Project 2

(MCP: Ghost-in-the-shell)
CIS 415 - Operating systems
Fall 2019 - Prof. Allen Malony
Due date: **11:59 pm, Tuesday, Nov 10th, 2020**

Introduction:

The British philosopher, Gilbert Ryle, in *The Concept of Mind* criticizes the classical Cartesian rationalism that the mind is distinct from the body (known as mind-body duality). He argues against the "dogma of the ghost in the machine" as an independent non-material entity, temporarily inhabiting and governing the body. In *The Ghost in the Machine*, Arthur Koestler furthers the position that mind and body are connected and that the personal experience of mind-body duality is emergent from the observation that everything in nature is both a whole and a part. The manga series by Masamune Shirow, *The Ghost in the Shell*, whose title is an homage to Koestler, explores the proposition that human consciousness and individuality *IS* the "ghost" and whether it can exist independently of a physical body, with all that implies. 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. Sadly, Burroughs no longer exists, having merged with Sperry Corporation to form Unisys, but they were a major player in computing systems for a long time. During my last summer internship at Burroughs, my job was to write modules for the ***Master Control Program***, affectionately known as the ***MCP***:

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

The MCP might be considered primitive in contrast to modern operating systems, but it was innovative in many respects, including: the first operating system to manage multiple processors, the first commercial implementation of virtual memory, and the first OS written exclusively in a high-level language. MCP executed "jobs" with each containing one or more tasks. Tasks within a job could run sequentially or in parallel. Logic could be implemented at the job level to control the flow of a job by writing in the MCP's workflow control language (WFL). Once all tasks in a job completed, the job itself was done. In some respects, we might regard these features of MCP as now being provided through shell scripts that access an operating system's services. I often wonder whether the ghosts of the modules I wrote for MCP continue to live in the shells being used today.

Project Details:

In this project, you will implement the ***MCP Ghost in the Shell*** (MCP for short) whose primary job is to launch a pool of sub-processes that will execute commands given in an input file. The MCP will read a list of commands (with arguments) to run from a file, it will then start up and run the process that will execute the command, and then schedule the processes to run concurrently in a time-sliced manner. In addition, the MCP will monitor the processes, keeping track of how the processes are using system resources. There are a total of 4 parts to the project, each building on the other. The objective is to give you a good introduction to processes, signals, signal handling, and scheduling. Each part is a complete program by itself and **must** be saved in a separate *.c file.

Part 1: MCP Launches the Workload

For part 1 of this project, you will be developing the first version of the MCP such that it can launch the workload and get all of the processes running together.

Program Requirements:

Your MCP v1.0 must perform the following steps:

1. Read the program workload from the specified input file (This should work just like the file mode from Project 1). Each line in the file contains the command and its arguments.
2. For each command, your MCP must launch the a separate process to run the command using some variant of the following system calls:
    a. **fork(2)**: http://man7.org/linux/man-pages/man2/fork.2.html
    b. **One of the exec() system calls (see exec)**:
        i. http://man7.org/linux/man-pages/man3/exec.3.html
3. Once all of the processes are running, your MCP must wait for each process to terminate using one of the wait family of system calls (wait/waitpid).
    a. **wait(2)**: http://man7.org/linux/man-pages/man2/waitid.2.html
4. After all processes have terminated, your MCP must exit using the exit() system call.
    a. **exit()**: http://man7.org/linux/man-pages/man2/exit.2.html

Fig. 1 presents the pseudocode for launching processes:

```
1.  while(condition) {
2.      getline(...) //read in the command and it's arguments
3.      if(getline reached the end of the file) {
4.          condition = 0; continue;
5.      }
6.      pid[i] = fork();
7.      if (pid[i] < 0) {
8.       // handle the error appropriately;
9.      }
10.     if (pid[i] == 0) {
11.         exec(command, arguments);
12.         // log error. starting program failed.
13.     exit(-1);
14.     }
15.     i++;
16. }
17. for j=0, j<i, j++ {
18.     waitpid(...); //wait for each thread to close.
19. }
```

Fig. 1: Pseudocode for launching processes

Remarks:

To make things simpler, assume that the programs will run in the environment used to execute the MCP (i.e. the VM). 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 of these system calls. Also, you will need to figure out how to read the commands and their arguments from the "workload" file and get them in the proper form to call exec. These commands can be anything that will run in the same environment as the MCP, meaning the environment in which the MCP is launched. A set of commands are provided to help to evaluate your work, but you should also construct your own. Lastely, the console output of your program has a heavy influence on your final grade so take some time to think about how your program represents what is going on at this stage. What is expected is that your output will be somewhat jumbled. This can be an excellent indicator that your code is working as intended. If you find that your messages are all synchronized then you know that something is going wrong.

Part 2: MCP Controls the Workload

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. This will be completed in two core steps:

Step 1:

Step 1 of Part 2 will implement a way for the MCP to stop all forked (MCC) child processes right before they call exec(). This gives the MCP back control before any of the workload programs are launched. The idea is to have each forked child process wait for a `SIGUSR1` signal before calling exec() with its associated workload program. The sigwait() system call will be useful here. ***Note:*** *that until the forked child process does the exec() call, it is running the same code as it's parent (i.e. MCP's code).* Once Step 1 is working, the MCP is in a state where each child process is waiting on a `SIGUSR1` signal. The first time a workload process is selected to run by the MCP scheduler, it will be started by the MCP sending the `SIGUSR1` signal to it. Once a child process receives the `SIGUSR1` signal, it launches the associated workload program with the exec() call.

Step 2:

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 MCP 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 bringing a suspended job into the foreground at the command line.

Program Requirements:

Thus, in Part 2, you will take your MCP v1.0 and implement both Step 1 and 2 to create MCP v2.0. Your MCP v2.0 **must** do the following:

1. Immediately after each program is created using the **fork()** system call, the forked MCP child process waits until it receives a `SIGUSR1` signal before calling **exec()**.
2. After all of the MCP child processes have been forked and are now waiting, the MCP parent process must send each child process a `SIGUSR1` signal (at the same time) to wake them up. Each child process will then return from the **sigwait()** and call **exec()** to run the workload program.

3. After all of the workload programs have been launched and are now executing, the MCP must send each child process a `SIGSTOP` signal to suspend them.
4. After all of the child processes have been suspended, the MCP must send each child process a `SIGCONT` signal to wake them up.
5. Again, once all of the processes are back up and running, the MCP must wait for each child process to terminate. After all processes have terminated, the MCP will exit and free any allocated memory.

Remarks:

MCP 2.0 demonstrates that we can control the suspending and continuing of processes. You should test out that things are working properly. One way to do this is to create messages that indicate what steps the MCP is presently taking and for which child process. The console output of your program has a heavy influence on your final grade so take some time to think about how your program represents what is going on at this stage.

Again, a set of programs will be provided to evaluate your work, but you should also construct your own. Handling asynchronous signaling is far more nuanced than described here. We recommend that you should spend some time studying to gain a better understanding of the system calls used in this project as well as 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.

Program Requirements:

MCP 2.0 knows how to resume a process, but we need a way to have it run for only a certain amount of time. For Part 3 of this project, you will be upgrading your MCP v2.0 to include process scheduling. *Note: if a child process is running, it is still the case that the MCP is running "concurrently" with it.* Thus, one way to approach the problem of process scheduling is for the MCP to poll the system time to determine when the time slice is expended. This is inefficient and not as accurate as it can be.

Thus, we will be using signal processing to implement a more intelligent way of process scheduling. In general, this is done by setting an alarm using the **alarm(2)** system call. This tells the operating system to deliver a `SIGALRM` signal after some specified time. In general, signal handling is done by registering a signal handling function with the operating system. When the

signal is delivered, the MCP will be interrupted and the signal handling function will be executed. The following must happen after the alarm signal is received by the MCP:

1. The MCP will suspend the currently running workload process using `SIGSTOP`.
2. The MCP decides on the next workload process to run, and sends it a `SIGCONT` signal.
3. The MCP will reset the alarm, and continue with whatever else it was doing.

Remarks:

Your new and improved MCP v3.0 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 v2.0, we just called wait(). Is that sufficient now? Be careful. (***Note:*** *You cannot use the return value of waitpid() to determine if a process has terminated. There are multiple correct answers to this issue. So give it some thought and choose appropriately. Again, be careful.)*

Finally, please think about how to test that your MCP v3.0 is working and provide some demonstration. Having the MCP produce output messages at scheduling points is one way to do this. The console output of your program has a heavy influence on your final grade so take some time to think about how your program represents what is going on at this stage.

Part 4: MCP Knows All

With MCP v3.0, the workload processes are able to be scheduled to run with each getting an "equal" share of the processor. *Note: MCP v3.0 should be able to work with any set of workload programs it reads in.* The workload programs we will provide you will (ideally) give some feedback to you that your MCP v3.0 is working correctly. However, you should 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.

Program Requirements:

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 may include information 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 must output the analyzed process information for every child process each time the scheduler completes a cycle. Of course, the format of the output is important too. One thought is to do something similar to what the Linux top(1) program does (i.e. format the information into a table that constantly updates each cycle).

Project Remarks:

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. 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 the 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.

Part 5: Extra Credit

When the MCP schedules a workload process to run, it assumes that the process will actually execute the entire 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. These time intervals should then be shown in a

new column in your tabular display along with another column indicating the type of work detected.

Note: Since this is for extra credit, we reserve the right to be strict here. The extra points may come in an all-or-none fashion. \Also be careful not to introduce new errors or bugs into your program.

Project Structure Requirements:

For a project to be accepted, the project must contain the following 5 files and meet the following requirements: (The naming conventions listed below **must** be followed. Additionally you must use the C programming language for this assignment. No projects written in another programming language will be accepted.)

part1.c: This is the c-file that contains your MCP v1.0. It must meet the specifications described in Part 1 of the project details.

part2.c: This is the c-file that contains your MCP v2.0. It must meet the specifications described in Part 2 of the project details.

part3.c: This is the c-file that contains your MCP v3.0. It must meet the specifications described in Part 3 of the project details.

part4.c: This is the c-file that contains your MCP v4.0. It must meet the specifications described in Part 4 of the project details.

part5.c (optional): This is the c-file that contains your MCP v5.0. It must meet the specifications described in the Extra Credit section of this project description.

MCP.h (optional): Your header file that contains any function definitions and structs. You can add whatever you wish to this header file or you could forgo this file completely.

Makefile: Your project must include a standard make file. It must produce exe's with the following names: **part1, part2, part3, part4, part5 (optional)**. *Note: do not include the attached sample programs to your makefile. We use different programs with their own names. As such doing so will cause your make file to fail and you will lose easy points.*

Report: Write a report (minimum 1 full page) on your project using the sample report collection format given. Feel free to write as much as you wish, we use the reports to aid in our grading. Report format and content suggestions are given in the report collection template.

Note: Additionally, you are allowed to add any other *.h and *.c files you wish. However, when we run your code we will only be running the part* files. Make sure your code runs in the VM before submission.

Submission Requirements:

Once your project is done, do the following:

1. Open a terminal and navigate to the project folder. Compile your code in the VM with the -g flag.
2. Run your code and take screenshots of the output as necessary (of each part).
3. Create valgrind logs of each respective part:
    a. "**valgrind --leak-check=full --tool=memcheck ./a.out > log*.txt 2>&1** "
4. Create a Tar containing only the files/logs/screenshots and submit it onto Canvas.
5. Submit a .pdf of your project report separately onto canvas. (i.e. you should upload two things: your report and the tar.gz)

Valgrind can help you spot memory leaks in your code. As a general rule any time you allocate memory you must free it. Points will be deducted in both the labs and the project for memory leaks so it is important that you learn how to use and read Valgrind's output. See (https://valgrind.org/) for more details. Finally, please abide by the submission guidelines, we will be grading your project using a standard procedure, failure to do so will result in lost points.

Grading Rubric:

Points	**Description**
15	Completed Part 1 as described above
30	Completed Part 2 as described above
30	Completed Part 3 as described above.
15	Completed Part 4 as described above.
5	No memory leaks when checked with valgrind
5	Report collection in proper format and filled out. (1-2 pages)
10 pts	Completed the extra credit

Late Homework Policy:

- 10% penalty (1 day late)
- 20% penalty (2 days late)
- 30% penalty (3 days late)
- 100% penalty (>3 days late) (i.e. no points will be given to homework received after 3 days)