Project 2

(MCP: Ghost-in-the-shell)

CIS 415 - Operating systems

2021 Spring- Prof. Allen Malony

Due date: **11:59 pm, Sunday, May 16th, 2021**

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 inhabitin ... , Arthur Koestler furthers the position t ... experience of mind-body duality is ... e is both a whole and a part. The manga ser ... se title is an homage to Koestler, explores ... ality *IS* the "ghost" and whether it can ex ... ies. When I was going to UCLA for m ... as an intern at Burroughs Corporatio ... . Sadly, Burroughs no longer exists, havi ... t they were a major player in computing s ... hip at Burroughs, my job was to write m ... y known as the ***MCP***:

htt ... MCP

The MCP mig ... ting systems, but it was innovative in ma ... nage multiple processors, the first c ... 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 **<u>must</u>** be saved in a separate *.c file in different directories with their own make file.

Note: For part 1, Do not test to spawn more than 7 processes.

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 a

Program Requireme

Your MCP v1.0 must

1. Read the prog … d work just like the file mode fror … d and its arguments.
2. For each com … run the command using some va
   a. **fork(2**
   b. **One o**
      i.
3. Once all of th … process to terminate using one of t
   a. **wait(2** … l
4. After all proc … exit() system call.
   a. **exit()**:

Fig. 1 presents the pseudocode for launching processes:

```
for (int i = 0; i < line_number; i++)
{
        pid_array[i] = fork();
        if (pid_array[i] < 0)
        {
                //error handling
        }

        if (pid_array[i] == 0)
        {
                if (execvp (path, arg) == -1)
                {
                        error handling
                }
```

[illegible]

Remarks:

To make things simpl [illegible] nt used to execute the MCP (i.e. the VM [illegible] many things that can go wrong. You shoul [illegible] f these system calls. Also, you will need to [illegible] ents from the "workload" file and g [illegible] ds can be anything that will run in the sa [illegible] in which the MCP is launched. A set of co [illegible] t you should also construct your own. I [illegible] 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 a [illegible] ess wait for a `SIGUSR1` signal bef [illegible] . The sigwait() system call will be us [illegible] *s the exec() call, it is running the same cod* [illegible] ing, the MCP is in a state where each chil [illegible] ne a workload process is selected to [illegible] P sending the `SIGUSR1` signal to it [illegible] t launches the associated workload [illegible]

Step 2:

Step 2 will im [illegible] a running process to stop (using the `SIGS` [illegible] `GCONT` signal). This is the mechanism that [illegible] the first time. Sending a `SIGSTOP` [illegible] t the command line and typing Ctrl-Z to s [illegible] `ONT` signal is like bringing a suspended [illegible]

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 progr [illegible]

Again, a set o [illegible] t you should also construct your own. H [illegible] han described here. We recommend that y [illegible] nderstanding of the system calls used in t [illegible]

Part 3: MCP Sch [illegible]

Now that the [illegible] ant to implement a MCP scheduler that w [illegible] licy. The simplest policy is to equally sh [illegible] ount of time to run (e.g., 1 second). The [illegible] cuting. After its time slice has completed, [illegible] process. The MCP decides which is the [illegible] nues that process.

Program Requireme [illegible]

MCP 2.0 know [illegible] e 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 … g and provide some demonstration. Havin … nts is one way to do this. The console outp … l grade so take some time to think about ho … ge.

Part 4: MCP Kno…

With MCP v3.0, the w … each getting an "equal" share of the p … *any set of workload programs it reads in.* … lly) give some feedback to you that y … uld also write your own simple test progr … tion is proceeding by looking in the /proc d…

Program Requireme…

In Part 4, you … ant data from */proc* that conveys some inf … process is consuming. You will pick out a minimum of 5 properties for each process and display them on your terminal. You will also update these informations according to the time slice you assigned from part 3. 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 ch [illegible] task, but a very important one. Code t [illegible] l have marks deducted. Fortunately [illegible] Remember that valgrind, while quite [illegible] find and fix any bugs that are located by va [illegible] y memory error in your code: especially [illegible]

Developing Your Co [illegible]

The best way to deve [illegible] chine image provided to you. This [illegible] start. This also gives you the benefit of tak [illegible] omething potentially risky or hazardous, so [illegible] ll 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- ... cifications described in Part 3 of the projec...

part4.c: This is the c- ... cifications described in Part 4 of the projec...

part5.c (optional): T... eet the specifications describ... n.

MCP.h (optional): Y... d structs. You can add whatever you wis... etely.

Makefile: Your proje... xe's with the following names: **par**... *t include the attached sample prog... their own names. As such doing so will ca...*

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 p result in lost points.

Grading Rubric:

Part	Points	Description
Part 1	15	5 error handling 5 wait for processes to finish 10 correct processes spawned
Part 2	30	10 correct implementation of sigwait() 10 correct sequence of signal sent 5 sleep() called in between signals for understandable format 5
Part 3	30	1(1(2(1(2
Part 4	15	4 pr 1 -5
Valgrind	5	1
Report	5	R ges)
Part 5	10	C

Note: some sections may have more detail points than the total points, meaning there are more than 1 way you can get a 0 in that section.

1. 0 if your program does not compile.
2. 0 if your program does not work in the required environment.
3. 5 points deduction if your makefile does not work.

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)