some similar non-printable key as meaning "clear the screen and put the cursor at the top" to make your testing experience more pleasant. You **do** need to support backspace and line-buffered input. That is, when a `read` system call is made to the terminal, you must only return data after the user presses enter. Until that time, data should be buffered in the driver and edited appropriately (deleted from both screen and buffer when backspace is seen, *etc.*).

Keep in mind that you will also want to have an external interface to support delivery of external data to the terminal output. In particular, `write` system calls to the terminal should integrate cleanly with keyboard input. The `hello` program in the file system will help to test the basics.

5.2.2 Parse the read-only file system

You will need to support operations on the file system image provided to you, including opening and reading from files, opening and reading the directory (there's only one—the structure is flat), and copying program images into contiguous physical memory from the randomly ordered 4 kB "disk" blocks that constitute their images in the file system.

5.2.3 The real-time clock driver

You will need to write the `open`, `read`, and `write` functions for the real-time clock and demonstrate that you can change the clock frequency. You will need to do some research on how the real time clock works and what the device driver needs to do to communicate with it.

5.2.4 Checkpoint 2 handin

You will need to demonstrate that your `open`, `read`, and `write` functions for the three device drivers work correctly. This functionality is independent of how your operating system may use the devices, but it is a good idea for you to start thinking about how you want to interface these functions with the corresponding system calls. You will need to show that when a key is pressed, the keyboard driver stores the corresponding letter in a buffer and that by explicitly calling the keyboard `read` function you can receive the correct letters and print them out on the screen. Similarly, you will need to demonstrate that you can change the rate of the RTC clock using `read`s and `write`s. A good way of doing so is to have the RTC interrupt handler increment a counter and write to a specific spot on the screen. Finally, you will need to demonstrate in Linux that you can read from the read-only file system. A good check for this test is to use the `xxd` command, which prints a hexadecimal dump, to compare the output of your file system code with the bytes stored in the file system image.

5.3 Checkpoint 3: System Calls and Tasks

For the third due date, the following functionality must be implemented:

5.3.1 Support system calls

Ten system calls must be supported via a common IDT entry, so you may want to set up some generic linkage along the lines of that used in Linux, including syscall value checking, register save and restore, and a jump table to C functions that implement the system calls themselves. The details of each call are provided in a later section.

5.3.2 Tasks

Your OS must support two tasks. The first is the `shell` executable provided with the file system, and the second is any other program run from the shell via a system call. Programs execute to completion, so you need not write a scheduler or deal with the timer chip, but you do need to be able to squash programs if they generate exceptions, returning control to the shell in such cases.

As in Linux, the tasks will share common mappings for kernel pages, in this case a single, global 4 MB page. Unlike Linux, we will provide you with set physical addresses for the images of the two tasks, and will stipulate that they require no more than 4 MB each, so you need only allocate a single page for each task's user-level memory.

5.3.3 Support a loader

Extend the code for your file system driver to copy a program image from the randomly ordered 4 kB "disk" blocks constituting the image in the file system into contiguous physical memory This process is normally performed by a program loader in cooperation with the OS, but in your case will be performed completely within the kernel, since the file system code and control of the memory is internal.

In addition, you will need to set up the stack properly and then return into user-level, since privilege level 0 cannot call down into privilege level 3, and your user-level code must execute at the lower privilege level.

5.3.4 Checkpoint 3 handin

For this handin, we expect that you have all of the system calls working and that all of the programs we have provided to you will execute without problems. You will also be expected to be able to answer questions about how you got various system calls to work and about the task structures that you created.

5.4 Checkpoint 4: Scheduling

For the final due date, the following functionality must be implemented:

5.4.1 Multiple terminals and active tasks

As you may already know, it is possible to switch between different terminals in Linux using the **ALT+Function-Key** combination. You will need to add a similar feature by running several instances of the `shell` executable. You must support 3 terminals, each associated with a different instance of `shell`. As an example, pressing **ALT+F2** while in the first terminal must switch to the active task of the second terminal. Further, you must support up to 6 processes in total. For example, each terminal running shell running another program. For the other extreme, have 2 terminals running 1 shell and have 1 terminal running 4 programs (a program on top of shell, on top of shell, etc.)

In order to support the notion of a terminal, you must have a separate input buffer associated with each terminal. In addition, each terminal should save the current text screen and cursor position in order to be able to return to the correct state. Switching between terminals is equivalent to switching between the associated active tasks of the terminals. Finally, your keyboard driver must intercept **ALT+Function-Key** combinations and perform terminal switches.

5.4.2 Scheduling

Until this point, task switching has been done by either executing a new task or by halting an existing one and returning to the parent task. By adding a scheduler, your OS will actively preempt a task in order to switch to the next one. Your OS scheduler should keep track of all tasks and schedule a timer interrupt every 10 to 50 milliseconds in order to switch to the next task in a round-robin fashion.

When adding a scheduler, it is important to keep in mind that tasks running in an inactive terminal should not write to the screen. In order to enforce this rule, a remapping of virtual memory needs to be done for each task. Specifically, the page tables of a task need to be updated to have a task write to non-display memory rather than display memory when the task is not the active one (see the previous section). If the task belongs to the active terminal, the video memory virtual address should be mapped to the physical video memory address,. Otherwise, these virtual addresses must map into different physical pages allocated as a backing store for the task's screen data. These backing pages are then written whenever the task calls `write` on the standard output. Eventually, when the user makes the task's terminal active again, your OS must copy the screen data from the backing store into the video memory and re-map the virtual addresses to point to the video memory's physical address.

5.4.3 Checkpoint 4 handin

For this final handin, we expect you to demonstrate that multiple terminals work by switching between active terminals. We will execute a program on one screen, switch to another screen, start another program, and then expect to be able to switch back and forth to see the programs running. For scheduling, we expect that programs running in the background (on an inactive terminal) will make progress. For example, we expect a fish program running on such a terminal to continue despite not actually being displayed on the screen.

5.5 Extra Credit

We will hold a design competition at the end of the semester. In order to be eligible for this competition, your operating system **must implement all required functionality correctly**. The competition will then be decided based on the amount and difficulty of additional functionality that your team has incorporated into your operating system.

You may also be able to earn some extra credit (or make up for other lost points) by choosing to include some of the features in this section. Extra credit earned in this way will be limited to 10 points of the baseline grade.

5.5.1 Signals

Add support for delivery of five different signals to a task. Appendix F details the specifications and implementation details to make this work.

5.5.2 Dynamic memory allocation

Create a dynamic memory allocator (such as `malloc`). This can be done by simply keeping track of where free pages are using some method, then creating a `malloc` system call that adds a new page to the program's page table or page directory.

5.5.3 Other ideas

Go wild, find something interesting to add to your operating system and explain it. We will consider the difficulty of any addition when determining whether and how much extra credit is merited. For example, don't expect a huge grade increase for adding color support for text.

6 Grading

The rubric here is a baseline score that will be adjusted for teamwork and for individual effort. Note that the "correct" behavior of certain routines includes interfaces that allow one to prevent buffer overflows (*i.e.*, the interfaces do not leave the size of a buffer as an unknown) and other such behavior. While the TAs will probably not have time to sift through all of your code in detail, they will read parts of it and look at your overall design.

comments (10 points total)
(2) full interface description given for each function;
commenting is adequate and appropriate within function bodies
(3) design choices are explained clearly in the code
(5) bug log details experience with debugging

compilation (5 points total)
(5) user code compiles and links with no errors or warnings

Style (15 points total)
(15) good code design; correct use of subroutines and passing of variables; code is easy to read

checkpoint 1 (8 points total)
checkpoint 2 (12 points total)
checkpoint 3 (25 points total)
checkpoint 4 (20 points total)
during each handin demo, the TA will perform a series of tests;
your functionality score will depend on the number of tests that your MP passes

questions (5 points total)
(5) adequately answers questions regarding design and implementation details

extra credit (up to 10 points total)
successfully implement extra functionality

A Note on Teamwork

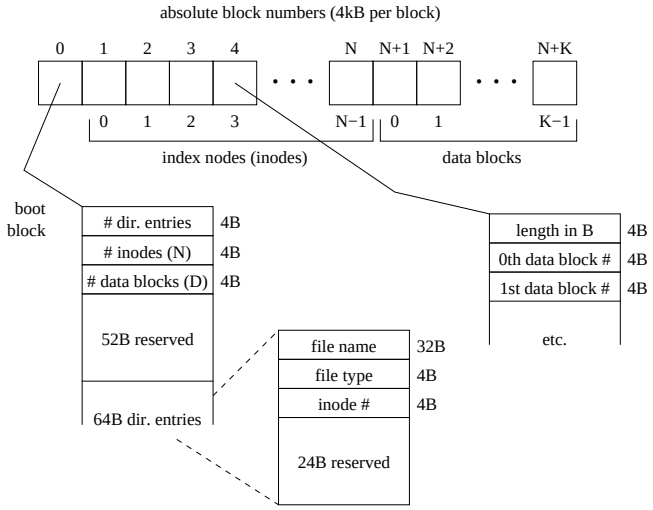
Teamwork is an important part of this class and will be used in the grading of this MP. We expect that you will work to operate effectively as a team, leveraging each member's strengths and making sure that everyone understands how the system operates to the extent that they can explain it. In the final handin, for example, we will ask each of the team members questions and expect that they will be able to answer at a reasonable level **without** referring to another team member. **Failure to operate as a team will significantly reduce your overall grade for this MP.**

We will also ask each of you to apportion credit for the overall MP to each of the other team members (not including yourself) in order to gauge how contributions were balanced amongst your team. This information will be used to adjust your final score for the MP. Teams that operate smoothly together are free to opt for simply marking the form as equal, which is fine, but the information that you provide about relative effort is treated as confidential. Note that you cannot affect your own grade through any choice of credit assignment, only those of your teammates (and vice-versa).

7 Appendix A: The File System

7.1 File System Utilities

The figure below shows the structure and contents of the file system. The file system memory is divided into 4 kB blocks. The first block is called the boot block, and holds both file system statistics and the directory entries. Both the statistics and each directory entry occupy 64B, so the file system can hold up to 63 files. The first directory entry always refers to the directory itself, and is named ".", so it can really hold only 62 files.



Each directory entry gives a name (up to 32 characters, zero-padded, but *not necessarily including a terminal EOS or 0-byte*), a file type, and an index node number for the file. File types are 0 for a file giving user-level access to the real-time clock, 1 for the directory, and 2 for a regular file. The index node number is only meaningful for regular files and should be ignored for the RTC and directory types.

Each regular file is described by an index node that specifies the file's size in bytes and the data blocks that make up the file. Each block contains 4 kB; only those blocks necessary to contain the specified size need be valid, so be careful not to read and make use of block numbers that lie beyond those necessary to contain the file data.

```
int32_t read_dentry_by_name (const uint8_t* fname, dentry_t* dentry);
int32_t read_dentry_by_index (uint32_t index, dentry_t* dentry);
int32_t read_data (uint32_t inode, uint32_t offset, uint8_t* buf, uint32_t length);
```

The three routines provided by the file system module return -1 on failure, indicating a non-existent file or invalid index in the case of the first two calls, or an invalid inode number in the case of the last routine. Note that the directory entries are indexed starting with 0. Also note that the `read_data` call can only check that the given inode is within the valid range. It does not check that the inode actually corresponds to a file (not all inodes are used). However, if a bad data block number is found within the file bounds of the given inode, the function should also return -1.

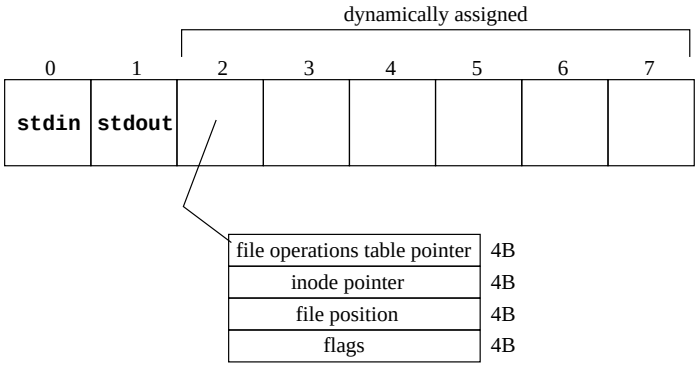
When successful, the first two calls fill in the `dentry_t` block passed as their second argument with the file name, file type, and inode number for the file, then return 0. The last routine works much like the `read` system call, reading up to `length` bytes starting from position `offset` in the file with inode number `inode` and returning the number of bytes read and placed in the buffer. A return value of 0 thus indicates that the end of the file has been reached.

7.2 File System Abstractions

Each task can have up to 8 open files. These open files are represented with a file array, stored in the process control block. The integer index into this array is called a **file descriptor**, and this integer is how user-level programs identify the open file.

This array should store a structure containing:

1. The file operations jump table associated with the correct file type. This jump table should contain entries for `open`, `read`, and `write`, to perform type-specific actions for each operation. `open` is used for performing type-specific initialization. For example, if we just `open`'d the real-time clock, the jump table pointer in this structure should store the RTC's file operations table.
2. A pointer to the inode for this file. This is only valid for data files, and should be `NULL` for directories and the RTC device file.
3. A "file position" member that keeps track of where the user is currently reading from in the file. Every `read` system call should update this member.
4. A "flags" member for, among other things, marking this file descriptor as "in-use."



When a process is started, the kernel should automatically open `stdin` and `stdout`, which correspond to file descriptors 0 and 1, respectively. `stdin` is a read-only file which corresponds to keyboard input. `stdout` is a write-only file corresponding to terminal output. "Opening" these files consists of storing appropriate jump tables in these two locations in the file array, and marking the files as in-use. For the remaining six file descriptors available, an entry in the file array is dynamically associated with the file being `open`'d whenever the `open` system call is made (return -1 if the array is full).

8 Appendix B: The System Calls

You must support ten system calls, numbered 1 through 10. As with Linux, they are invoked using `int $0x80`, and use a similar calling convention. In particular, the call number is placed in EAX, the first argument in EBX, then ECX, and finally EDX. No call uses more than three arguments, although you should protect all of the registers from modification by the system call to avoid leaking information to the user programs. The return value is placed in EAX if the call returns (not all do); a value of -1 indicates an error, while others indicate some form of success.

Prototypes appear below. Unless otherwise specified, successful calls should return 0, and failed calls should return -1.

```
 1. int32_t halt (uint8_t status);
 2. int32_t execute (const uint8_t* command);
 3. int32_t read (int32_t fd, void* buf, int32_t nbytes);
 4. int32_t write (int32_t fd, const void* buf, int32_t nbytes);
 5. int32_t open (const uint8_t* filename);
 6. int32_t close (int32_t fd);
 7. int32_t getargs (uint8_t* buf, int32_t nbytes);
 8. int32_t vidmap (uint8_t** screen_start);
 9. int32_t set_handler (int32_t signum, void* handler_address);[1]
10. int32_t sigreturn (void);[2]
```

The `execute` system call attempts to load and execute a new program, handing off the processor to the new program until it terminates. The command is a space-separated sequence of words. The first word is the file name of the program to be executed, and the rest of the command—stripped of leading spaces—should be provided to the new program on request via the `getargs` system call. The `execute` call returns -1 if the command cannot be executed, *e.g.*, if the program does not exist or the filename specified is not an executable, 256 if the program dies by an exception, or a value in the range 0 to 255 if the program executes a `halt` system call, in which case the value returned is that given by the program's call to `halt`.

The `halt` system call terminates a process, returning the specified value to its parent process. The system call handler itself is responsible for expanding the 8-bit argument from BL into the 32-bit return value to the parent program's `execute` system call. Be careful not to return all 32 bits from EBX. This call should never return to the caller.

The `read` system call reads data from the keyboard, a file, device (RTC), or directory. This call returns the number of bytes read, or 0 to indicate that the end of the file has been reached (for normal files and the directory). In the case of the keyboard, read should return data from one line that has been terminated by pressing Enter, or as much as fits in the buffer from one such line. The line returned should include the line feed character. In the case of a file, data should be read to the end of the file or the end of the buffer provided, whichever occurs sooner. In the case of reads to the directory, only the filename should be provided (as much as fits, or all 32 bytes), and subsequent reads should read from successive directory entries until the last is reached, at which point `read` should repeatedly return 0. For the real-time clock, this call should always return 0, but only after an interrupt has occurred (set a flag and wait until the interrupt handler clears it, then return 0). You should use a jump table referenced by the task's file array to call from a generic handler for this call into a file-type-specific function. This jump table should be inserted into the file array on the `open` system call (see below).

The `write` system call writes data to the terminal or to a device (RTC). In the case of the terminal, all data should be displayed to the screen immediately. In the case of the real-time clock, the system call should always accept only a 4-byte integer specifying the interrupt rate in Hz, and should set the rate of periodic interrupts accordingly. Writes to regular files should always return -1 to indicate failure since the file system is read-only. The call returns the number of bytes written, or -1 on failure.

The RTC device itself can only generate interrupts at a rate that is a power of 2 (do a parameter check), and only up to 8192 Hz. Your kernel should limit this further to 1024 Hz — an operating system shouldn't allow user

[1] Extra credit.
[2] Also for extra credit.

space programs to generate more than 1024 interrupts per second by default. Look at `drivers/char/rtc.c`, `include/linux/mc146818rtc.h` and possibly other associated header files for the macros and port numbers for interfacing with the RTC device.

Also, see the datasheet at `http://courses.ece.uiuc.edu/ece398/references/mc146818.pdf` (old) and `http://courses.ece.uiuc.edu/ece398/references/ds12887.pdf` (new) for details on registers and other parameters of the hardware.

Note that you should be using the RTC's Periodic Interrupt function to generate interrupts at a programmable rate. The RTC interrupt rate should be set to a default value of 2 Hz (2 interrupts per second) when the real-time clock device is opened, only one program should be allowed to open the device at any time, and the program should only be allowed to open the device once (one file descriptor). For simplicity, RTC interrupts should remain on at all times.

The `open` system call provides access to the file system. The call should find the directory entry corresponding to the named file, allocate an unused file descriptor, and set up any data necessary to handle the given type of file (directory, real-time clock device, or regular file). If the named file does not exist or no descriptors are free, the call returns -1.

The `close` system call closes the specified file descriptor and makes it available for return from later calls to `open`. You should not allow the user to close the default descriptors (0 for input and 1 for output). Trying to close an invalid descriptor should result in a return value of -1; successful closes should return 0.

The `getargs` call reads the program's command line arguments into a user-level buffer. Obviously, these arguments must be stored as part of the task data when a new program is loaded. Here they are merely copied into user space. If the arguments and a terminal NULL (0-byte) do not fit in the buffer, simply return -1. The shell does not request arguments, but you should probably still initialize the shell task's argument data to the empty string.

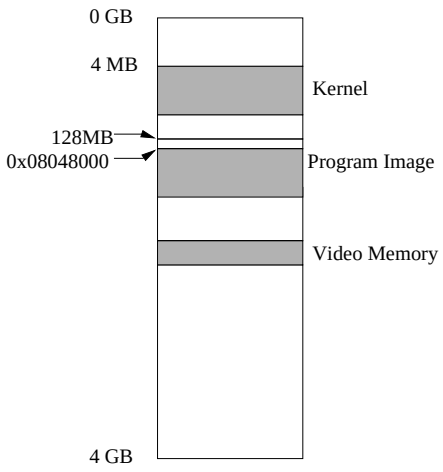
The `vidmap` call maps the text-mode video memory into user space at a pre-set virtual address. Although the address returned is always the same (see the memory map section later in this handout), it should be written into the memory location provided by the caller (which must be checked for validity). If the location is invalid, the call should return -1. To avoid adding kernel-side exception handling for this sort of check, you can simply check whether the address falls within the address range covered by the single user-level page. Note that the video memory will require you to add another page mapping for the program, in this case a 4 kB page.

The `set_handler` and `sigreturn` calls are related to signal handling and are discussed in the section **Signals** below. Even if your operating system does not support signals, you must support these system calls; in such a case, however, you may immediately return failure from these calls.

Note that some system calls need to synchronize with interrupt handlers. For example, the `read` system call made on the RTC device should wait until the next RTC interrupt has occurred before it returns. Use simple volatile flag variables to do this synchronization (e.g., something like `int rtc_interrupt_occurred;`) when possible (try something more complicated only after everything works!), and small critical sections with `cli`/`sti`. For example, writing to the RTC should block interrupts to interact with the device. Writing to the terminal also probably needs to block interrupts, if only briefly, to update screen data when printing (keyboard input is also printed to the screen from the interrupt handler).

9 Appendix C: Memory Map and Task Specification

When processing the `execute` system call, your kernel must create a virtual address space for the new process. This will involve setting up a new Page Directory with entries corresponding to the figure shown on the right.



The virtual memory map for each task is show in the figure. The kernel is loaded at physical address `0x400000` (4MB), and also mapped at virtual address 4MB. A global page directory entry with its Supervisor bit set should be set up to map the kernel to virtual address `0x400000` (4MB). This ensures that the kernel, which is linked to run with its starting address at 4MB, will continue to work even after paging is turned on.

To make physical memory management easy, you may assume there is at least 16MB of physical memory on the system. Then, use the following (static) strategy: the first user-level program (the shell) should be loaded at physical 8MB, and the second user-level program, when it is executed by the shell, should be loaded at physical 12MB. The program image itself is linked to execute at virtual address `0x08048000`. The way to get this working is to set up a single 4MB page directory entry that maps virtual address `0x08000000` (128MB) to the right physical memory address (either 8MB or 12MB). Then, the program image must be copied to the correct offset (`0x00048000`) within that page.

Both the kernel mapping and the user-level program mapping are critical; memory references in neither the kernel nor the program will not work correctly unless they are mapped at these exact addresses.

The layout of executable files in the file system is simple: the entire file stored in the file system is the image of the program to be executed. In this file, a header that occupies the first 40 bytes gives information for loading and starting the program. The first 4 bytes of the file represent a "magic number" that identifies the file as an executable. These bytes are, respectively, 0: `0x7f`; 1: `0x45`; 2: `0x4c`; 3: `0x46`. If the magic number is not present, the `execute` system call should fail. The other important bit of information that you need to execute programs is the entry point into the program, i.e., the virtual address of the first instruction that should be executed. This information is stored as a 4-byte unsigned integer in bytes 24-27 of the executable, and the value of it falls somewhere near `0x08048000` for all programs we have provided to you. When processing the `execute` system call, your code should make a note of the entry point, and then copy the entire file to memory starting at virtual address `0x08048000`. It then must jump to the entry point of the program to begin execution. The details of how to jump to this entry point are explained in the next section.

10 Appendix D: Executing User-level Code

Kernel code executes at privilege level 0, while user-level code must execute at privilege level 3. The x86 processor does not allow a simple function call from privilege level 0 code to privilege level 3, so you must use an x86-specific convention to accomplish this privilege switch.

The convention to use is the `IRET` instruction. Read the ISA reference manual for the details of this instruction. You must set up the correct values for the user-level `EIP`, `CS`, `EFLAGS`, `ESP`, and `SS` registers on the kernel-mode stack, and then execute an `IRET` instruction. The processor will pop the values off the stack into those registers, and by doing this, will perform a privilege switch into the privilege level specified by the low 2 bites of the `CS` register. The values for the `CS` and `SS` registers must point to the correct entries in the Global Descriptor Table that correspond to the user-mode code and stack segments, respectively. The `EIP` you need to jump to is the entry point from bytes 24-27 of the executable that you have just loaded. Finally, you need to set up a user-level stack for the process. For simplicity, you may simply set the stack pointer to the bottom of the 4MB page already holding the executable image. Two final bits: the `DS` register must be set to point to the correct entry in the Global Descriptor Table for the user mode data segment (`USER_DS`) before you execute the `IRET` instruction (conversely, when an entry into the kernel happens, *e.g.*, a system call, exception, or interrupt, you should set `DS` to point to the `KERNEL_DS` segment). Finally, you will need to modify the TSS; this is explained in the later section called "Stack Switching and the TSS".

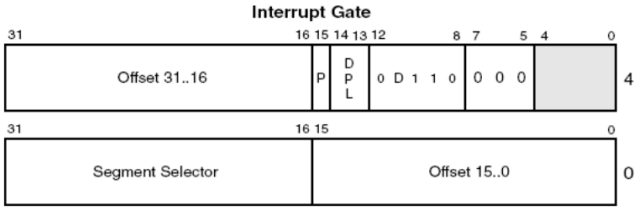
10.1 Process Control Block

The next piece to support tasks in your operating system is per-task data structures, *e.g.*, the process control block. One bit of per-task state that needs to be saved is the file array, described earlier; another is the signal information. You may need to store some other things in the process control block; you must figure the rest out on your own. The final bit of per-task state that needs to be allocated is a kernel stack for each user-level program. Since you only need to support two tasks, you may simply place the first task's kernel stack at the bottom of the 4MB kernel page that you have already allocated. The second task's stack can then go 8KB above it. This way, both tasks will have 8KB kernel stacks to use when inside the kernel. Each process's process control block should be stored at the top of its 8KB stack, and the stack should grow towards them. Since you've put both stacks inside the 4MB page, there is no need to "allocate" memory for the process control block. To get at each process's process control block, you need only AND the process's `ESP` register with an appropriate bit mask to reach the top of its 8KB kernel stack, which is the start of its process control block. Finally, when a new task is started with the `execute` system call, you'll need to store the parent task's process control block pointer in the child task's process control block so that when the child program calls `halt`, you are able to return control to the parent task.

11 Appendix E: System Calls, Exceptions, and Interrupts

Recall that when a hardware interrupt is asserted or a hardware exception is detected, a specific number is associated with the exception or interrupt to differentiate between different types of exceptions, or different hardware devices (for example, differentiating between the keyboard interrupt, the network card interrupt, and a divide-by-zero exception). This number is used to index a table, called the **Interrupt Descriptor Table**, or IDT. The format of an IDT entry is shown in the figure below, and the details of it are found in the Intel architecture manuals (see the "Getting Started" section for more info on references). Each IDT entry contains, among other things, a pointer to the corresponding interrupt handler function to be run when that interrupt is received. When an exception or hardware interrupt is detected, the processor switches into privilege level 0 (kernel mode), saves some, but not all of the processor registers on the kernel stack (see the section "Stack Switching and the TSS" for more details), and jumps to the function address specified in the entry. Now, kernel code, specifically the interrupt handler for the correct interrupt number, is now executing.

For system calls, you will use a similar mechanism. A user-level program will execute a `int $0x80` instruction. The `int` instruction functions similar to an exception. It specifies that the processor should use entry `0x80` in the IDT as the handler when this instruction is executed. The same privilege-level switching, stack switching, *etc.*, are all performed, so after this instruction is run, kernel code will be executing. You must set up a "system call handler" IDT entry at index `0x80`, as well as a function that will be run for all system calls. This function can then differentiate different system calls based on the parameter passed in EAX.



For this to work properly, you must pay attention to a few details of the x86 architecture and protection scheme when setting up the IDT. The IDT will contain entries for exception handlers, hardware interrupt handlers, and the system call handler. Each entry in the IDT has a Descriptor Privilege Level (DPL) that specifies the privilege level needed to use that descriptor. Hardware interrupt handlers and exception handlers should have their DPL set to 0 to prevent user-level applications from calling into these routines with the `int` instruction. The system call handler should have its DPL set to 3 so that it *is* accessible from user space via the `int` instruction. Finally, each IDT entry also contains a segment selector field that specifies a code segment in the GDT, and you should set this field to be the kernel's code segment descriptor. When the x86 sees that a new CS is specified, it will perform a privilege switch, and the handler for the IDT entry will run in the new privilege level. This way, the system call interface is accessible to user space but the code executes in the kernel.

11.1 Stack Switching and the TSS

The last detail of user space to kernel transitions on system calls, interrupts, or exceptions is stack switching. The stack switch is taken care of by the x86 hardware. The x86 processor supports the notion of a task; this hardware support is encapsulated in a **Task State Segment**, or TSS. You will not use the full x86 hardware support for tasks in this project, but the x86 requires that you set up one TSS for, among other things, privilege level stack switching. The TSS in the given code is a placeholder. Read the Intel manuals for details on the fields in it; the important fields are SS0 and ESP0. These fields contain the stack segment and stack pointer that the x86 will put into SS and ESP when performing a privilege switch from privilege level 3 to privilege level 0 (for example, when a user-level program makes a system call, or when a hardware interrupt occurs while a user-level program is executing). These fields must be set to point to the kernel's stack segment and the process's kernel-mode stack, respectively. Note that when you start a new process, just before you switch to that process and start executing its user-level code, you must alter the TSS entry to contain its new kernel-mode stack pointer. This way, when a privilege switch is needed, the correct stack will be set up by the x86 processor.

12 Appendix F: Signals

Your OS will provide an infrastructure for user-level signals, similar to Linux. The table details the signals that will be supported.

Signal name	Signal number	Default action
DIV_ZERO	0	Kill the task
SEGFAULT	1	Kill the task
INTERRUPT	2	Kill the task
ALARM	3	Ignore
USER1	4	Ignore

The `set_handler` system call changes the default action taken when a signal is received: the `signum` parameter specifies which signal's handler to change, and the `handler_address` points to a user-level function to be run when that signal is received. It returns 0 if the handler was successful set, and -1 on failure. If `handler_address` is NULL (zero), the kernel should reset the action taken to be the default action.

`DIV_ZERO` should be sent to a task when the x86 processor generates a divide-by-zero exception while executing user-level code. `SEGFAULT` should be sent when any other exception occurs, including any illegal instructions, illegal memory references, page faults, general protection faults, illegal opcodes, *etc*. The `INTERRUPT` signal should be sent when a CTRL+C is pressed on the keyboard. The `ALARM` signal should be sent to the currently-executing task (there is only one currently-executing task in this OS) every 10 seconds. This should be implemented by knowing at what rate the real-time clock's Periodic Interrupts occur, counting how many Periodic Interrupts have occurred, and sending an `ALARM` signal after 10 seconds have elapsed. Finally, `USER1` is user-defined and can be used to implement any other signal of your choosing.

Signals should only be delivered to a task when returning to user space from the kernel, so you'll want to add some code in your return-to-user space linkage to check for pending signals. To support signal delivery, you should use a mechanism similar to what Linux uses:

1. Mask all other signals.
2. Set up the signal handler's stack frame. You'll need the current value of the user-level ESP register to find the user's current stack location. The signal handler stack frame goes directly above this on the stack.

   The signal handler stack frame is shown in Figure 1. Setting up the signal handler stack frame involves: copying a return address and a signal number parameter to the user-level stack, copying the process's hardware context (see Figure 2) from the point when the program was interrupted for the signal, and copying a small amount of assembly linkage to the user-level stack that calls `sigreturn` when the signal handler is finished.
3. Finally, execute (in user space) the handler specified in the signal descriptor. No other information needs to be passed to the signal handler (no `siginfo_t` structure like the modern Linux signals).

When the user-level signal handler returns, it will use the return address you have copied on its stack, which will jump to the assembly linkage (also on the stack). This assembly linkage should make the `sigreturn` system call (using the standard `int $0x80` user-level system call calling convention).

The `sigreturn` system call should copy the hardware context that was on the user-level stack back onto the processor. To find the hardware context, you will need to know the user-level value of ESP (will be saved by your system call handler) as well as the exact setup of the user-level stack frame. To copy the hardware context back onto the processor, you will actually overwrite the kernel's copy of the process's hardware context that was saved on the *kernel* stack when it handled the `sigreturn` system call. In this way, when the `sigreturn` system call handler returns to user space, the hardware context will automatically be copied back onto the processor by your return-from-kernel code that you have already written. One thing to be careful of: you'll probably have system calls set up to return a value (in EAX) to user space. Be sure you don't clobber the user's EAX value from its hardware context with a bogus "return value" from `sigreturn` – have `sigreturn` return the hardware context's EAX value so that you won't have to special-case the return from `sigreturn`.

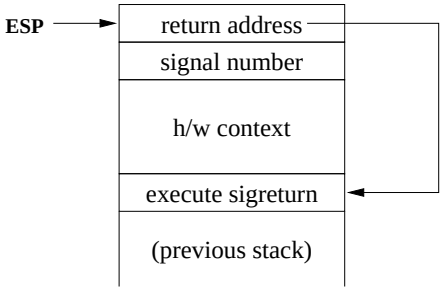


Figure 1: User-level signal handler stack frame.

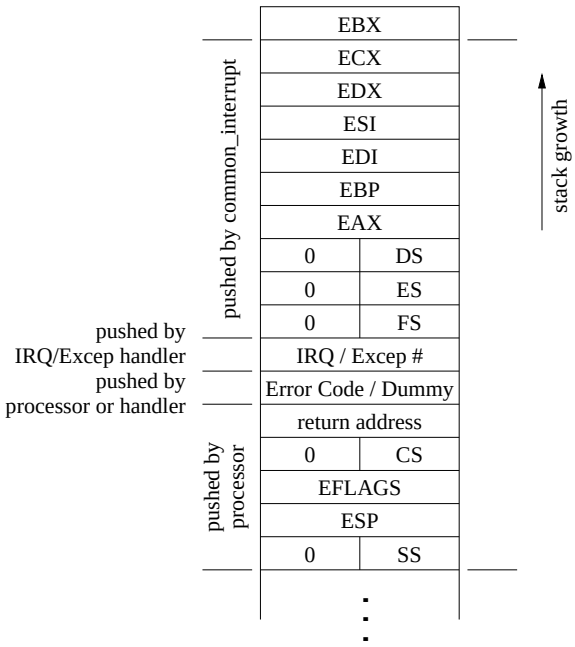


Figure 2: Hardware context structure.

Shown in Figure 2 is a slightly-modified version of the `struct ptregs` structure that Linux uses for its hardware context; this modified structure is what you should use in this MP. The "Error Code / Dummy" field has been added to the hardware context to simplify exception handling. For some exceptions, the processor pushes an error code onto the stack after XCS (for example, page faults do) whereas other exceptions do not (divide-by-zeros do not). For exceptions that do not push this error code, your exception handler should push a dummy value to take up the error code slot[3]. x86 interrupts never push the error code, so you must also push a dummy value in all of your interrupt handlers and the system call handler. You can find documentation about which exceptions push error codes (and much more about exceptions) in *Volume III: System Programming* of the Intel ISA manual on the Tools, References, and Links section of the website.

Finally, signal handling information should go in the process control block. You will need to keep track of pending signals, masked signals, and handler actions / addresses for each signal. Much information on Linux's implementation of signals (which your implementation will closely match) can be found in *Understanding the Linux Kernel* chapter 10.

[3]Linux's `struct ptregs` does not include this error code field; instead, the Linux exception handlers play tricks with the stack and registers to avoid adding an extra slot.

13 Appendix G: Debugging with QEMU

This section describes how to build your new OS which will be embedded, together with the filesys_img, in mp3.img.

Whenever a change is made to your kernel or file system, you need to `sudo make` a new kernel, which consequently prepare the mp3.img file which is used in the newly modified `debug.lnk` file.

Your new `debug.lnk` file should be modified to pass the new mp3 QEMU image. Change the target line to `"c:\Program Files\Qemu\qemu.exe" -hda "<mp3 directory>/mp3.img" -m 256 -no-kqemu -s -S -L pc-bios`

You may also write the following to a .bat file to accomplish the same thing:

```
c:
cd "c:\Program Files\qemu"
qemu.exe -hda <mp3 directory>\mp3.img -m 256 -no-kqemu -s -S -L pc-bios
```

where `<mp3 directory>` is likely to be: `z:\source\mp3\student_distrib`. In order to start debugging with GDB, run your `debug.lnk` file, then run the `devel.lnk` file and issue the following commands in the development machine:

```
cd <mp3_directory>\student_distrib
gdb bootimg
target remote 10.0.2.2:1234
```

GDB should now be started and connected to QEMU. From here, you can set up break points and everything else that you would normally do with GDB. Type `c` to continue execution instead of `r` since you connected to QEMU which is already running. When you do continue execution in GDB, GRUB will load first in QEMU. You need to hit enter, or wait 5 seconds, for your OS to load. QEMU is known to crash if your page tables are incorrectly setup. You might have to use the task manager in Windows to kill QEMU if this happens.