Projects for an Operating Systems Class

This repository holds a number of projects that can be used in an operating systems class aimed at upper-level undergraduates, from either **Southeast University**, or **Efrei Paris**.

Also available are some tests to see if your code works. A specific testing script, found in each project directory, can be used to run the tests against your code.

For example, in the initial utilities project, the relatively simple `seucat` program that you create can be tested by running the `test-seucat.sh` script. This could be accomplished by the following commands:

```
prompt> cd projects/initial-utilities/seucat
prompt> emacs -nw seucat.c
prompt> gcc -o seucat seucat.c -Wall
prompt> sudo chmod 777 test-seucat.sh
prompt> ./test-seucat.sh
test 1: passed
test 2: passed
test 3: passed
test 4: passed
test 5: passed
test 6: passed
test 7: passed
prompt>
```

Of course, this sequence assumes (a) you use `emacs`, (b) your code is written in one shot, and (c) that it works perfectly. Even for simple assignments, it is likely that the compile/run/debug cycle might take a few iterations.

Syllabus of OS Labs

Chapter	**# Project**
Introduction	1 Reverse, Unix Utilities
Operating System Structures	2 Xv6 Syscall (part 1)
Processes	3 Unix Shell
Threads	4 Xv6 Kernel Threads
CPU Scheduling	5 Xv6 Scheduling (Lottery)
Process Synchronization	6 Xv6 Syscall (part 2)
Deadlocks	7 Map Reduce
Main Memory	
Virtual Memory	8 Xv6 Virtual Memory
Mass-Storage Structure	
File-System Interface	
File-System Implementation	9 File System Checker
I/O	

The projects marked as blue are **kernel hacking projects**. They are to be done inside the xv6 kernel based on an early version of Unix and developed at MIT. Unlike the C/Linux projects, these give you direct experience inside a real, working operating system.

The `run-tests.sh` script is called by various testers to do the work of testing. Each test is actually fairly simple: it is a comparison of standard output and standard error, as per the program specification.

In any given program specification directory, there exists a specific `tests/` directory which holds the expected return code, standard output, and standard error in files called `n.rc`, `n.out`, and `n.err` (respectively) for each test `n`. The testing framework just starts at `1` and keeps incrementing tests until it can't find any more or encounters a failure. Thus, adding new tests is easy; just add the relevant files to the tests directory at the lowest available number.

The files needed to describe a test number `n` are: - `n.rc`: The return code the program should return (usually 0 or 1) - `n.out`: The standard output expected from the test - `n.err`: The standard error expected from the test - `n.run`: How to run the test (which arguments it needs, etc.) - `n.desc`: A short text description of the test - `n.pre` (optional): Code to run before the test, to set something up - `n.post` (optional): Code to run after the test, to clean something up

There is also a single file called `pre` which gets run once at the beginning of testing; this is often used to do a more complex build of a code base, for example. To prevent repeated time-wasting pre-test activity, suppress this with the `-s` flag (as described below).

In most cases, a wrapper script is used to call `run-tests.sh` to do the necessary work.

The options for `run-tests.sh` include: * `-h` (the help message) * `-v` (verbose: print what each test is doing) * `-t n` (run only test `n`) * `-c` (continue even after a test fails) * `-d` (run tests not from `tests/` directory but from this directory instead) * `-s` (suppress running the one-time set of commands in `pre` file)

There is also another script used in testing of `xv6` projects, called `run-xv6-command.exp`. This is an `expect` script which launches the qemu emulator and runs the relevant testing command in the xv6 environment before automatically terminating the test. It is used by the `run-tests.sh` script as described above and thus not generally called by users directly.

Reverse

This project is a simple warm-up to get you used to how this whole project thing will go. It also serves to get you into the mindset of a C programmer. You will write a simple program called `reverse`. This program should be invoked in one of the following ways:

```
prompt> ./reverse
prompt> ./reverse input.txt
prompt> ./reverse input.txt output.txt
```

The above line means the users typed in the name of the reversing program `reverse` (the `./` in front of it simply refers to the current working directory (called dot, referred to as `.`) and the slash (`/`) is a separator; thus, in this directory, look for a program named `reverse`) and gave it either no command-line arguments, one command-line argument (an input file, `input.txt`), or two command-line arguments (an input file and an output file `output.txt`).

An input file might look like this:

```
hello
this
is
a file
```

The goal of the reversing program is to read in the data from the specified input file and reverse it; thus, the lines should be printed out in the reverse order of the input stream. Thus, for the aforementioned example, the output should be:

```
a file
is
this
hello
```

The different ways to invoke the file (as above) all correspond to slightly different ways of using this simple new Unix utility. For example, when invoked with two command-line arguments, the program should read from the input file the user supplies and write the reversed version of said file to the output file the user supplies.

When invoked with just one command-line argument, the user supplies the input file, but the file should be printed to the screen. In Unix-based systems, printing to the screen is the same as writing to a special file known as **standard output**, or `stdout` for short.

Finally, when invoked without any arguments, your reversing program should read from **standard input** (`stdin`), which is the input that a user types in, and write to standard output (i.e., the screen).

Sounds easy, right? It should. But there are a few details...

Details

Assumptions and Errors

- **Input is the same as output:** If the input file and output file are the same file, you should print out an error message "Input and output file must differ" and exit with return code 1.
- **String length:** You may not assume anything about how long a line should be. Thus, you may have to read in a very long input line...
- **File length:** You may not assume anything about the length of the file, i.e., it may be **VERY** long.
- **Invalid files:** If the user specifies an input file or output file, and for some reason, when you try to open said file (e.g., `input.txt`) and fail, you should print out the following exact error message: `error: cannot open file 'input.txt'` and then exit with return code 1 (i.e., call `exit(1);`).

- **Malloc fails:** If you call `malloc()` to allocate some memory, and malloc fails, you should print the error message `malloc failed` and exit with return code 1.
- **Too many arguments passed to program:** If the user runs `reverse` with too many arguments, print `usage: reverse <input> <output>` and exit with return code 1.
- **How to print error messages:** On any error, you should print the error to the screen using `fprintf()`, and send the error message to `stderr` (standard error) and not `stdout` (standard output). This is accomplished in your C code as follows: `fprintf(stderr, "whatever the error message is\n");`

Useful Routines

To exit, call `exit(1)`. The number you pass to `exit()`, in this case 1, is then available to the user to see if the program returned an error (i.e., return a non-zero) or exited cleanly (i.e., returned 0).

For reading in the input file, the following routines will make your life easy: `fopen()`, `getline()`, and `fclose()`.

For printing (to screen, or to a file), use `fprintf()`. Note that it is easy to write to standard output by passing `stdout` to `fprintf()`; it is also easy to write to a file by passing in the `FILE *` returned by `fopen`, e.g., `fp=fopen(...); fprintf(fp, ...);`.

The routine `malloc()` is useful for memory allocation. Perhaps for adding elements to a list?

If you don't know how to use these functions, use the man pages. For example, typing `man malloc` at the command line will give you a lot of information on malloc.

Tips

Start small, and get things working incrementally. For example, first get a program that simply reads in the input file, one line at a time, and prints out what it reads in. Then, slowly add features and test them as you go.

For example, the way we wrote this code was first to write some code that used `fopen()`, `getline()`, and `fclose()` to read an input file and print it out. Then, we wrote code to store each input line into a linked list and made sure that worked. Then, we printed out the list in reverse order. Then we made sure to handle error cases. And so forth...

Testing is critical. A great programmer we once knew said you have to write five to ten lines of test code for every line of code you produce; testing your code to make sure it works is crucial. Write tests to see if your code handles all the cases you think it should. Be as comprehensive as you can be. Of course, when grading your projects, we will be. Thus, it is better if you find your bugs first, before we do.

Keep old versions around. Keep copies of older versions of your program around, as you may introduce bugs and not be able to easily undo them. A simple way to do this is to keep copies around, by explicitly making copies of the file at various points during development. For example, let's say you get a simple version of `reverse.c` working (say, that just reads in the file); type `cp reverse.c reverse.v1.c` to make a copy into the file `reverse.v1.c`. More sophisticated developers use version control systems git (perhaps through github); such a tool is well worth learning, so do it!

Unix Utilities

In this project, you'll build a few different UNIX utilities, simple versions of commonly used commands like **cat**, **ls**, etc. We'll call each of them a slightly different name to avoid confusion; for example, instead of **cat**, you'll be implementing **seucat** (i.e., "Southeast University" cat).

Objectives:

- Re-familiarize yourself with the C programming language
- Re-familiarize yourself with a shell / terminal / command-line of UNIX
- Learn (as a side effect) how to use a proper code editor such as emacs
- Learn a little about how UNIX utilities are implemented

While the project focuses upon writing simple C programs, you can see from the above that even that requires a bunch of other previous knowledge, including a basic idea of what a shell is and how to use the command line on some UNIX-based systems (e.g., Linux or macOS), how to use an editor such as emacs, and of course a basic understanding of C programming. If you **do not** have these skills already, this is not the right place to start.

Summary of what gets turned in:

- A bunch of single .c files for each of the utilities below: **seucat.c**, **seugrep.c**, **seuzip.c**, and **seuunzip.c**.
- Each should compile successfully when compiled with the **-Wall** and **-Werror** flags.
- Each should (hopefully) pass the tests we supply to you.

seucat

The program **seucat** is a simple program. Generally, it reads a file as specified by the user and prints its contents. A typical usage is as follows, in which the user wants to see the contents of main.c, and thus types:

```
prompt> ./seucat seucat.c
#include <stdio.h>
...
```

As shown, **seucat** reads the file **seucat.c** and prints out its contents. The "**./**" before the **seucat** above is a UNIX thing; it just tells the system which directory to find **seucat** in (in this case, in the "." (dot) directory, which means the current working directory).

To create the **seucat** binary, you'll be creating a single source file, **seucat.c**, and writing a little C code to implement this simplified version of **cat**. To compile this program, you will do the following:

```
prompt> gcc -o seucat seucat.c -Wall -Werror
prompt>
```

This will make a single *executable binary* called **seucat** which you can then run as above.

You'll need to learn how to use a few library routines from the C standard library (often called **libc**) to implement the source code for this program, which we'll assume is in a file called **seucat.c**. All C code is automatically linked with the C library, which is full of useful functions you can call to implement your program.

For this project, we recommend using the following routines to do file input and output: **fopen()**, **fgets()**, and **fclose()**, etc. Whenever you use a new function like this, the first thing you should do is read about it -- how else will you learn to use it properly?

On UNIX systems, the best way to read about such functions is to use what are called the **man** pages (short for **manual**). In our HTML/web-driven world, the man pages feel a bit antiquated, but they are useful and informative and generally quite easy to use.

To access the man page for **fopen()**, for example, just type the following at your UNIX shell prompt:

```
prompt> man fopen
```

Then, read! Reading man pages effectively takes practice; why not start learning now?

We will also give a simple overview here. The **fopen()** function "opens" a file, which is a common way in UNIX systems to begin the process of file access. In this case, opening a file just gives you back a pointer to a structure of type **FILE**, which can then be passed to other routines to read, write, etc.

Here is a typical usage of **fopen()**:

```
FILE *fp = fopen("main.c", "r");
if (fp == NULL) {
    printf("cannot open file\n");
    exit(1);
}
```

A couple of points here. First, note that **fopen()** takes two arguments: the *name* of the file and the *mode*. The latter just indicates what we plan to do with the file. In this case, because we wish to read the file, we pass "r" as the second argument. Read the man pages to see what other options are available.

Second, note the *critical* checking of whether the **fopen()** actually succeeded. This is not Java where an exception will be thrown when things goes wrong; rather, it is C, and it is expected (in good programs, i.e., the only kind you'd want to write) that you always will check if the call succeeded. Reading the man page tells you the details of what is returned when an error is encountered; in this case, the macOS man page says:

```
Upon successful completion fopen(), fdopen(), freopen() and fmemopen() return
a FILE pointer.  Otherwise, NULL is returned and the global variable errno is
set to indicate the error.
```

Thus, as the code above does, please check that **fopen()** does not return NULL before trying to use the FILE pointer it returns.

Third, note that when the error case occurs, the program prints a message and then exits with error status of 1. In UNIX systems, it is traditional to return 0 upon success, and non-zero upon failure. Here, we will use 1 to indicate failure.

Side note: if **fopen()** does fail, there are many reasons possible as to why. You can use the functions **perror()** or **strerror()** to print out more about *why* the error occurred; learn about those on your own (using ... you guessed it ... the man pages!).

Once a file is open, there are many different ways to read from it. The one we're suggesting here to you is **fgets()**, which is used to get input from files, one line at a time.

To print out file contents, just use **printf()**. For example, after reading in a line with **fgets()** into a variable **buffer**, you can just print out the buffer as follows:

```
printf("%s", buffer);
```

Note that you should *not* add a newline (\n) character to the printf(), because that would be changing the output of the file to have extra newlines. Just print the exact contents of the read-in buffer (which, of course, many include a newline).

Finally, when you are done reading and printing, use **fclose()** to close the file (thus indicating you no longer need to read from it).

Details

- Your program **seucat** can be invoked with one or more files on the command line; it should just print out each file in turn.
- In all non-error cases, **seucat** should exit with status code 0, usually by returning a 0 from **main()** (or by calling **exit(0)**).
- If *no files* are specified on the command line, **seucat** should just exit and return 0. Note that this is slightly

different than the behavior of normal UNIX **cat** (if you'd like to, figure out the difference).
- If the program tries to **fopen()** a file and fails, it should print the exact message "cannot open file" (followed by a newline) and exit with status code 1. If multiple files are specified on the command line, the files should be printed out in order until the end of the file list is reached or an error opening a file is reached (at which point the error message is printed and **seucat** exits).

seugrep

The second utility you will build is called **seugrep**, a variant of the UNIX tool **grep**. This tool looks through a file, line by line, trying to find a user-specified search term in the line. If a line has the word within it, the line is printed out, otherwise it is not.

Here is how a user would look for the term **foo** in the file **bar.txt**:

```
prompt> ./seugrep foo bar.txt
this line has foo in it
so does this foolish line; do you see where?
even this line, which has barfood in it, will be printed.
```

Details

- Your program **seugrep** is always passed a search term and zero or more files to grep through (thus, more than one is possible). It should go through each line and see if the search term is in it; if so, the line should be printed, and if not, the line should be skipped.
- The matching is case sensitive. Thus, if searching for **foo**, lines with **Foo** will *not* match.
- Lines can be arbitrarily long (that is, you may see many many characters before you encounter a newline character, \n). **seugrep** should work as expected even with very long lines. For this, you might want to look into the **getline()** library call (instead of **fgets()**), or roll your own.
- If **seugrep** is passed no command-line arguments, it should print "searchterm [file ...]" (followed by a newline) and exit with status 1.
- If **seugrep** encounters a file that it cannot open, it should print "cannot open file" (followed by a newline) and exit with status 1.
- In all other cases, **seugrep** should exit with return code 0.
- If a search term, but no file, is specified, **seugrep** should work, but instead of reading from a file, **seugrep** should read from *standard input*. Doing so is easy, because the file stream **stdin** is already open; you can use **fgets()** (or similar routines) to read from it.
- For simplicity, if passed the empty string as a search string, **seugrep** can either match NO lines or match ALL lines, both are acceptable.

seuzip and seuunzip

The next tools you will build come in a pair, because one (**seuzip**) is a file compression tool, and the other (**seuunzip**) is a file decompression tool.

The type of compression used here is a simple form of compression called *run-length encoding* (*RLE*). RLE is quite simple: when you encounter **n** characters of the same type in a row, the compression tool (**seuzip**) will turn that into the number **n** and a single instance of the character.

Thus, if we had a file with the following contents:

```
aaaaaaaaaabbbb
```

the tool would turn it (logically) into:

```
10a4b
```

However, the exact format of the compressed file is quite important; here, you will write out a 4-byte integer in binary format followed by the single character in ASCII. Thus, a compressed file will consist of some number of 5-byte entries, each of which is comprised of a 4-byte integer (the run length) and the single character.

To write out an integer in binary format (not ASCII), you should use **fwrite()**. Read the man page for more details. For **seuzip**, all output should be written to standard output (the **stdout** file stream, which, as with **stdin**, is already open when the program starts running).

Note that typical usage of the **seuzip** tool would thus use shell redirection in order to write the compressed output to a file. For example, to compress the file **file.txt** into a (hopefully smaller) **file.z**, you would type:

```
prompt> ./seuzip file.txt > file.z
```

The "greater than" sign is a UNIX shell redirection; in this case, it ensures that the output from **seuzip** is written to the file **file.z** (instead of being printed to the screen). You'll learn more about how this works a little later in the course.

The **seuunzip** tool simply does the reverse of the **seuzip** tool, taking in a compressed file and writing (to standard output again) the uncompressed results. For example, to see the contents of **file.txt**, you would type:

```
prompt> ./seuunzip file.z
```

seuunzip should read in the compressed file (likely using **fread()**) and print out the uncompressed output to standard output using **printf()**.

Details

- Correct invocation should pass one or more files via the command line to the program; if no files are specified, the program should exit with return code 1 and print "seuzip: file1 [file2 ...]" (followed by a newline) or "seuunzip: file1 [file2 ...]" (followed by a newline) for **seuzip** and **seuunzip** respectively.
- The format of the compressed file must match the description above exactly (a 4-byte integer followed by a character for each run).
- Do note that if multiple files are passed to **seuzip**, they are compressed into a single compressed output, and when unzipped, will turn into a single uncompressed stream of text (thus, the information that multiple files were originally input into **seuzip** is lost). The same thing holds for **seuunzip**.

Intro To Kernel Hacking

To develop a better sense of how an operating system works, you will also do a few projects *inside* a real OS kernel. The kernel we'll be using is a port of the original Unix (version 6), and is runnable on modern x86 processors. It was developed at MIT and is a small and relatively understandable OS and thus an excellent focus for simple projects.

This first project is just a warmup, and thus relatively light on work. The goal of the project is simple: to add a system call to xv6. Your system call, **getreadcount()**, simply returns how many times that the **read()** system call has been called by user processes since the time that the kernel was booted.

Your System Call

Your new system call should look have the following return codes and parameters:

```
int getreadcount(void)
```

Your system call returns the value of a counter (perhaps called **readcount** or something like that) which is incremented every time any process calls the **read()** system call. That's it!

Tips

Before hacking xv6, you need to succesfully unpack it, build it, and then modify it to make this project successful. Xv6+Qemu is strongly recommended, you can get start by following this small guide:

1. Get the latest xv6 source.

   ```
   prompt> git clone git://github.com/mit-pdos/xv6-public.git
   ```

2. Test your compiler toolchain.

   ```
   prompt> objdump -i
   ```

   The second line should say elf32-i386.

   ```
   prompt> gcc -m32 -print-libgcc-file-name
   ```

   The command should print something like /usr/lib/gcc/i486-linux-gnu/version/libgcc.a or /usr/lib/gcc/x86_64-linux-gnu/version/32/libgcc.a
   If both these commands succeed, you're all set, and don't need to compile your own toolchain. Otherwise, trouble shooting.
3. Compile xv6.

   ```
   prompt> ./Xv6-master/make
   ```

4. Install qemu.

   ```
   prompt> sudo apt-get install qemu
   ```

5. Run xv6.

   ```
   prompt> ./Xv6-master/make qemu
   ```

   If success, you can run **ls** to see all command files.

You might want to read background.html first. More information about xv6, including a very useful book written by the MIT folks who built xv6, is available here.

One good way to start hacking inside a large code base is to find something similar to what you want to do and to carefully copy/modify that. Here, you should find some other system call, like **getpid()** (or any other simple call). Copy it in all the ways you think are needed, and then modify it to do what you need.

Most of the time will be spent on understanding the code. There shouldn't be a whole lot of code added.

Using gdb (the debugger) may be helpful in understanding code, doing code traces, and is helpful for later projects too. Get familiar with this fine tool!

Running Tests

Before running tests, get tcl, tk, expect installed in your working operating system:

```
prompt> sudo apt-get install tcl tk expect
```

Running tests for your system call is easy. Just do the following from inside the `initial-xv6` directory:

```
prompt> ./test-getreadcounts.sh
```

If you implemented things correctly, you should get some notification that the tests passed. If not ...

The tests assume that xv6 source code is found in the `src/` subdirectory. If it's not there, the script will complain.

The test script does a one-time clean build of your xv6 source code using a newly generated makefile called `Makefile.test`. You can use this when debugging (assuming you ever make mistakes, that is), e.g.:

```
prompt> cd src/
prompt> make -f Makefile.test qemu-nox
```

You can suppress the repeated building of xv6 in the tests with the `-s` flag. This should make repeated testing faster:

```
prompt> ./test-getreadcounts.sh -s
```

xv6 System Call Background

To be able to implement this project, you'll have to understand a little bit about how xv6 implements system calls. A system call is a protected transfer of control from an application (running in *user mode*) to the OS (running in *kernel mode*). The general approach enables the kernel to maintain control of the machine while generally letting user applications run efficiently and without kernel intervention.

We'll specifically trace what happens in the code in order to understand a *system call*. System calls allow the operating system to run code on the behalf of user requests but in a protected manner, both by jumping into the kernel (in a very specific and restricted way) and also by simultaneously raising the privilege level of the hardware, so that the OS can perform certain restricted operations.

System Call Overview

Before delving into the details, we first provide an overview of the entire process. The problem we are trying to solve is simple: how can we build a system such that the OS is allowed access to all of the resources of the machine (including access to special instructions, to physical memory, and to any devices) while user programs are only able to do so in a restricted manner?

The way we achieve this goal is with hardware support. The hardware must explicitly have a notion of privilege built into it, and thus be able to distinguish when the OS is running versus typical user applications.

Getting Into The Kernel: A Trap

The first step in a system call begins at user-level with an application. The application that wishes to make a system call (such as **read()**) calls the relevant library routine. However, all the library version of the system call does is to place the proper arguments in relevant registers and issue some kind of **trap** instruction, as we see in an expanded version of **usys.S** (Some macros are used to define these functions so as to make life easier for the kernel developer; the example shows the macro expanded to the actual assembly code).

```
.globl read;
read:
  movl $6, %eax;
  int $64;
  ret
```

File: **usys.S**

Here we can see that the **read()** library function actually doesn't do much at all; it moves the value 5 into the register **%eax** and issues the x86 trap instruction which is confusingly called **int** (short for *interrupt*). The value in **%eax** is going to be used by the kernel to *vector* to the right system call, i.e., it determines which system call is being invoked. The **int** instruction takes one argument (here it is 64), which tells the hardware which trap type this is. In xv6, trap 64 is used to handle system calls. Any other arguments which are passed to the system call are passed on the stack.

Kernel Side: Trap Tables

Once the **int** instruction is executed, the hardware takes over and does a bunch of work on behalf of the caller. One important thing the hardware does is to raise the *privilege level* of the CPU to kernel mode; on x86 this is usually means moving from a *CPL (Current Privilege Level)* of 3 (the level at which user applications run) to CPL 0 (in which the kernel runs). Yes, there are a couple of in-between privilege levels, but most systems do not make use of these.

The second important thing the hardware does is to transfer control to the *trap vectors* of the system. To enable the hardware to know what code to run when a particular trap occurs, the OS, when booting, must make sure to inform the hardware of the location of the code to run when such traps take place. This is done in **main.c** as follows:

```
int
mainc(void)
{
   ...
   tvinit(); // trap vectors initialized here
   ...
}
```

FILE: **main.c**

The routine **tvinit()** is the relevant one here. Peeking inside of it, we see:

```
void tvinit(void)
{
  int i;

  for(i = 0; i < 256; i++)
    SETGATE(idt[i], 0, SEG_KCODE<<3, vectors[i], 0);

  // this is the line we care about...
  SETGATE(idt[T_SYSCALL], 1, SEG_KCODE<<3, vectors[T_SYSCALL], DPL_USER);

  initlock(&tickslock, "time");
}
```

FILE: **trap.c**

The **SETGATE()** macro is the relevant code here. It is used to set the **idt** array to point to the proper code to execute when various traps and interrupts occur. For system calls, the single **SETGATE()** call (which comes after the loop) is the one we're interested in. Here is what the macro does (as well as the gate descriptor it sets):

```
// Interrupt descriptor table (shared by all CPUs).
struct gatedesc idt[256];
...

// Gate descriptors for interrupts and traps
struct gatedesc {
  uint off_15_0 : 16;   // low 16 bits of offset in segment
  uint cs : 16;         // code segment selector
  uint args : 5;        // # args, 0 for interrupt/trap gates
  uint rsv1 : 3;        // reserved(should be zero I guess)
  uint type : 4;        // type(STS_{TG,IG32,TG32})
  uint s : 1;           // must be 0 (system)
  uint dpl : 2;         // descriptor(meaning new) privilege level
  uint p : 1;           // Present
  uint off_31_16 : 16;  // high bits of offset in segment
};

// Set up a normal interrupt/trap gate descriptor.
// - istrap: 1 for a trap (= exception) gate, 0 for an interrupt gate.
//   interrupt gate clears FL_IF, trap gate leaves FL_IF alone
// - sel: Code segment selector for interrupt/trap handler
// - off: Offset in code segment for interrupt/trap handler
// - dpl: Descriptor Privilege Level -
//        the privilege level required for software to invoke
//        this interrupt/trap gate explicitly using an int instruction.
#define SETGATE(gate, istrap, sel, off, d)                \
{                                                         \
  (gate).off_15_0 = (uint) (off) & 0xffff;                \
  (gate).cs = (sel);                                      \
  (gate).args = 0;                                        \
  (gate).rsv1 = 0;                                        \
  (gate).type = (istrap) ? STS_TG32 : STS_IG32;           \
  (gate).s = 0;                                           \
  (gate).dpl = (d);                                       \
  (gate).p = 1;                                           \
  (gate).off_31_16 = (uint) (off) >> 16;                  \
}
```

FILE: **mmu.h**

As you can see from the code, all the **SETGATE()** macros does is set the values of an in-memory data structure. Most important is the **off** parameter, which tells the hardware where the trap handling code is. In the initialization code, the value **vectors[T_SYSCALL]** is passed in; thus, whatever the **vectors** array points to will be the code to run when a system call takes place. There are other details (which are important too); consult an x86 hardware architecture manuals (particularly Chapters 3a and 3b) for more information.

Note, however, that we still have not informed the hardware of this information, but rather filled a data structure. The actual hardware informing occurs a little later in the boot sequence; in xv6, it happens in the routine **mpmain()** in the file **main.c**, which calls **idtinit** in **trap.c**, which calls **lidt()** in the include file **x86.h**:

```
static void
mpmain(void)
{
  idtinit();
  ...

void
idtinit(void)
{
  lidt(idt, sizeof(idt));
}

static inline void
lidt(struct gatedesc *p, int size)
{
  volatile ushort pd[3];

  pd[0] = size-1;
  pd[1] = (uint)p;
  pd[2] = (uint)p >> 16;

  asm volatile("lidt (%0)" : : "r" (pd));
}
```

Here, you can see how (eventually) a single assembly instruction is called to tell the hardware where to find the *interrupt descriptor table (IDT)* in memory. Note this is done in **mpmain()** as each processor in the system must have such a table (they all use the same one of course). Finally, after executing this instruction (which is only possible when the kernel is running, in privileged mode), we are ready to think about what happens when a user application invokes a system call.

```
struct trapframe {
  // registers as pushed by pusha
  uint edi;
  uint esi;
  uint ebp;
  uint oesp;      // useless & ignored
  uint ebx;
  uint edx;
  uint ecx;
  uint eax;

  // rest of trap frame
  ushort es;
  ushort padding1;
  ushort ds;
  ushort padding2;
  uint trapno;

  // below here defined by x86 hardware
  uint err;
  uint eip;
  ushort cs;
  ushort padding3;
  uint eflags;

  // below here only when crossing rings, such as from user to kernel
  uint esp;
  ushort ss;
  ushort padding4;
};
```

File: **x86.h**

From Low-level To The C Trap Handler

The OS has carefully set up its trap handlers, and thus we are ready to see what happens on the OS side once an application issues a system call via the **int** instruction. Before any code is run, the hardware must perform a number of tasks. The first thing it does are those tasks which are difficult/impossible for the software to do itself, including saving the current PC (IP or EIP in Intel terminology) onto the stack, as well as a number of other registers such as the **eflags** register (which contains the current status of the CPU while the program was running), stack pointer, and so forth. One can see what the hardware is expected to save by looking at the **trapframe** structure as defined in

x86.h.

As you can see from the bottom of the trapframe structure, some pieces of the trap frame are filled in by the hardware (up to the **err** field); the rest will be saved by the OS. The first code OS that is run is **vector64()** as found in **vectors.S** (which is automatically generated by the script **vectors.pl**).

```
.globl vector64
vector64:
  pushl $64
  jmp alltraps
```

File: **vectors.S** (generated by **vectors.pl**)

This code pushes the trap number onto the stack (filling in the **trapno** field of the trap frame) and then calls **alltraps()** to do most of the saving of context into the trap frame.

```
  # vectors.S sends all traps here.
.globl alltraps
alltraps:
  # Build trap frame.
  pushl %ds
  pushl %es
  pushal

  # Set up data segments.
  movl $SEG_KDATA_SEL, %eax
  movw %ax,%ds
  movw %ax,%es

  # Call trap(tf), where tf=%esp
  pushl %esp
  call trap
  addl $4, %esp
```

File: **trapasm.S**

The code in **alltraps()** pushes a few more segment registers (not described here, yet) onto the stack before pushing the remaining general purpose registers onto the trap frame via a **pushal** instruction. Then, the OS changes the descriptor segment and extra segment registers so that it can access its own (kernel) memory. Finally, the C trap handler is called.

The C Trap Handler

Once done with the low-level details of setting up the trap frame, the low-level assembly code calls up into a generic C trap handler called **trap()**, which is passed a pointer to the trap frame. This trap handler is called upon all types of interrupts and traps, and thus check the trap number field of the trap frame (**trapno**) to determine what to do. The first check is for the system call trap number (**T_SYSCALL**, or 64 as defined somewhat arbitrarily in **traps.h**), which then handles the system call, as you see here:

```
void
trap(struct trapframe *tf)
{
  if(tf->trapno == T_SYSCALL){
    if(cp->killed)
      exit();
    cp->tf = tf;
    syscall();
    if(cp->killed)
      exit();
    return;
  }
  ... // continues
}
```

FILE: **trap.c**

The code isn't too complicated. It checks if the current process (that made the system call) has been killed; if so, it simply exits and cleans up the process (and thus does not proceed with the system call). It then calls **syscall()** to actually perform the system call; more details on that below. Finally, it checks whether the process has been killed

again before returning. Note that we'll follow the return path below in more detail.

```
static int (*syscalls[])(void) = {
[SYS_chdir]   sys_chdir,
[SYS_close]   sys_close,
[SYS_dup]     sys_dup,
[SYS_exec]    sys_exec,
[SYS_exit]    sys_exit,
[SYS_fork]    sys_fork,
[SYS_fstat]   sys_fstat,
[SYS_getpid]  sys_getpid,
[SYS_kill]    sys_kill,
[SYS_link]    sys_link,
[SYS_mkdir]   sys_mkdir,
[SYS_mknod]   sys_mknod,
[SYS_open]    sys_open,
[SYS_pipe]    sys_pipe,
[SYS_read]    sys_read,
[SYS_sbrk]    sys_sbrk,
[SYS_sleep]   sys_sleep,
[SYS_unlink]  sys_unlink,
[SYS_wait]    sys_wait,
[SYS_write]   sys_write,
};

void
syscall(void)
{
  int num;

  num = cp->tf->eax;
  if(num >= 0 && num < NELEM(syscalls) && syscalls[num])
    cp->tf->eax = syscalls[num]();
  else {
    cprintf("%d %s: unknown sys call %d\n",
        cp->pid, cp->name, num);
    cp->tf->eax = -1;
  }
}
]
```

File: **syscall.c**

Vectoring To The System Call

Once we finally get to the **syscall()** routine in **syscall.c**, not much work is left to do (see above). The system call number has been passed to us in the register **%eax**, and now we unpack that number from the trap frame and use it to call the appropriate routine as defined in the system call table **syscalls[]**. Pretty much all operating systems have a table similar to this to define the various system calls they support. After carefully checking that the system call number is in bounds, the pointed-to routine is called to handle the call. For example, if the system call **read()** was called by the user, the routine **sys_read()** will be invoked here. The return value, you might note, is stored in **%eax** to pass back to the user.

The Return Path

The return path is pretty easy. First, the system call returns an integer value, which the code in **syscall()** grabs and places into the **%eax** field of the trap frame. The code then returns into **trap()**, which simply returns into where it was called from in the assembly trap handler.

```
  # Return falls through to trapret...
.globl trapret
trapret:
  popal
  popl %es
  popl %ds
  addl $0x8, %esp  # trapno and errcode
  iret
```

File: **trapasm.S**

This return code doesn't do too much, just making sure to pop the relevant values off the stack to restore the context of the running process. Finally, one more special instruction is called: **iret**, or the **return-from-trap** instruction.

This instruction is similar to a return from a procedure call, but simultaneously lowers the privilege level back to user mode and jumps back to the instruction immediately following the **int** instruction called to invoke the system call, restoring all the state that has been saved into the trap frame. At this point, the user stub for **read()** (as seen in the **usys.S** code) is run again, which just uses a normal return-from-procedure-call instruction (**ret**) in order to return to the caller.

Summary

We have seen the path in and out of the kernel on a system call. As you can tell, it is much more complex than a simple procedure call, and requires a careful protocol on behalf of the OS and hardware to ensure that application state is properly saved and restored on entry and return. As always, the concept is easy: with operating systems, the devil is always in the details.

Unix Shell

In this project, you'll build a simple Unix shell. The shell is the heart of the command-line interface, and thus is central to the Unix/C programming environment. Mastering use of the shell is necessary to become proficient in this world; knowing how the shell itself is built is the focus of this project.

There are three specific objectives to this assignment:

- To further familiarize yourself with the Linux programming environment.
- To learn how processes are created, destroyed, and managed.
- To gain exposure to the necessary functionality in shells.

Overview

In this assignment, you will implement a *command line interpreter (CLI)* or, as it is more commonly known, a *shell*. The shell should operate in this basic way: when you type in a command (in response to its prompt), the shell creates a child process that executes the command you entered and then prompts for more user input when it has finished.

The shells you implement will be similar to, but simpler than, the one you run every day in Unix. If you don't know what shell you are running, it's probably `bash`. One thing you should do on your own time is learn more about your shell, by reading the man pages or other online materials.

Program Specifications

Basic Shell: `seush`

Your basic shell, called `seush` (short for Southeast University Shell, naturally), is basically an interactive loop: it repeatedly prints a prompt `seush>` (note the space after the greater-than sign), parses the input, executes the command specified on that line of input, and waits for the command to finish. This is repeated until the user types `exit`. The name of your final executable should be `seush`.

The shell can be invoked with either no arguments or a single argument; anything else is an error. Here is the no-argument way:

```
prompt> ./seush
seush>
```

At this point, `seush` is running, and ready to accept commands. Type away!

The mode above is called *interactive* mode, and allows the user to type commands directly. The shell also supports a *batch mode*, which instead reads input from a batch file and executes commands from therein. Here is how you run the shell with a batch file named `batch.txt`:

```
prompt> ./seush batch.txt
```

One difference between batch and interactive modes: in interactive mode, a prompt is printed (`seush>`). In batch mode, no prompt should be printed.

You should structure your shell such that it creates a process for each new command (the exception are *built-in commands*, discussed below). Your basic shell should be able to parse a command and run the program corresponding to the command. For example, if the user types `ls -la /tmp`, your shell should run the program `/bin/ls` with the given arguments `-la` and `/tmp` (how does the shell know to run `/bin/ls`? It's something called the shell **path**; more on this below).

Structure

Basic Shell

The shell is very simple (conceptually): it runs in a while loop, repeatedly asking for input to tell it what command to execute. It then executes that command. The loop continues indefinitely, until the user types the built-in command `exit`, at which point it exits. That's it!

For reading lines of input, you should use `getline()`. This allows you to obtain arbitrarily long input lines with ease. Generally, the shell will be run in *interactive mode*, where the user types a command (one at a time) and the shell acts on it. However, your shell will also support *batch mode*, in which the shell is given an input file of commands; in this case, the shell should not read user input (from `stdin`) but rather from this file to get the commands to execute.

In either mode, if you hit the end-of-file marker (EOF), you should call `exit(0)` and exit gracefully.

To parse the input line into constituent pieces, you might want to use `strsep()`. Read the man page (carefully) for more details.

To execute commands, look into `fork()`, `exec()`, and `wait()/waitpid()`. See the man pages for these functions.

You will note that there are a variety of commands in the `exec` family; for this project, you must use `execv`. You should **not** use the `system()` library function call to run a command. Remember that if `execv()` is successful, it will not return; if it does return, there was an error (e.g., the command does not exist). The most challenging part is getting the arguments correctly specified.

Paths

In our example above, the user typed `ls` but the shell knew to execute the program `/bin/ls`. How does your shell know this?

It turns out that the user must specify a **path** variable to describe the set of directories to search for executables; the set of directories that comprise the path are sometimes called the *search path* of the shell. The path variable contains the list of all directories to search, in order, when the user types a command.

Important: Note that the shell itself does not *implement* `ls` or other commands (except built-ins). All it does is find those executables in one of the directories specified by `path` and create a new process to run them.

To check if a particular file exists in a directory and is executable, consider the `access()` system call. For example, when the user types `ls`, and path is set to include both `/bin` and `/usr/bin`, try `access("/bin/ls", X_OK)`. If that fails, try "/usr/bin/ls". If that fails too, it is an error.

Your initial shell path should contain one directory: `/bin'

Note: Most shells allow you to specify a binary specifically without using a search path, using either **absolute paths** or **relative paths**. For example, a user could type the **absolute path** `/bin/ls` and execute the `ls` binary without a search path being needed. A user could also specify a **relative path** which starts with the current working directory and specifies the executable directly, e.g., `./main`. In this project, you **do not** have to worry about these features.

Built-in Commands

Whenever your shell accepts a command, it should check whether the command is a **built-in command** or not. If it is, it should not be executed like other programs. Instead, your shell will invoke your implementation of the built-in command. For example, to implement the `exit` built-in command, you simply call `exit(0);` in your wish source code, which then will exit the shell.

In this project, you should implement `exit`, `cd`, and `path` as built-in commands.

- `exit`: When the user types `exit`, your shell should simply call the `exit` system call with 0 as a parameter. It is an error to pass any arguments to `exit`.

- `cd`: `cd` always take one argument (0 or >1 args should be signaled as an error). To change directories, use the `chdir()` system call with the argument supplied by the user; if `chdir` fails, that is also an error.

- `path`: The `path` command takes 0 or more arguments, with each argument separated by whitespace from the others. A typical usage would be like this: `wish> path /bin /usr/bin`, which would add `/bin` and `/usr/bin` to the search path of the shell. If the user sets path to be empty, then the shell should not be able to run any programs (except built-in commands). The `path` command always overwrites the old path with the newly specified path.

Redirection

Many times, a shell user prefers to send the output of a program to a file rather than to the screen. Usually, a shell provides this nice feature with the > character. Formally this is named as redirection of standard output. To make your shell users happy, your shell should also include this feature, but with a slight twist (explained below).

For example, if a user types `ls -la /tmp > output`, nothing should be printed on the screen. Instead, the standard output of the `ls` program should be rerouted to the file `output`. In addition, the standard error output of the program should be rerouted to the file `output` (the twist is that this is a little different than standard redirection).

If the `output` file exists before you run your program, you should simple overwrite it (after truncating it).

The exact format of redirection is a command (and possibly some arguments) followed by the redirection symbol followed by a filename. Multiple redirection operators or multiple files to the right of the redirection sign are errors.

Note: don't worry about redirection for built-in commands (e.g., we will not test what happens when you type `path /bin > file`).

Parallel Commands

Your shell will also allow the user to launch parallel commands. This is accomplished with the ampersand operator as follows:

```
wish> cmd1 & cmd2 args1 args2 & cmd3 args1
```

In this case, instead of running `cmd1` and then waiting for it to finish, your shell should run `cmd1`, `cmd2`, and `cmd3` (each with whatever arguments the user has passed to it) in parallel, *before* waiting for any of them to complete.

Then, after starting all such processes, you must make sure to use `wait()` (or `waitpid`) to wait for them to complete. After all processes are done, return control to the user as usual (or, if in batch mode, move on to the next line).

Program Errors

The one and only error message. You should print this one and only error message whenever you encounter an error of any type:

```
char error_message[30] = "An error has occurred\n";
write(STDERR_FILENO, error_message, strlen(error_message));
```

The error message should be printed to stderr (standard error), as shown above.

After ~~*any*~~ most errors, your shell simply *continue processing* after printing the one and only error message. However, if the shell is invoked with more than one file, or if the shell is passed a bad batch file, it should exit by calling `exit(1)`.

There is a difference between errors that your shell catches and those that the program catches. Your shell should catch all the syntax errors specified in this project page. If the syntax of the command looks perfect, you simply run the specified program. If there are any program-related errors (e.g., invalid arguments to `ls` when you run it, for example), the shell does not have to worry about that (rather, the program will print its own error messages and exit).

Miscellaneous Hints

Remember to get the **basic functionality** of your shell working before worrying about all of the error conditions and end cases. For example, first get a single command running (probably first a command with no arguments, such as `ls`).

Next, add built-in commands. Then, try working on redirection. Finally, think about parallel commands. Each of these requires a little more effort on parsing, but each should not be too hard to implement.

At some point, you should make sure your code is robust to white space of various kinds, including spaces and tabs. In general, the user should be able to put variable amounts of white space before and after commands, arguments, and various operators; however, the operators (redirection and parallel commands) do not require whitespace.

Check the return codes of all system calls from the very beginning of your work. This will often catch errors in how you are invoking these new system calls. It's also just good programming sense.

Beat up your own code! You are the best (and in this case, the only) tester of this code. Throw lots of different inputs at it and make sure the shell behaves well. Good code comes through testing; you must run many different tests to make sure things work as desired. Don't be gentle -- other users certainly won't be.

Finally, keep versions of your code. More advanced programmers will use a source control system such as git. Minimally, when you get a piece of functionality working, make a copy of your .c file (perhaps a subdirectory with a version number, such as v1, v2, etc.). By keeping older, working versions around, you can comfortably work on adding new functionality, safe in the knowledge you can always go back to an older, working version if need be.

Kernel Threads

In this project, you'll be adding real kernel threads to xv6. Sound like fun? Well, it should. Because you are on your way to becoming a real kernel hacker. And what could be more fun than that?

Specifically, you'll do three things. First, you'll define a new system call to create a kernel thread, called `clone()`, as well as one to wait for a thread called `join()`. Then, you'll use `clone()` to build a little thread library, with a `thread_create()` call and `lock_acquire()` and `lock_release()` functions. That's it! And now, for some details.

Overview

Your new clone system call should look like this: `int clone(void(*fcn)(void *, void *), void *arg1, void *arg2, void *stack)`. This call creates a new kernel thread which shares the calling process's address space. File descriptors are copied as in `fork()`. The new process uses `stack` as its user stack, which is passed two arguments (`arg1` and `arg2`) and uses a fake return PC (`0xffffffff`); a proper thread will simply call `exit()` when it is done (and not `return`). The stack should be one page in size and page-aligned. The new thread starts executing at the address specified by `fcn`. As with `fork()`, the PID of the new thread is returned to the parent (for simplicity, threads each have their own process ID).

The other new system call is `int join(void **stack)`. This call waits for a child thread that shares the address space with the calling process to exit. It returns the PID of waited-for child or -1 if none. The location of the child's user stack is copied into the argument `stack` (which can then be freed).

You also need to think about the semantics of a couple of existing system calls. For example, `int wait()` should wait for a child process that does not share the address space with this process. It should also free the address space if this is last reference to it. Also, `exit()` should work as before but for both processes and threads; little change is required here.

Your thread library will be built on top of this, and just have a simple `int thread_create(void (*start_routine)(void *, void *), void *arg1, void *arg2)` routine. This routine should call `malloc()` to create a new user stack, use `clone()` to create the child thread and get it running. It returns the newly created PID to the parent and 0 to the child (if successful), -1 otherwise. An `int thread_join()` call should also be created, which calls the underlying `join()` system call, frees the user stack, and then returns. It returns the waited-for PID (when successful), -1 otherwise.

Your thread library should also have a simple *ticket lock* (read this book chapter for more information on this). There should be a type `lock_t` that one uses to declare a lock, and two routines `void lock_acquire(lock_t *)` and `void lock_release(lock_t *)`, which acquire and release the lock. The spin lock should use x86 atomic add to build the lock -- see this wikipedia page for a way to create an atomic fetch-and-add routine using the x86 `xaddl` instruction. One last routine, `void lock_init(lock_t *)`, is used to initialize the lock as need be (it should only be called by one thread).

The thread library should be available as part of every program that runs in xv6. Thus, you should add prototypes to `user/user.h` and the actual code to implement the library routines in `user/ulib.c`.

One thing you need to be careful with is when an address space is grown by a thread in a multi-threaded process (for example, when `malloc()` is called, it may call `sbrk` to grow the address space of the process). Trace this code path carefully and see where a new lock is needed and what else needs to be updated to grow an address space in a multi-threaded process correctly.

Building `clone()` from `fork()`

To implement `clone()`, you should study (and mostly copy) the `fork()` system call. The `fork()` system call will serve as a template for `clone()`, with some modifications. For example, in `kernel/proc.c`, we see the beginning of the `fork()` implementation:

```
int
fork(void)
{
  int i, pid;
  struct proc *np;

  // Allocate process.
  if((np = allocproc()) == 0)
    return -1;

  // Copy process state from p.
  if((np->pgdir = copyuvm(proc->pgdir, proc->sz)) == 0){
    kfree(np->kstack);
    np->kstack = 0;
    np->state = UNUSED;
    return -1;
  }
  np->sz = proc->sz;
  np->parent = proc;
  *np->tf = *proc->tf;
```

This code does some work you need to have done for `clone()`, for example, calling `allocproc()` to allocate a slot in the process table, creating a kernel stack for the new thread, etc.

However, as you can see, the next thing `fork()` does is copy the address space and point the page directory (`np->pgdir`) to a new page table for that address space. When creating a thread (as `clone()` does), you'll want the new child thread to be in the *same* address space as the parent; thus, there is no need to create a copy of the address space, and the new thread's `np->pgdir` should be the same as the parent's -- they now share the address space, and thus have the same page table.

Once that part is complete, there is a little more effort you'll have to apply inside `clone()` to make it work. Specifically, you'll have to set up the kernel stack so that when `clone()` returns in the child (i.e., in the newly created thread), it runs on the user stack passed into clone (`stack`), that the function `fcn` is the starting point of the child thread, and that the arguments `arg1` and `arg2` are available to that function. This will be a little work on your part to figure out; have fun!

x86 Calling Convention

One other thing you'll have to understand to make this all work is the x86 calling convention, and exactly how the stack works when calling a function. This is you can read about in Programming From The Ground Up, a free online book. Specifically, you should understand Chapter 4 (and maybe Chapter 3) and the details of call/return. All of this will be useful in getting `clone()` above to set things up properly on the user stack of the child thread.

An xv6 Lottery Scheduler

In this project, you'll be putting a new scheduler into xv6. It is called a **lottery scheduler**, and the full version is described in this chapter of the online book; you'll be building a simpler one. The basic idea is simple: assign each running process a slice of the processor based in proportion to the number of tickets it has; the more tickets a process has, the more it runs. Each time slice, a randomized lottery determines the winner of the lottery; that winning process is the one that runs for that time slice.

The objectives for this project: * To gain further knowledge of a real kernel, xv6. * To familiarize yourself with a scheduler. * To change that scheduler to a new algorithm. * To make a graph to show your project behaves appropriately.

Details

You'll need two new system calls to implement this scheduler. The first is `int settickets(int number)`, which sets the number of tickets of the calling process. By default, each process should get one ticket; calling this routine makes it such that a process can raise the number of tickets it receives, and thus receive a higher proportion of CPU cycles. This routine should return 0 if successful, and -1 otherwise (if, for example, the caller passes in a number less than one).

The second is `int getpinfo(struct pstat *)`. This routine returns some information about all running processes, including how many times each has been chosen to run and the process ID of each. You can use this system call to build a variant of the command line program `ps`, which can then be called to see what is going on. The structure `pstat` is defined below; note, you cannot change this structure, and must use it exactly as is. This routine should return 0 if successful, and -1 otherwise (if, for example, a bad or NULL pointer is passed into the kernel).

Most of the code for the scheduler is quite localized and can be found in `proc.c`; the associated header file, `proc.h` is also quite useful to examine. To change the scheduler, not much needs to be done; study its control flow and then try some small changes.

You'll need to assign tickets to a process when it is created. Specfically, you'll need to make sure a child process *inherits* the same number of tickets as its parents. Thus, if the parent has 10 tickets, and calls **fork()** to create a child process, the child should also get 10 tickets.

You'll also need to figure out how to generate random numbers in the kernel; some searching should lead you to a simple pseudo-random number generator, which you can then include in the kernel and use as appropriate.

Finally, you'll need to understand how to fill in the structure `pstat` in the kernel and pass the results to user space. The structure should look like what you see here, in a file you'll have to include called `pstat.h`:

```
#ifndef _PSTAT_H_
#define _PSTAT_H_

#include "param.h"

struct pstat {
  int inuse[NPROC];   // whether this slot of the process table is in use (1 or 0)
  int tickets[NPROC]; // the number of tickets this process has
  int pid[NPROC];     // the PID of each process
  int ticks[NPROC];   // the number of ticks each process has accumulated
};

#endif // _PSTAT_H_
```

Good examples of how to pass arguments into the kernel are found in existing system calls. In particular, follow the path of `read()`, which will lead you to `sys_read()`, which will show you how to use `argptr()` (and related calls) to obtain a pointer that has been passed into the kernel. Note how careful the kernel is with pointers passed from user space -- they are a security threat(!), and thus must be checked very carefully before usage.

Graph

Beyond the usual code, you'll have to make a graph for this assignment. The graph should show the number of time slices a set of three processes receives over time, where the processes have a 3:2:1 ratio of tickets (e.g., process A might have 30 tickets, process B 20, and process C 10). The graph is likely to be pretty boring, but should clearly show that your lottery scheduler works as desired.

Map Reduce

In 2004, engineers at Google introduced a new paradigm for large-scale parallel data processing known as MapReduce. One key aspect of MapReduce is that it makes programming such tasks on large-scale clusters easy for developers; instead of worrying about how to manage parallelism, handle machine crashes, and many other complexities common within clusters of machines, the developer can instead just focus on writing little bits of code (described below) and the infrastructure handles the rest.

In this project, you'll be building a simplified version of MapReduce for just a single machine. While somewhat easier to build MapReduce for a single machine, there are still numerous challenges, mostly in building the correct concurrency support. Thus, you'll have to think a bit about how to build the MapReduce implementation, and then build it to work efficiently and correctly.

There are three specific objectives to this assignment:

- To learn about the general nature of the MapReduce paradigm.
- To implement a correct and efficient MapReduce framework using threads and related functions.
- To gain more experience writing concurrent code.

General Idea

Let's now get into the exact code you'll have to build. The MapReduce infrastructure you will build supports the execution of user-defined `Map()` and `Reduce()` functions.

As from the original paper: "`Map()`, written by the user, takes an input pair and produces a set of intermediate key/value pairs. The MapReduce library groups together all intermediate values associated with the same intermediate key K and passes them to the `Reduce()` function."

"The `Reduce()` function, also written by the user, accepts an intermediate key K and a set of values for that key. It merges together these values to form a possibly smaller set of values; typically just zero or one output value is produced per `Reduce()` invocation. The intermediate values are supplied to the user's reduce function via an iterator."

A classic example, written here in pseudocode, shows how to count the number of occurrences of each word in a set of documents:

```
map(String key, String value):
    // key: document name
    // value: document contents
    for each word w in value:
        EmitIntermediate(w, "1");

reduce(String key, Iterator values):
    // key: a word
    // values: a list of counts
    int result = 0;
    for each v in values:
        result += ParseInt(v);
    print key, result;
```

What's fascinating about MapReduce is that so many different kinds of relevant computations can be mapped onto this framework. The original paper lists many examples, including word counting (as above), a distributed grep, a URL frequency access counters, a reverse web-link graph application, a term-vector per host analysis, and others.

What's also quite interesting is how easy it is to parallelize: many mappers can be running at the same time, and later, many reducers can be running at the same time. Users don't have to worry about how to parallelize their application; rather, they just write `Map()` and `Reduce()` functions and the infrastructure does the rest.

Code Overview

We give you here the mapreduce.h header file that specifies exactly what you must build in your MapReduce library:

```
#ifndef __mapreduce_h__
#define __mapreduce_h__

// Different function pointer types used by MR
typedef char *(*Getter)(char *key, int partition_number);
typedef void (*Mapper)(char *file_name);
typedef void (*Reducer)(char *key, Getter get_func, int partition_number);
typedef unsigned long (*Partitioner)(char *key, int num_partitions);

// External functions: these are what you must define
void MR_Emit(char *key, char *value);

unsigned long MR_DefaultHashPartition(char *key, int num_partitions);

void MR_Run(int argc, char *argv[],
        Mapper map, int num_mappers,
        Reducer reduce, int num_reducers,
        Partitioner partition);

#endif // __mapreduce_h__
```

The most important function is `MR_Run`, which takes the command line parameters of a given program, a pointer to a Map function (type `Mapper`, called `map`), the number of mapper threads your library should create (`num_mappers`), a pointer to a Reduce function (type `Reducer`, called `reduce`), the number of reducers (`num_reducers`), and finally, a pointer to a Partition function (`partition`, described below).

Thus, when a user is writing a MapReduce computation with your library, they will implement a Map function, implement a Reduce function, possibly implement a Partition function, and then call `MR_Run()`. The infrastructure will then create threads as appropriate and run the computation.

One basic assumption is that the library will create `num_mappers` threads (in a thread pool) that perform the map tasks. Another is that your library will create `num_reducers` threads to perform the reduction tasks. Finally, your library will create some kind of internal data structure to pass keys and values from mappers to reducers; more on this below.

Simple Example: Wordcount

Here is a simple (but functional) wordcount program, written to use this infrastructure:

```
#include <assert.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "mapreduce.h"

void Map(char *file_name) {
    FILE *fp = fopen(file_name, "r");
    assert(fp != NULL);

    char *line = NULL;
    size_t size = 0;
    while (getline(&line, &size, fp) != -1) {
        char *token, *dummy = line;
        while ((token = strsep(&dummy, " \t\n\r")) != NULL) {
            MR_Emit(token, "1");
        }
    }
    free(line);
    fclose(fp);
}

void Reduce(char *key, Getter get_next, int partition_number) {
    int count = 0;
    char *value;
    while ((value = get_next(key, partition_number)) != NULL)
        count++;
    printf("%s %d\n", key, count);
}

int main(int argc, char *argv[]) {
    MR_Run(argc, argv, Map, 10, Reduce, 10, MR_DefaultHashPartition);
}
```

Let's walk through this code, in order to see what it is doing. First, notice that `Map()` is called with a file name. In general, we assume that this type of computation is being run over many files; each invocation of `Map()` is thus handed one file name and is expected to process that file in its entirety.

In this example, the code above just reads through the file, one line at a time, and uses `strsep()` to chop the line into tokens. Each token is then emitted using the `MR_Emit()` function, which takes two strings as input: a key and a value. The key here is the word itself, and the token is just a count, in this case, 1 (as a string). It then closes the file.

The `MR_Emit()` function is thus another key part of your library; it needs to take key/value pairs from the many different mappers and store them in a way that later reducers can access them, given constraints described below. Designing and implementing this data structure is thus a central challenge of the project.

After the mappers are finished, your library should have stored the key/value pairs in such a way that the `Reduce()` function can be called. `Reduce()` is invoked once per key, and is passed the key along with a function that enables iteration over all of the values that produced that same key. To iterate, the code just calls `get_next()` repeatedly until a NULL value is returned; `get_next` returns a pointer to the value passed in by the `MR_Emit()` function above, or NULL when the key's values have been processed. The output, in the example, is just a count of how many times a given word has appeared, and is just printed to standard output.

All of this computation is started off by a call to `MR_Run()` in the `main()` routine of the user program. This function is passed the `argv` array, and assumes that `argv[1]` ... `argv[n-1]` (with `argc` equal to `n`) all contain file names that will be passed to the mappers.

One interesting function that you also need to pass to `MR_Run()` is the partitioning function. In most cases, programs will use the default function (`MR_DefaultHashPartition`), which should be implemented by your code. Here is its implementation:

```
unsigned long MR_DefaultHashPartition(char *key, int num_partitions) {
    unsigned long hash = 5381;
    int c;
    while ((c = *key++) != '\0')
        hash = hash * 33 + c;
    return hash % num_partitions;
}
```

The function's role is to take a given `key` and map it to a number, from `0` to `num_partitions - 1`. Its use is internal to the MapReduce library, but critical. Specifically, your MR library should use this function to decide which partition (and hence, which reducer thread) gets a particular key/list of values to process. For some applications, which reducer thread processes a particular key is not important (and thus the default function above should be passed in to `MR_Run()`); for others, it is, and this is why the user can pass in their own partitioning function as need be.

One last requirement: For each partition, keys (and the value list associated with said keys) should be **sorted** in ascending key order; thus, when a particular reducer thread (and its associated partition) are working, the `Reduce()` function should be called on each key in order for that partition.

Considerations

Here are a few things to consider in your implementation:

- **Thread Management**. This part is fairly straightforward. You should create `num_mappers` mapping threads, and assign a file to each `Map()` invocation in some manner you think is best (e.g., Round Robin, Shortest-File-First, etc.). Which way might lead to best performance? You should also create `num_reducers` reducer threads at some point, to work on the map'd output.

- **Partitioning and Sorting**. Your central data structure should be concurrent, allowing mappers to each put values into different partitions correctly and efficiently. Once the mappers have completed, a sorting phase should order the key/value-lists. Then, finally, each reducer thread should start calling the user-defined `Reduce()` function on the keys in sorted order per partition. You should think about what type of locking is needed throughout this process for correctness.

- **Memory Management**. One last concern is memory management. The `MR_Emit()` function is passed a key/value pair; it is the responsibility of the MR library to make copies of each of these. Then, when the entire mapping and reduction is complete, it is the responsibility of the MR library to free everything.

Grading

Your code should turn in `mapreduce.c` which implements the above functions correctly and efficiently. It will be compiled with test applications with the `-Wall -Werror -pthread -O` flags; it will also be valgrinded to check for memory errors.

Your code will first be measured for correctness, ensuring that it performs the maps and reductions correctly. If you pass the correctness tests, your code will be tested for performance; higher performance will lead to better scores.

Intro To xv6 Virtual Memory

In this project, you'll be changing xv6 to support a feature virtually every modern OS does: causing an exception to occur when your program dereferences a null pointer, and adding the ability to change the protection levels of some pages in a process's address space.

Null-pointer Dereference

In xv6, the VM system uses a simple two-level page table as discussed in class. As it currently is structured, user code is loaded into the very first part of the address space. Thus, if you dereference a null pointer, you will not see an exception (as you might expect); rather, you will see whatever code is the first bit of code in the program that is running. Try it and see!

Thus, the first thing you might do is create a program that dereferences a null pointer. It is simple! See if you can do it. Then run it on Linux as well as xv6, to see the difference.

Your job here will be to figure out how xv6 sets up a page table. Thus, once again, this project is mostly about understanding the code, and not writing very much. Look at how **exec()** works to better understand how address spaces get filled with code and in general initialized.

You should also look at `fork()`, in particular the part where the address space of the child is created by copying the address space of the parent. What needs to change in there?

The rest of your task will be completed by looking through the code to figure out where there are checks or assumptions made about the address space. Think about what happens when you pass a parameter into the kernel, for example; if passing a pointer, the kernel needs to be very careful with it, to ensure you haven't passed it a bad pointer. How does it do this now? Does this code need to change in order to work in your new version of xv6?

One last hint: you'll have to look at the xv6 makefile as well. In there user programs are compiled so as to set their entry point (where the first instruction is) to 0. If you change xv6 to make the first page invalid, clearly the entry point will have to be somewhere else (e.g., the next page, or 0x1000). Thus, something in the makefile will need to change to reflect this as well.

Read-only Code

In most operating systems, code is marked read-only instead of read-write. However, in xv6 this is not the case, so a buggy program could accidentally overwrite its own text. Try it and see!

In this portion of the xv6 project, you'll change the protection bits of parts of the page table to be read-only, thus preventing such over-writes, and also be able to change them back.

To do this, you'll be adding two system calls: `int mprotect(void *addr, int len)` and `int munprotect(void *addr, int len)`.

Calling `mprotect()` should change the protection bits of the page range starting at `addr` and of `len` pages to be read only. Thus, the program could still read the pages in this range after `mprotect()` finishes, but a write to this region should cause a trap (and thus kill the process). The `munprotect()` call does the opposite: sets the region back to both readable and writeable.

Also required: the page protections should be inherited on fork(). Thus, if a process has mprotected some of its pages, when the process calls fork, the OS should copy those protections to the child process.

There are some failure cases to consider: if `addr` is not page aligned, or `addr` points to a region that is not currently a part of the address space, or `len` is less than or equal to zero, return -1 and do not change anything. Otherwise, return 0 upon success.

Hint: after changing a page-table entry, you need to make sure the hardware knows of the change. On 32-bit x86, this is readily accomplished by updating the `CR3` register (what we generically call the *page-table base register* in class). When the hardware sees that you overwrote `CR3` (even with the same value), it guarantees that your PTE updates will be used upon subsequent accesses. The `lcr3()` function will help you in this pursuit.

Handling Illegal Accesses

In both the cases above, you should be able to demonstrate what happens when user code tries to (a) access a null pointer or (b) overwrite an mprotected region of memory. In both cases, xv6 should trap and kill the process (this will happen without too much trouble on your part, if you do the project in a sensible way).

File System Checking

In this assignment, you will be developing a working file system checker. A checker reads in a file system image and makes sure that it is consistent. When it isn't, the checker takes steps to repair the problems it sees; however, you won't be doing any repairs to keep this project a little simpler.

A Basic Checker

For this project, you will use the xv6 file system image as the basic image that you will be reading and checking. The file `include/fs.h` includes the basic structures you need to understand, including the superblock, on disk inode format (`struct dinode`), and directory entry format (`struct dirent`). The tool `tools/mkfs.c` will also be useful to look at, in order to see how an empty file-system image is created.

Much of this project will be puzzling out the exact on-disk format xv6 uses for its simple file system, and then writing checks to see if various parts of that structure are consistent. Thus, reading through `mkfs.c` and the file system code itself will help you understand how xv6 uses the bits in the image to record persistent information.

Your checker should read through the file system image and determine the consistency of a number of things, including the following. When a problem is detected, print the error message (shown below) to **standard error** and exit immediately with **exit code 1** (i.e., call `exit(1)`).

1. Each inode is either unallocated or one of the valid types (`T_FILE`, `T_DIR`, `T_DEV`). If not, print `ERROR: bad inode.`
2. For in-use inodes, each address that is used by inode is valid (points to a valid datablock address within the image). If the direct block is used and is invalid, print `ERROR: bad direct address in inode.`; if the indirect block is in use and is invalid, print `ERROR: bad indirect address in inode.`
3. Root directory exists, its inode number is 1, and the parent of the root directory is itself. If not, print `ERROR: root directory does not exist.`
4. Each directory contains `.` and `..` entries, and the `.` entry points to the directory itself. If not, print `ERROR: directory not properly formatted.`
5. For in-use inodes, each address in use is also marked in use in the bitmap. If not, print `ERROR: address used by inode but marked free in bitmap.`
6. For blocks marked in-use in bitmap, the block should actually be in-use in an inode or indirect block somewhere. If not, print `ERROR: bitmap marks block in use but it is not in use.`
7. For in-use inodes, each direct address in use is only used once. If not, print `ERROR: direct address used more than once.`
8. For in-use inodes, each indirect address in use is only used once. If not, print `ERROR: indirect address used more than once.`
9. For all inodes marked in use, each must be referred to in at least one directory. If not, print `ERROR: inode marked use but not found in a directory.`
10. For each inode number that is referred to in a valid directory, it is actually marked in use. If not, print `ERROR: inode referred to in directory but marked free.`
11. Reference counts (number of links) for regular files match the number of times file is referred to in directories (i.e., hard links work correctly). If not, print `ERROR: bad reference count for file.`
12. No extra links allowed for directories (each directory only appears in one other directory). If not, print `ERROR: directory appears more than once in file system.`

Other Specifications

Your checker program, called `xcheck`, must be invoked exactly as follows:

```
prompt> xcheck file_system_image
```

The image file is a file that contains the file system image. If no image file is provided, you should print the usage error shown below:

```
prompt> xcheck
Usage: xcheck <file_system_image>
```

This output must be printed to standard error and exit with the error code of 1.

If the file system image does not exist, you should print the error `image not found.` to standard error and exit with the error code of 1.

If the checker detects any one of the 12 errors above, it should print the specific error to standard error and exit with error code 1.

If the checker detects none of the problems listed above, it should exit with return code of 0 and not print anything.

Hints

It may be worth looking into using `mmap()` for the project. Like, seriously, use `mmap()` to access the file-system image, it will make your life so much better.

It should be very helpful to read Chapter 6 of the xv6 book here. Note that the version of xv6 we're using does not include the logging feature described in the book; you can safely ignore the parts that pertain to that.

Make sure to look at `fs.img`, which is a file system image created when you make xv6 by the tool mkfs (found in the `tools/` directory of xv6). The output of this tool is the file `fs.img` and it is a consistent file-system image. The tests, of course, will put inconsistencies into this image, but your tool should work over a consistent image as well. Study `mkfs` and its output to begin to make progress on this project.

Contest: A Better Checker

For this project, there is a contest, which will compare checkers that can handle these more challenging condition checks:

1. Each `..` entry in directory refers to the proper parent inode, and parent inode points back to it. If not, print `ERROR: parent directory mismatch.`
2. Every directory traces back to the root directory. (i.e. no loops in the directory tree). If not, print `ERROR: inaccessible directory exists.`

This better checker will also have to do something new: actually repair the image, in one specific case. Specifically, your task will be to repair the "inode marked use but not found in a directory" error.

We will provide you with an xv6 image that has a number of in-use inodes that are not linked by any directory. Your job is to collect these nodes and put them into the `lost_found` directory (which is already in the provided image under the root directory). Real checkers do this in order to preserve files that may be useful but for some reason are not linked into a directory.

To do so, you will need to obtain write access to the file system image in order to modify it. This repair operation of your checker program should only be performed when `-r` flag is specified:

```
prompt> xcheck -r image_to_be_repaired
```

In this repair mode, your program should **not** exit when an error is encountered, but rather continue processing. For simplicity, you can also assume there is no other types of error in the provided image. It should exit only after it has created an entry under the `lost_found` directory for every lost inode.

The contest will be judged based on whether all extra tests are passed. If they are, the winner will be given to the most readable implementation.