Project #1 – MCP Ghost in the Shell

CIS 415 - Operating Systems
Spring 2019 - Prof. Malony

Due date: 11:59pm, Friday, May 10

Overview

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 the 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.

In this assignment, you will implement the ***MCP Ghost in the Shell*** (MCP for short) whose job it is to run and schedule a workload of programs on a system. The MCP will read a list of programs (with arguments) to run from a file, start up and run 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, keeping 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, signal handling, and scheduling. Part 5 is for extra credit.

All programming must be done in the C programming language. Your MCP and the programs it runs must be compilable and runnable in the CIS 415 Linux virtual machine (VM) environment. Learning the Linux systems calls is the main purpose of the project. These are found in Section 2 of the Linux manual.

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()`, `execvp()` (or some varient), and other system calls. 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:

```
for i=0, numprograms-1 {
  read program[i] and its args[i] from the file;
  pid[i] = fork();
  if (pid[i] < 0) {
    // handle the error appropriately;
  }
  if (pid[i] == 0) {
    execvp(program[i], arguments[i]);
    // log error. starting program failed.
    // exit so duplicate copy of MCP doesn't run/fork bomb.
    exit(-1);
  }
}
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 of these system calls.**

Also, you will need to figure out how to read the programs and their arguments from the "workload" file and get them in the proper form to call `execvp`. These programs 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 programs will be provided to evaluate your work, but you should also construct your own.

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. Actually, it will take two steps.

Step 1 of Part 2 will implement a way for the MCP to stop all forked (MCC) child processes right before they call `execvp()`. 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 `execvp()` with its associated workload program. The `sigwait()` system call will be useful here. Note, until the forked child process does the `execvp()` call, it is running the MCP progam 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 `execvp()` call.

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.

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 created using `fork()`, the forked MCP child process waits on the `SIGUSR1` signal before calling `execvp()`.
- After all of the MCP child processes have been forked and are now waiting, the MCP parent process sends each child process a `SIGUSR1` signal to wake them up. Each child process will then return from the `sigwait()` and call `execvp()` to run the workload program.
- After all of the workload programs have been launched and are now executing, the MCP sends each workload process a `SIGSTOP` signal to suspend them.
- After all of the workload processes have been suspended, the MCP send each program a `SIGCONT` signal to wake them up.
- Again, once all of the processes are back up and running, for each program, the MCP waits 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. You should test out that things are working properly. One way to do this is to put in MCP output messages to indicate what steps the MCP is presently taking and for which child process. 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. 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 resume 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 and not as accurate as it can be. 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.

Again, please think about how to test that your MCP 3.0 is working and provide some demonstration. Having the MCP produce output messages at scheduling points is one way to do this.

Part 4 – MCP Knows All

With MCP 3.0, the workload processes are able to be scheduled to run with each getting an "equal" share of the processor. Note, MCP 3.0 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 3.0 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.

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(1)` 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 project, you will likely want to learn about these system calls:

- `fork(2)`
- `execvp(2)`
- `wait(2)`
- `sigwait(2)`
- `alarm(2)`
- `signal(2)`
- `kill(2)`
- `exit(2)`
- `read(2)`
- `write(2)`
- `open(2)`
- `close(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 is provided as part of your Linux VM. 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 room 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.

Helping Your Classmates

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.

In your status report on the project, you should provide the names of classmates that you have assisted, with an indication of the type of help you provided. You should also indicate the names of classmates from whom you have received help, and the nature of that assistance.

Note, that this is not a license to collude. We will be checking for collusion, in different ways. It is better to turn in an incomplete solution that is your own than copy of someone elses work without attribution.

Marking Scheme

You must be sure that your code works correctly on the virtual machine under VirtualBox, regardless of which platform you use for development and testing. Leave enough time in your development to fully test on the virtual machine before submission. The marking scheme is as follows:

Points	Description
10	Your report - honestly describe the state of your submission
20	Part 1: Launch Workload. 6 Working Process launcher. 1 Successfully compiles. 1 Compiles with no warnings. 1 Successfully inks. 1 Links with no warnings. 6 Works correctly. 4 Valgrind reports no memory errors.
20	Part 2: Controlled Launch. 6 Working Controlled launch. 1 Successfully compiles. 1 Compiles with no warnings. 1 Successfully inks. 1 Links with no warnings. 6 Works correctly. 4 Valgrind reports no memory errors.
30	Part 3: Time Slice Scheduler . 10 Working Time Slice Scheduler. 1 Successfully compiles. 1 Compiles with no warnings. 1 Successfully inks. 1 Links with no warnings. 10 Works correctly. 6 Valgrind reports no memory errors.
20	Part 4: Gather Process Information. 6 Working Display of Process information. 1 Successfully compiles. 1 Compiles with no warnings. 1 Successfully inks. 1 Links with no warnings. 6 Works correctly. 4 Valgrind reports no memory errors.
25	Part 5: Controlled Launch. 8 Working Dynamic Scheduler. 1 Successfully compiles. 1 Compiles with no warnings. 1 Successfully inks. 1 Links with no warnings. 8 Works correctly. 5 Valgrind reports no memory errors.

Several things should be noted about the marking schemes:

- Your report needs to be honest: Stating that everything compiles and it doesn't on Arch means you didn't compile it. This is the easiest 10 points you will earn if your honest.
- Fork bomb or segfault on good input will lose 100% of working points.
- Fork bomb or segfault on bad input will lose 50% of working points.

Turning in Your Code

There are at least 7 files to turn in as a TGZ archive:

- part1.c
- part2.c
- part3.c
- part4.c
- part5.c (optional)
- header.h (your header file with structs)
- Makefile

You might have other .c/.h helper files, as needed by your project. It might be helpful to have a README file, but it is not necessary.

Please make a TGZ archive the same way you did in Project 0. You will turn it in on canvas.