Assignment #2: The Kernel and Process Execution

Due: February 12, 2021 at 23:55 on myCourses

1. Preliminary Notes

- The question is presented from a Linux point of view using the computer science server `mimi.cs.mcgill.ca`, which you can reach remotely using `ssh` or `putty` from your laptop (see lab 1). We suggest you do this assignment directly from `mimi.cs.mcgill.ca` since these are the computers the TA will be using.
- Your solution must be built using the modular programming techniques we discussed in the in-class exercise sessions and the C labs.
- **You must write this assignment in the C Programming language.**

2. Assignment Question: Building a Kernel

2.1 Overview:

For this assignment, you will need the **fully working OS Shell from Assignment 1**. You can either build on top of your past submission, or use the official solution provided on myCourses. The goal of this assignment is to add one new command to your shell (detailed command description in Section 2.5):

COMMAND	**DESCRIPTION**
`exec prog1 prog2 prog3`	*Executes up to 3 concurrent programs provided as arguments*

High-level notes on `exec` command behavior:

- `exec` takes **one to three arguments**. Each argument is the name of a **different** `mysh` script filename. For this assignment, `exec` does not permit us to launch multiple scripts with the same filename. If you try to do that your shell displays the error, *"Error: Script <name> already loaded"* and all program script execution terminates.
- If there is a load error, then no programs run. The user will have to input the `exec` command again.
- To simplify the assignment, we will assume that "compiled" programs are the same as `mysh` scripts. We will reuse the shell's interpreter to both run shell scripts and execute the kernel programs. A more detailed description follows in Section 2.5.

Important: For simplicity, you are simulating a **single core CPU, that does NOT support threading**. We will also not be testing recursive `exec` calls.

Even though you only need to add one command to the shell, this is a much more complex task than the previous commands in Assignment 1 (so start early, use your time wisely, and ask the TAs for guidance if you get stuck). To support this new command, you need to simulate the kernel run-time environment, which includes the **PCB**, the **ready queue**, the **CPU scheduler**, and **a temporary simple memory** (which we will upgrade in assignment 3).

This rest of this document provides a description of the file structure in Section 2.2, compilation notes in Section 2.3, a description of the data structures and their behavior in Section 2.4, a description of the `exec` command and its behavior in Section 2.5, and a description of the scheduling algorithms in Section 2.6.

2.2 Your source files:

Your entire application must contain the following source files:

- **shell.c**, **interpreter.c**, and **shellmemory.c,** together with their respective .h files from Assignment 1.
- **kernel.c**: This is the true home of the `main()` function in the OS, so you will move the `main()` function from shell.c to kernel.c. The kernel.c `main()` function calls the shell.c `int shellUI()` function (which is your previous `main()` function from Assignment 1), to display the command-line to the user. The kernel `main()` will also instantiate all the kernel data structures.
- **cpu.c:** Contains data-structures and algorithms to simulate task switching.
- **pcb.c:** Contains data-structures and helper functions for the process control blocks.
- **ram.c**: Contains data-structures and helper functions for RAM simulation.

Add .h files as you see fit.

All source files must be built using modular programming techniques (see lab 2).

2.3 Compiling and running your kernel:

Compile your application using `gcc` with the name **`mykernel`**. To do modular programming, you must compile your application in the following manner:

```
gcc -c shell.c interpreter.c shellmemory.c kernel.c cpu.c pcb.c ram.c
gcc -o mykernel shell.o interpreter.o shellmemory.o kernel.o cpu.o pcb.o
ram.o
```

To run your kernel, from the command line prompt type: `./mykernel`

Similar to the last assignment, `mykernel` will display:

```
Kernel 1.0 loaded!
Welcome to the <your name goes here> shell!
Shell version 2.0 Updated February 2021
$
```

As before, the dollar sign is the prompt of your fully functional shell from Assignment 1. From this prompt the user will type in their command and the shell will display the result from that command on the screen (or an error message), and then the dollar sign is displayed again prompting the next command. The user stays in this interface until they ask to quit.

2.4 Kernel data structures:

You will need to implement the following data structures: RAM, Ready Queue, PCB, and the CPU scheduler.

2.4.1. RAM

Each program executed by the shell's `exec` command is loaded into the RAM. **This is not true for the mysh scripts, they are loaded and executed as in Assignment 1**. The `exec` command runs programs concurrently, therefore the programs need to be in the RAM data structure at the same time.

Implementation. RAM is implemented as array of `char*` pointers:

```
char *ram[1000];
```

- The **entire source file** of each program is loaded into the `ram[]` array.
- Each line from the source file is loaded into its own cell in the `ram[]` array.
- This RAM has space for 1000 lines of code, from at most 3 programs at the same time (i.e., about 330 lines of code, on average, per program).
- A `NULL` pointer indicates that there is no code at that cell location in `ram[]`.
- A program is said to be in memory when all its lines of code are copied into the cells of the `ram[]` array. If there is not enough space to store all the lines in the script then the load terminates with an error message: *"ERROR: Not enough RAM to add program."*

Steps for loading a program *P* into RAM:

1. `fopen` the program *P*.
2. Find the next available cell in `ram[]`.
3. Copy all the lines of code into `ram[]`. *Hint: To read a line from the program do*
   `fgets(file,buffer,limit);`
   `ram[k] = strdup(buffer);` *// k is the current line the program occupies in ram[].*
4. Remember the start cell number and the ending cell number of *P*'s location in `ram[]`.
   Hint: it helps to keep track of the next free cell in ram[] with "private" variables in ram.c.

Steps for unloading a program *P* from RAM, when the program completed its execution:

1. For all the lines of code in `ram[]` occupied by *P*, do `ram[k]=NULL;`

2.4.2. Ready Queue

Implementation. The ready queue is where the PCBs of processes that are ready to run are enqueued. This queue is used by the scheduler. In this assignment, the Ready Queue is FIFO and Round Robin (RR). Ready Queue is implemented as a **linked list** of PCBs, with head and tail pointers. Pointer "head" points to the first PCB in the list and pointer "tail" points to the last PCB in the list.

- The first PCB is the one that gets the CPU.
- New PCBs are appended at the tail.
- Make sure to update "tail" to point to the new PCB after appending it to the list.

2.4.3. PCB

Implementation. For this assignment our PCB will be very simple. It will contain only three variables: **a program counter (PC),** the **program's start address,** and the **program's ending address.** PCB is implemented as a struct with three fields:

```
struct PCB {int PC; int start; int end;};
```

- **PC** will be an integer number that refers to the cell number of `ram[]` containing the instruction to execute.
  Note 1: *the PCB's PC field is not the CPU instruction pointer, therefore it is updated only after a task switch. The PCB's PC is updated after the quanta is finished (see Section 2.4.4. below).*
  Note 2*: when a program is launched its PCB is created and the PC field points to the first line of the program.*
- **start** contains the cell number of `ram[]` of the first instruction of the program.
- **end** variable contains the cell number of `ram[]` of the last instruction of the program.

2.4.4. CPU

Implementation. Our CPU is simulated by a struct having an **instruction pointer (IP)**, **instruction register (IR)**, and a **quanta** field.

```
struct CPU {int IP; char IR[1000]; int quanta=2;}
```

- **IP** is similar to the PCB PC field. IP points to the next instruction to execute from `ram[]`.
- **IR.** The currently executing instruction is stored in the IR. IR stores the instruction that will be sent to the interpreter for execution. This simulates how the assembler instruction is loaded from RAM into the CPU's Instruction Register. The `interpreter()` function simulates the CPU's sequencer, which executes the instruction.
- **quanta** represents the amount of time that each process executes before being switched out by the scheduler. For this assignment, we assume the quanta is equal to 2 lines of code for each program.

Since we are not implementing threads, all concurrent programs access the same shell memory space. We are using the **shell memory of Assignment 1** as one global memory space where all the scripts will read and write, even the command line scripts. We will upgrade the memory in Assignment 3.

2.5 Detailed behavior and operation of the `exec` command:

The user can input any of the following at the command line:

- `exec p1.txt` **Behavior**: one program is launched in the kernel.
- `exec p1.txt p2.txt` **Behavior**: two programs are launched.
- `exec p1.txt p2.txt p3.txt` **Behavior**: all three programs are launched.

Launching a program *P* entails the following:

1. *P*'s source file is `fopen`-ed and the source code is loaded completely into `ram[]`.
2. A PCB is created (`malloc`) for *P* and the PCB's fields are initialized.
3. The PCB is added to the Ready Queue.

Important: Steps 1-3 are done for all the programs in the `exec` arguments list.

Completing the execution for a program *P* entails the following cleanup operations:

1. Assign `ram[]` cells to `NULL` for all of *P*'s instructions.
2. Remove *P*'s PCB from the Ready Queue.
3. Call `free(PCB)` to release C language memory.

Since we are not implementing threads the shell prompt only displays after the `exec` command has completed executing all the programs.

2.6 Kernel algorithms and program execution:

You will implement the following algorithms: **scheduler**, **task switch**, and **basic memory management**. The program execution flow and algorithm details are presented below.

Starting the execution. The execution starts in the **`exec()`command,** an interpreter function in interpreter.c. **`exec()`** does the following:

- It handles the filename argument verification error.
- It opens each program file.
- It calls the **`myinit()` function** in kernel.c (see below) to load each program into the simulation. For example, if there are three programs then `myinit`() is called three times.
- **Only if `myinit()` is successful,** it starts the execution of all the loaded programs by calling the **`scheduler()` function** in kernel.c.
- The `exec()` function does not terminate until all the programs it loaded have **all** completed execution.

Loading the programs. The programs are loaded through the **`myinit()` function** lives in kernel.c. Its role is to load data in `ram[]`, create PCBs, and append them to the Ready Queue, as described in Sections 2.4.1, 2.4.2., and 2.4.3. It is defined as:

```
int myinit(char *filename)
```

`myinit()` calls the following functions:

- `void addToRAM(FILE *p, int *start, int *end)` from ram.c to add the source code of the program to the next available cells in `ram[]`.
- `PCB* makePCB(int start, int end)` from pcb.c to create a new PCB *using malloc.*
- `void addToReady(PCB *)` from kernel.c to add the PCB to the tail of the Ready Queue. Each PCB in the Ready Queue is sorted First-in First-out (FIFO) and Round Robin (RR), in the same order as they appeared in the `exec` command.

The programs can start execution only after these three steps are successful.

Running the programs. Kernel.c has the function **`scheduler()`** which is called after all the programs have been loaded into the simulator. The `scheduler()` function is our main execution loop. It is defined as:

```
int scheduler()
```

`scheduler()` does the following:

- Check if CPU is available. This means that the quanta is finished or nothing is currently assigned to the CPU.
- Copy the PC from the PCB into the IP of the CPU.
- Call `int run(int quanta)` in cpu.c to run the program, by copying quanta lines of code from ram[] using IP into the IR, which then calls `interpreter(IR)`. This executes quanta instructions from the program or until the script file is at end.
- If the program is not at the end, the PCB PC pointer is updated with the IP value and the PCB is placed at the tail of the ready queue.
- If the program is at the end, then it terminates (as described in Section 2.5 above).

Ending the execution. When the Ready Queue is empty, all the programs have terminated. At this point the `exec()` function ends, and the user sees the shell command line prompt.

2.7. Testing your kernel:

The TAs will use and modify the provided text file to test your kernel. This text file will contain the same tests from Assignment 1 with additional `exec` command calls. Since we are not implementing threads, **we will not be testing recursive `exec` calls**. You can also use this file to test your kernel or you can test it the old fashion way by typing input from the keyboard. To use the provided text file, you will need to run your program from the OS command line as follows:

```
$ ./mykernel < testfile.txt
```

Each line from testfile.txt will be used as input for each prompt your program displays to the user. Instead of typing the input from the keyboard the program will take the next line from the file testfile.txt as the keyboard input.

Important: When testfile.txt is exhausted of input the shell command line prompt is displayed to the user (unless testfile.txt had a quit command, in that case `mykernel` terminates).

Make sure your program works in the above way.

2.8. Comparing scheduling policies:

In this last exercise, we will compare scheduling policies for the kernel. You were provided with 3 programs: sched1.txt, sched2.txt, and sched3.txt. Using the same technique from Assignment 1 Section 2.7, time the total execution of the command:

```
exec sched1.txt sched2.txt sched3.txt
```

Answer the following questions in a file called **SCHEDULING.TXT:**

3.1. Report the output and average number of cycles that the `exec` command above took to execute, over 5 measurements.

3.2. Now **change the quanta to 5** and do the same as 3.1 (*Hint: don't forget to recompile*).
Do the same with a **quanta of 20**.

3.3. Comment on the results you noticed in 3.1. and 3.2:

   a. What differences (if any) do you notice in output? Why?

b. What differences (if any) do you notice in the execution time? Why?
c. How would the results change if quanta was an amount of time that the processes are allowed to run, instead of a number of code lines? You do not need to implement a different algorithm. Just say what you think would happen in this alternative scenario and explain why.

4. WHAT TO HAND IN

Your assignment has a **due date on February 12, 2021 at 23:55** plus two late days. If you choose to submit your assignment during the late days, then your grade will be reduced by -5% per day. Submit your assignment to the Assignment #2 submission box in myCourses. You need to submit the following:

- A README.TXT file stating any special instructions you think the TA needs to know to run your program.
- Your version of TESTFILE.TXT. This will be you telling the TA that you know for sure that your program can at least do the following. The TA will run your program with this file and they will also run it with their own version of the file.
- Submit all the .c file described in the assignment (you may want to create .h files, if so, please hand those in as well).
- **Submit a bash file to compile your source files into an executable**.
- Your SCHEDULING.TXT file with the answers to Question 2.8.

Note: You must submit your own work. You can speak to each other for help but copied code will be handled as to McGill regulations. Submissions are automatically checked via plagiarism detection tools.

5. HOW IT WILL BE GRADED

Your assignment is graded out of **20 points** and it will follow this rubric:

- The student is responsible to provide a working solution for every requirement.
- The TA grades each requirement proportionally. This means, if the requirement is only 40% correct (for instance), then the student receives only 40% of the points assigned to that requirement.
- **Your program must run to be graded**. If it does not run, then the student receives **zero** for the entire assignment. If your program only runs partially or sometimes, you should still hand it in. You will receive partial points.
- The TA looks at your source code only if the program runs (correctly or not). The TA looks at your code to verify that you (A) implemented the requirement as requested, and (B) to check if the submission was copied.

Mark breakdown:
- **1 point** - Source file names. The TA must verify that the student created the specified source files as the only source files in the application. In addition, the source files must be populated with at least what was specified in the assignment description for each file. The student is permitted to supply additional helper functions as they see fit, if it does not hinder the assignment requirements.

- **2 points** - Modular programming. If the student wrote their application using modular programming techniques as described in lab 2 (or seen from another course) then they receive these points.
- **1 point** - Executable named `mykernel`. The students compiled program must be named `mykernel`.
- **1 point** - A fully working Assignment 1 is contained within `mykernel`.
- **1 point** - Ability to use the shell to input the `exec` command (and see error message).
- **2 points** - The `myinit()` function.
- **2 points** - The scheduler() function.
- **1 point** - The Ready Queue implementation.
- **3 points** - The `run()` function.
- **2 points** - Terminating a program.
- **1 point** - The `ram[]` data structure.
- **1 point** - The CPU data structure.
- **1 point** - Program can be tested with the TESTFILE.TXT file.
- **1 point** - The answers to Question 2.8 in the SCHEDULING.TXT file (0.5 points for reporting the results, and 0.5 points for the interpretation of the results).

Good luck!