Assignment #2

The Kernel and Process Execution

Due: February 24, 2020 at 23:55 on myCourses

Labs 3 and 4 will provide some background for this assignment.

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). If you do this assignment from an MS Windows machine, then make sure to provide the DLL libraries your program uses (if any) so that the TA can run it from their MS Windows machine. It is not the TA's responsibility to make your program run. The TAs will not debug your program.

You must write this assignment in the C Programming language.

Assignment Question: Building a Kernel

This assignment builds from assignment 1. You will add a new command line command:

```
exec prog1 prog2 prog3
```

This new command will simulate the kernel run-time environment, which includes the PCB, the ready queue, the CPU, and a temporary simple memory (which we will upgrade in assignment 3). To make this easier we will repurpose code from assignment 1.

In assignment 2 you will maintain a fully working assignment 1 and add to it a kernel run time environment with the additional command line command: `exec`.

You will need to implement the following data structures: CPU, PCB, ready queue, and RAM. You will need to implement the following algorithms: schedular, task switch, and basic memory management.

You are NOT implementing threading. You are simulating a single core CPU.

This assignment first provides a description of the files and data structures and how they work. Then a description of the new shell command and how it works. Lastly a description of the new algorithms.

Your source files:

> Your entire application must contain the source files you used from assignment 1: shell.c, interpreter.c, and shellmemory.c. You must add the following assignment 2 files: kernel.c, cpu.c, pcb.c, and ram.c. Add .h files as you see fit. These source files must be built using **modular programming techniques** (see lab 2).
>
> Move the main() function from shell.c to kernel.c. This is the true home of the main() function. 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.

Compiling assignment 2:

> 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
```

Running assignment 2:

From the command line prompt type: `./mykernel`

What mykernel displays to the user:

```
Kernel 1.0 loaded!
Welcome to the <your name goes here> shell!
Shell version 2.0 Updated February 2020
$
```

The above dollar sign is the prompt. The cursor waits beside the prompt waiting for the user's input. 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.

Your shell from assignment #1 is fully functional in this assignment. A single new command is added to the shell, called exec.

New command supported by your shell:

COMMAND	DESCRIPTION
exec p1 p2 p3	Executes concurrent programs $ exec prog.txt prog2.txt

The exec command 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". 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.

Assignment 2 data structures description:

- The RAM
  - Implemented as an array of `char *` pointers.
    - `char *ram[1000];`
  - Each program executed by the shell's exec command is loaded into 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 ram[] at the same time. To simplify this, RAM is an array of char * pointers and the entire source file is loaded into this array. Each line from the source file is loaded into its own cell in the array. This RAM has space for 1000 lines of code, from at most 3 programs at the

same time. This means about 33 lines of code, on average, per program. A program is said to be in memory when all of its lines of code are copied into cells of the array. If there is not enough space to store all the lines in the script then the load terminates with an error. A NULL pointer indicates that there is no code at that cell location in ram[].

- When a program is loaded to ram[] the following operations happen:
  - Step 1: fopen the script
  - Step 2: find the next available cell in ram[]
  - Step 3: copy all the lines of code into ram[]
  - Step 4: remember the start cell number and the ending cell number of that script in ram[]
  - Notes:
    - To read a line from the program do:
      ```
      fgets(file,buffer,limit);
      ram[k] = strdup(buffer);
      ```
      Where k is the current line of the program.
    - When the program has finished executing do:
      `ram[k]=NULL;` For all the lines of code in ram[]

- The Ready Queue
  - Implemented as a simple linked list with head and tail pointers.
    - PCB *head, *tail;
  - The ready queue is FIFO and RR. Pointer "head" points to the first PCB in the list. The first PCB is the one that gets the CPU. Pointer "tail" points to the last PCB in the list. New PCBs are appended at the tail. Make sure to update "tail" to point to the new PCB after appending it to the list.
- The PCB
  - For this assignment our PCB will be very simple. It will contain only three variables: a program counter, the program's start address, and the programs ending address. The program counter will be an integer number that refers to the cell number of ram[] containing the instruction to execute. The start variable contains the cell number of ram[] of the first instruction of the program. The end variable contains the cell number of ram[] of the last instruction of the program.
    - struct PCB { int PC; int start; int end; };
  - Note: the PCB's PC field is not the CPU instruction pointer, therefore it is updated only after a task switch.
  - Note: when a program is launched, a PCB is created, and the PCB's PC field points to the first line of the program. The PCB's PC is updated after the quanta is finished.
- The CPU
  - The CPU is simulated by a struct having an instruction pointer (IP), instruction register (IR), and a quanta field. The IP is like the PCB PC field, it points to the next instruction to execute from ram[]. The currently executing instruction is stored in the IR.
    - struct CPU { int IP; char IR[1000]; int quanta=2; }
  - Note: the 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.

- Note: since we are not implementing threads, all concurrent programs access the same shell memory space. Yes, 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 this is assignment 3.
- Note: for this assignment we assume the quanta is equal to 2 lines of code for each program.

Assignment 2 operation of the exec command:

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

```
exec prog.txt
exec prog.txt prog2.txt
exec prog.txt prog2.txt prog3.txt
```

In the first case only one program is launched in the kernel. In the second case two programs are launched. In the third case three programs are launched. Launched means the following:

(1) each file is fopened and the source code is loaded completely into ram[].
(2) a PCB is created (malloc) for that program and the PCB's fields are initialized, assuming the load was successful.
(3) the PCB is added to the Ready Queue.

This is done for all the programs in the list.

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

This is how a program completes execution:

(a) the ram[] cells are assigned NULL for all instructions,
(b) the PCB is removed from the Ready Queue, and
(c) free(PCB) is called to release C language memory.

Assignment 2 program execution:

Only after the "exec" command loads all the programs into ram[] and appends the PCBs to the Ready Queue, do the programs start to run. 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 exec() function is an interpreter function stored in interpreter.c. It handles the filename argument verification error. It opens each program file. It calls the kernel function to load each program into the simulation. The final thing it does, it starts the execution of all the loaded programs. The exec() function does not terminate until all the programs it loaded have **all** completed execution.

Specifically exec() first calls myinit() for each program to add each program into the simulation. For example, if there are three programs then myinit() is called three times. It then calls scheduler() to run the loaded programs.

Kernel.c has the function myinit(char *filename) which does the following:

1. It calls void addToRAM(FILE *p, int *start, int *end) from ram.c to add the source code to the next available cells in ram[].
2. It calls PCB* makePCB(int start, int end) from pcb.c to create a PCB instance using malloc.
3. It calls void addToReady(PCB *) from kernel.c to add the PCB to the tail of the Ready Queue

Kernel.c has the function scheduler() which is called after all the programs have been loaded into the simulator. The scheduler() function will be our main simulation execution loop. It will do the following:

a. It checks to see if the CPU is available. This means that the quanta is finished or nothing is currently assigned to the CPU
b. It copies the PC from the PCB into the IP of the CPU
c. It calls the run(quanta) function within cpu.c to run the script by copying quanta lines of code from ram[] using IP into the IR, which then calls: interpreter(IR)
d. This executes quanta instructions from the script or until the script file is at end.
e. If the program is not at the end, then the PCB PC pointer is updated with the IP value and the PCB is placed at the tail of the ready queue.
f. If the program is at the end, then the PCB terminates (as described previously / above)

When the Ready queue is empty, this means that all the programs have terminated. At this point the exec() function ends, and the user sees the shell command line prompt.

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.

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.

WHAT TO HAND IN

Your assignment has a due date 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 what OS you used: mimi.cs.mcgill.ca or MS Windows and any other 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**.
- If you used MS Windows and you used a DLL then upload those as well.

You must submit your own work. You can speak to each other for help but copied code will be handled as to McGill regulations.

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
- 2 points - Program can be tested with the TESTFILE.TXT file

You have three weeks for this assignment. Please use that time to your advantage.