When I was going to UCLA for my B.S. and Master’s degrees, I worked for 3 summers as an intern at Burroughs Corporation’s Medium Systems Division in Pasadena, California. During my last summer internship at Burroughs, my job was to write a module for the Master Control Program, affectionately known as the MCP:

http://en.wikipedia.org/wiki/Burroughs_MCP

In this assignment, you will implement a MCP whose job is to run and schedule a workload of other programs. The MCP will read a list of programs to run from a file, start up the programs as processes, and then schedule the processes to run concurrently in a time-sliced manner. In addition, the MCP will monitor the processes and keep track of how the processes are using system resources. For extra credit, you can try to improve the MCP scheduling strategy to give more time to CPU-bound processes and less time to I/O-bound processes.

There are 4 parts to the project, each building on the other. The objective is to give you a good introduction to processes, signals and signal handling, and scheduling. Part 5 is for extra credit.

All coding work must be done in the C programming language, and should be compilable and runnable in a Unix/Linux environment.

Part 1 – MCP Launches the Workload

The goal of Part 1 is to develop the first version of the MCP such that it can launch the workload and get all of the processes running together. MCP v1.0 will perform the following steps:

- Read the program workload from the specified input file. Each line in the file contains the name of the program and its arguments.
- For each program, launch the program to run as a process using the `fork()`, `exec()`, and other system calls; see below. To make things simpler, assume that the programs will run in the environment used to execute the MCP.
- Once all of the programs are running, wait for each program to terminate.
- After all programs have terminated, the MCP will exit.

The launching of each workload program will look something like this in pseudocode:

```c
for i=0, numprograms-1 {
    pid[i] = fork();
    if (!pid) {
        execve(program[i], ...);  
    }
}
for i=0, numprograms-1 {
    wait(pid[i]);
}
```

While this may appear to be simple, there are many, many things that can go wrong. You should spend some time reading the entire man page for all four of these system calls.
Part 2 – MCP Takes Control

Successful completion of Part 1 will give you a basic working MCP. However, if we just wanted to run all the workload programs at the same time, we might as well use a shell script. Rather, our ultimate goal is to schedule the programs in the workload to run in a time-shared manner. Part 2 will take the first step to allow the MCP to gain control for this purpose. Actually, it will take two steps.

Step 1 will implement a way for the MCP to stop all the processes right before they call `exec()` so that the MCP can decide which process to run first. The idea is to have each forked child process to wait for a `SIGUSR1` signal before calling `exec()`. The `sigwait()` system call will be useful here. The MPC parent process will send the `SIGUSR1` to the corresponding forked (MPC) child process. Note, until the forked child process does the `exec()` call, it is running the MPC program code.

Once Step 1 is working, the MPC is in a state (after launching all workload programs) where each workload process is waiting on a `SIGUSR1` signal. The first time a workload process is selected to run by the MPC scheduler, it will be started by the MPC sending the `SIGUSR1` signal.

Step 2 will implement the needed mechanism for the MCP to signal a running process to stop (using the `SIGSTOP` signal) and then to continue it again (using the `SIGCONT` signal). This is the mechanism that the MPC will use on a process after it has been started the first time. Sending a `SIGSTOP` signal to a running process is like running a program at the command line and typing `Ctrl-Z` to suspend (stop) it. Sending a suspended process a `SIGCONT` signal is like foregrounding a suspended job at the command line.

Thus, in Part 2, you will implement Step 1 and 2 to create a MCP v2.0 building on MCP v1.0 in the following way:

- Immediately after each program is launched, the forked MPC child process waits on the `SIGUSR1` signal before calling `exec()`.
- After all of the programs have been launched and are now waiting, the MPC parent process sends each program a `SIGUSR1` signal to wake them up. Each process will then return from the `sigwait()` and call `exec()` to run the workload process.
- After all of the workload programs have been launched and are now executing, the MPC sends each workload process a `SIGSTOP` signal to suspend them.
- After all of the workload processes have been suspended, the MPC sends each program a `SIGCONT` signal to wake them up.
- Again, once all of the processes are back up and running, for each program, wait for it to terminate. After all programs have terminated, the MCP will exit.

MCP 2.0 demonstrates that we can control the suspending and continuing of processes.

Handling asynchronous signaling is far more nuanced than described here: you should spend some time reading the entire man pages for these system calls and referencing online and printed resources (such as the books suggested on the course web page) to gain a better understanding of signals and signal handling.

Part 3 – MCP Schedules Processes

Now that the MCP can stop and continue workload processes, we want to implement a MCP scheduler that will run the processes according to some scheduling policy. The simplest policy is to equally share the processor by giving each process the same amount of time to run (e.g., 1 second). The general situation is that there is 1 workload process executing. After its `time slice` has completed, we want to stop that process.
and start up another **ready** process. The MCP decides which is the next workload process to run, starts the timer, and continues that process.

MCP 2.0 knows how to wake up a process, but we need a way to have it run for only a certain amount of time. Note, if some workload process is running, it is still the case that the MCP is running concurrently with it. Thus, one way to approach the problem is for the MCP to poll the system time to determine when the time slice is expended. This is inefficient. Alternatively, you can set an alarm using the `alarm(2)` system call. This tells the operating system to deliver a `SIGALRM` signal after some specified time. Signal handling is done by registering a signal handling function with the operating system. This `SIGALRM` signal handler is implemented in the MCP. When the signal is delivered, the MCP will be interrupted and the signal handling function will be executed. When it does, the MCP will suspend the running workload process, decide on the next workload process to run, and send it a `SIGCONT` signal, reset the alarm, and continuing with whatever else it is doing.

Your new and improved MCP 3.0 is now a working workload process scheduler. However, you need to take care of a few things. For instance, there is the question of how to determine if a workload process is still executing. At some point (we hope), the workload process is going to terminate. Remember, this workload process is a child process of the MCP. How does the MCP know that the workload process has terminated? In MCP 2.0, we just called `wait()`. Is that sufficient now? Be careful.

### Part 4 – MCP as Big Brother

With MCP 3.0, the workload processes are able to be scheduled to run with each getting an “equal” shared of the processor. Note, MCP 3.0 should be able to work with any set of workload programs it reads in. In particular, we will provide you with some workload to run (to be determined) that will (ideally) give some feedback to you that your MCP 3.0 is working correctly. (Note, you can also write your own simple test programs.) It is also possible to see how the workload execution is proceeding by looking in the `/proc` directory for information on the workload processes.

In Part 4, you will add some functionality to the MCP to gather relevant data from `/proc` that conveys some information about what system resources each workload process is consuming. This should include something about execution time, memory used, and I/O. It is up to you to decide what to look at, analyze, and present. Do not just dump out everything in `/proc` for each process. The objective is to give you some experience with reading, interpreting, and analyzing process information. Your MCP 4.0 should output the analyzed process information periodically as the workload programs are executing. One thought is to do something similar to what the Linux `top()` program does.

### Part 5 (Extra Credit) – Adaptive MCP

When the MCP schedules a workload process to run, it assumes that the process will actually execute the entire time of the time slice. However, suppose that the process is doing I/O, for example, waiting for the user to enter something on the keyboard. In general, workload processes could be doing different things and thus have different execution behaviors. Some processes may be **compute bound** while others may be **I/O bound**. If the MCP knew something about process behavior, it is possible that the time slice could be adjusted for each type of process. For instance, I/O bound processes might be given a little less time and compute bound processes a bit more. By adjusting the time slice, it is possible that the entire workload could run more efficiently.

Part 5 is to implement some form of adjustable scheduler that uses process information to set process-specific time intervals.
System Calls

In this assignment, you will likely want to learn about these system calls:

- `execve(2)`
- `fork(2)`
- `wait(2)`
- `read(2)`
- `write(2)`
- `signal(2)`
- `alarm(2)`
- `kill(2)`
- `exit(2)`

Error Handling

All system call functions that you use will report errors via the return value. As a general rule, if the return value is less than zero, then an error has occurred and `errno` is set accordingly. You must check your error conditions and report errors. To expedite the error checking process, we will allow you to use the `perror(3)` library function. Although you are allowed to use `perror`, it does not mean that you should report errors with voluminous verbosity. Report fully but concisely.

Memory Errors

You are required to check your code for memory errors. This is a non-trivial task, but a very important one. Code that contains memory leaks and memory violations will have marks deducted. Fortunately, the `valgrind` tool can help you clean these issues up. It can be installed inside your Linux VM (or native distro, if that’s how you’re running it). Remember that `valgrind`, while quite useful, is only a tool and not a solution. You must still find and fix any bugs that are located by `valgrind`, but there are no guarantees that it will find every memory error in your code; especially those that rely on user input.

Developing Your Code

The best way to develop your code is in Linux running inside the virtual machine image provided to you. This way, if you crash the system, it is straightforward to restart. This also gives you the benefit of taking snapshots of system state right before you do something potentially risky or hazardous, so that if something goes horribly awry you can easily roll back to a safe state. You may use the Deschutes 100 machines to run the Linux VM image within a Virtualbox environment, or run natively or within a VM on your own personal machine. Importantly, do not use ix for this assignment.

You should use your Bitbucket GIT repositories for keeping track of your programming work. You will also need to turn in your project in this way. As a reference, you can perform the command line steps below to create a new project directory and upload it to your uoregon-cis415 repository. If your
uoregon-cis415 repository is currently empty, then first set up a local uoregon-cis415 directory as follows:

```plaintext
cd /path/to/your/uoregon-cis415
git init
git remote add origin https://username@bitbucket.org/
    username/uoregon-cis415.git
```

Now, you can make a `project1` folder, and start creating and saving files to your remote repository. The following commands show how to do this with an initial test file:

```plaintext
cd /path/to/your/uoregon-cis415/project1
echo "This is a test file." >> testFile.txt
cd /path/to/your/uoregon-cis415
```

```plaintext
git add project1
```

```plaintext
git commit -m "Initial commit of project1"
git push -u origin master
```

The next time you login to your virtual machine, the repository contents can be imported with the `clone` command:

```plaintext
git clone https://username@bitbucket.org/username/uoregon-cis415.git
```

Any subsequent changes or additions can be saved via the set of `add`, `commit`, and `push` commands above.

**Helping Your Classmate**

This is an individual assignment. You all should be reading the manuals, hunting for information, and learning those things that enable you to do the project. However, it is important for everyone to make progress and hopefully obtain the same level of knowledge by the project’s end. If you get stuck, seek out help to get unstuck. Sometimes just having another pair of eyes looking at your code is all you need. It is understood that students have different levels of programming skills. If you can not get help from the TA, it is possible that a classmate can be of assistance.

To motivate classmate interaction, 5% of your grade will be associated with providing assistance to others. This does not mean getting 20 friends and each writing 5% of the project and then everyone turning in the same code. The objective is to be off assistance in some way to help you classmates make progress. Please provide a list of others that you helped and how. Ideally, this would be people who you do not already know.