McGill University

COMP-310 – Operating Systems – ECSE-427

Assignment #2

Due: October 20, 2008 at 23:55 on Web CT

About Processes

QUESTION 1: Programming Question

The next step in creating your personal operating system is to build the kernel and connect that with the shell you built in the first assignment.

The Kernel is the base portion of the OS. It is the first thing loaded into the computer. It defines, or formats, RAM in the likeness it wants. Kernels are normally constructed as a C main program with a library of important OS service functions. Functions not critical to the operation of the computer are stored as application programs or DLLs on the hard disk and loaded when needed. The shell, from assignment 1, is an example of something not important enough to be kept in the Kernel.

We will build the functionality of the Kernel and shell gradually through these assignments.

The Kernel can be launched from the command-line prompt as follows:

$ mykernel -h

Your Kernel can be invoked with or without the -h. At present, the -h will simply display the correct invocation syntax and then terminate. Without the -h your operating system will load. We will add other switches later.

Your Kernel's main program will look something like this:

```
int main (int argc, char *argv[])
{
        // Step 1: check for switches

        // Step 2: initialize RAM

        // Step 3: initialize run-time environment

        // Step 4: start run-time environment with login screen and shell
}
```

The compromise we will make in this assignment is that our operating system will not run programs but will multi-process scripts. I know, it is a cop-out but we only have four assignments.

Step #1: I will not comment on. You know how to do this.

Step #2: Initialize RAM

Our RAM will be similar to real RAM. It will be a one dimensional array having 100 cells. To make our life simpler. Instead of each cell representing a byte, each cell will represent a string. In assembler one instruction fits within a single cell or two of RAM. In our case, one script sentence will fit into one of our RAM cells.

Create a 1D global array called : char *FREERAM[100];

Make sure to initialize this array to NULL in every cell. This will represent free space in RAM where your user programs can run. A cell marked as NULL is not occupied by an instruction.

Step #3: Initialize Run-Time Environment

At this time your operating system will simulate one PCB Ready queue, on PCB Terminated queue and one PCB Run state. You will initialize all these data structures to NULL and zero at this time to indicate that nothing is running. Your PCB data structure will look like this:

```
struct PCB
{
        char filename[50];     // stores the name of the script loaded into memory
        int start;             // start address loaded into RAM
        int length;            // the number of instructions
        int current;           // current instruction being executed
        char *memory[20];      // list of Shell Memory variables and values
        struct PCB *next;      // next node in the linked-list
} ;
```

The queues and run-time environment will be implemented with linked lists. Therefore there will be three head pointers: struct PCB *Ready, struct PCB *Terminate and struct PCB *Running. The Ready pointer is the head pointer to the queue of waiting scripts. The Terminate pointer is the head pointer to the queue of scripts that have terminated. The pointer Running is not a linked-list. It is a pointer that simply points to the currently executing script. Even though we can execute everything from the Running pointer we will simulate a task-switch and CPU with the following data structure:

```
struct CPU
{
        int quanta; int returnCode;
        int pc;
        char *memory[];
};
```

Step #4: Start Run-Time Environment with Login Screen and Shell

Every kernel has a simple interface into itself for the user. This is normally implemented as a login procedure. Upon successful login the default shell is launched. To make things a little easier for us we will include the login procedure and the shell as part of the Kernel.

Create a function called: int login(void). This function displays a simple user name and password prompt. It will open an unencrypted text file called pass.txt that stores the valid user names and passwords (you can create this file with a text editor). The function returns 1 if the user input a valid user name and password, otherwise it returns 0. Your Kernel repeatedly shows the login prompt until a valid login is made.

#include your shell from assignment #1 into this new assignment. Your shell is invoked by the Kernel after the user properly logs in. Modify your shell so that it operates like a function and not a program. In other words, the Kernel will call your shell. Your shell will display the prompt and wait for the user's input. It will then execute the user's command (as in assignment #1) but once that is done, instead of looping it will return back to the Kernel. The Kernel will be the main loop for the entire system. Keep the argc and argv feature from the shell's memory but treat it as simple parameters in this case.

The code in Step #4 will look something like this:

```
while (!login());

while (returnCode != -1)
{
        returnCode = shell(); // returns -1 when user inputs EXIT or LOGOUT

        // OS Overhead – Process execution

        Running = PopReadyQueue();
        TaskSwitchIn(Running);
        SetQuanta(int);
        RunForQuanta();
        TaskSwitchOut(Running); // task switch back
        PushReadQueue(Running);

        // OS Overhead – Services

        TerminateProcesses();
}
```

As the code above implies, you will need to include additional commands to your shell. Here is what you need to add:

- The commands EXIT and LOGOUT that will simply return a -1 to the calling program. All other shell commands return a zero.
- RUN FILENAME1 FILENAME2 ... will be the command to initiate script execution in our multi-processing environment. In some ways it looks like SCRIPT FILENAME but SCRIPT executes a script within the shell and not part of the multi-processing environment. RUN will do the following things:
  - Find space in FREERAM and load the entire script file into FREERAM. Use a simple first-come first-serve strategy for finding space in FREERAM.
  - Create a PCB and insert that into the Ready list. You will have to malloc the struct.
  - The program is not executed at this moment. It simply resides in the Ready list.
  - An important note, unlike our shell's memory, this script's memory will be stored within the space provided by the PCB.
- An additional command for the shell will be ECHO SENTENCE. ECHO will print on the screen the words in SENTENCE as is. This does not implement variables.

The code above also shows how you are to implement the multi-processing environment. You will need to implement a queue, a task switcher and a quanta. This I think is all obvious from the text already but to make sure:

- The Ready list will have two functions, PopReadyQueue and PushReadyQueue. Pop will remove the PCB at the head of the queue and assign it to the pointer Running. Push will take the PCB pointed to by Running and insert it back into the Ready queue. The queue implements a simple round-robin ordering. PushReadyQueue also looks at the CPU returnCode to push the Running process to the Terminated queue instead of the Ready queue (this will be described more below) when a script has asked to terminate.
- The function TaskSwitchIn will copy the important values from the PCB pointed to by Running into the CPU data structure. The function TaskSwitchOut does the opposite, it copies the values back out to Running.
- SetQuanta and RunForQuanta implement the multi-processing run-time environment. SetQuanta initializes the CPU quanta variable to the value permitted for this process. For now, simply set this always to the number 3. RunForQuanta will use the quanta variable in the CPU to execute those many instructions from the script loaded into RAM. It will execute this starting at the cell pointed to by PC. PC is incremented after each instruction is executed.
- The function TerminatedProcess uses the Terminate queue. This queue stores all the PCBs that have issued an EXIT or LOGOUT command. These scripts have terminated and need to be removed from memory. Your Kernel will process all these PCBs on this queue in a first-come first-serve basis. Each PCB's structure will be freed and the cells in FREERAM that contain its instructions will also be freed and assigned to NULL. You need to free first and then assign to NULL, unlike Java.

The simulation of the CPU is carried out by the CPU data structure and the script interpreter. You must view the CPU struct as the registers of the CPU. The script interpreter must be converted into a line-interpreter and viewed as the decoder and circuitry of the CPU. Converting your assignment #1 script interpreter into a line-interpreter is easy (you may have already done it). Your line-interpreter will have the following syntax: int interpreter(char *command, char *memory[]). The function's return value

will always be zero unless the script command was EXIT or LOGOUT. Then it will return -1. All line-interpreter return codes are stored in the CPU struct's returnCode variable. All your scripts will need to terminate with an EXIT or LOGOUT as their last statement. To do this well the line-interpreter should be able to handle both shell scripts, as from assignment #1, and these multi-processing scripts. This is easy to do. Char *command is the command to interpret. It comes either from the shell SCRIPT command or the multi-processed script in FREERAM. Char *memory[] is set to NULL when using the shell's memory from assignment #1, or it points to memory from the PCB record.

To test your Kernel you should be able to load 3 scripts into this run-time environment. You should implement SET and GET commands for the scripts memory and ECHO statements to see the interleaving of statements as these multiple processes execute at the same time. The GET command also outputs to the screen the contents of the variable that was SET.

Have fun with this one.

QUESTION 2: An optimal execution environment is defined to be the proper balance between the use of PCB's, LWP's and the execution states (ready, running, blocked, terminated and loaded), the task switch (etc) and the ration of time the CPU spends executing OS tasks versus user tasks. Properly balanced means that OS task CPU usage has been minimized. This then allows maximized CPU usage for user tasks. Using only a couple of sentences, outline the optimal execution environment for the following situations (justify your answer):

a) A single user and single process computer.
b) A single user but multi-process computer.
c) A multi-user and multi-process computer.
d) A multi-user, multi-process and multi-processor computer.

Provide an argument to justify your optimality statement.

QUESTION 3: In a block/wakeup mechanism, a process blocks itself to wait for an event to occur. Another process must detect that the event has occurred, and wakeup the blocked process.

A) Is it possible for a process to block itself to wait for an event that will never occur? How?
B) Can the operating system detect that a blocked process is waiting for an event that will never occur? How?
C) What reasonable safeguard might be built into an operating system to prevent processes from waiting indefinitely for an event?

QUESTION 4: The ability of one process to spawn a new process is an important capability, but it is not without its dangers. Consider the consequences of allowing a user to run the following process:

```
Begin: do some calculation;
        WHILE true DO
                Spawn a new process just like me
        END;
```

A) Assuming that a system allowed such a process to run, what would the consequences be:
   a. Assume that SPAWN generated a new process
   b. Assume that SPAWN generated a new thread

B) Suppose that you as an operating system designer have been asked to build in safeguards against such processes. We know (from the Halting Problem of computability theory) that it is impossible, in the general case, to predict the path of execution a program will take. What are the consequences of this basic theoretical result from computer science on your ability to prevent processes like the above from running?

C) Suppose you decide that it is inappropriate to reject certain processes, and that the best approach is to place certain run-time controls on them.
   a. What controls might the operating system use to detect processes like the above at run-time?
   b. Would the controls you propose hinder a process's ability to spawn new processes?
   c. How would the implementation of the controls you propose affect the design of the system's process handling mechanism?

WHAT TO HAND IN

On Web CT in electronic form:

- A single WORD, or PDF document answering the non-programming questions.
- The .c files and make file for the programming question.
- Include a text file describing what operating system your program was developed under. It would be good if you tested your program on our machines on the third floor of Trottier.
- Please provide comments in your code so that it will be easier to grade.

HOW IT WILL BE GRADED

This assignments is worth 30 points, all graded proportionally:

- Question 1: Total of 15 points
  - Proportional based on the specifications
  - Step 1) 2 points
  - Step 2) 3 points
  - Step 3) 3 points
  - Step 4) 7 points
- Question 2: 5 points
  - 2 points for statement of optimality
  - 3 points for justification of optimality
- Question 3: 5 points
  - 3A) 1 point
  - 3B) 2 points
  - 3C) 2 points
- Question 4: 5 points
  - 4A) 1 point
  - 4B) 1 point
  - 4C) 3 points