Assignment #3: The Kernel and Process Execution

Due: March 19, 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 Virtual Memory for the Kernel

2.1 Overview:

For this assignment, you will need the **fully working OS Kernel from Assignment 2**. 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 a memory manager to your kernel.

You will add the following new elements to your OS kernel:

1. The **OS Boot Sequence** to create some necessary OS structures.
2. The **Backing Store,** initialized in the OS Boot Sequence.
3. The **Memory Manager** to handle paging and memory allocation for processes.

In addition, you will modify existing modules in you Kernel:

- PCB Modifications (in pcb.c)
  - Addition of the page table
- Page Fault support (primarily implemented in the memory manager, but modifications are necessary in kernel.c and cpu.c)
  - Find page, swap in, select victim, *we will not do a swap out.*
  - Generate Page Fault and properly assigns addresses
- Prepare RAM for paging (in ram.c)

Fig. 1 below gives a high-level overview of the kernel modules implemented already (shown in black), and the modules we are adding (or heavily modifying) in this assignment (shown in orange). The figure also shows the main interactions between the modules, through the arrows. The arrows highlighted in orange show the functionality that we will add in this assignment. The blue arrows represent functionality that was already implemented in assignments 1 and 2.

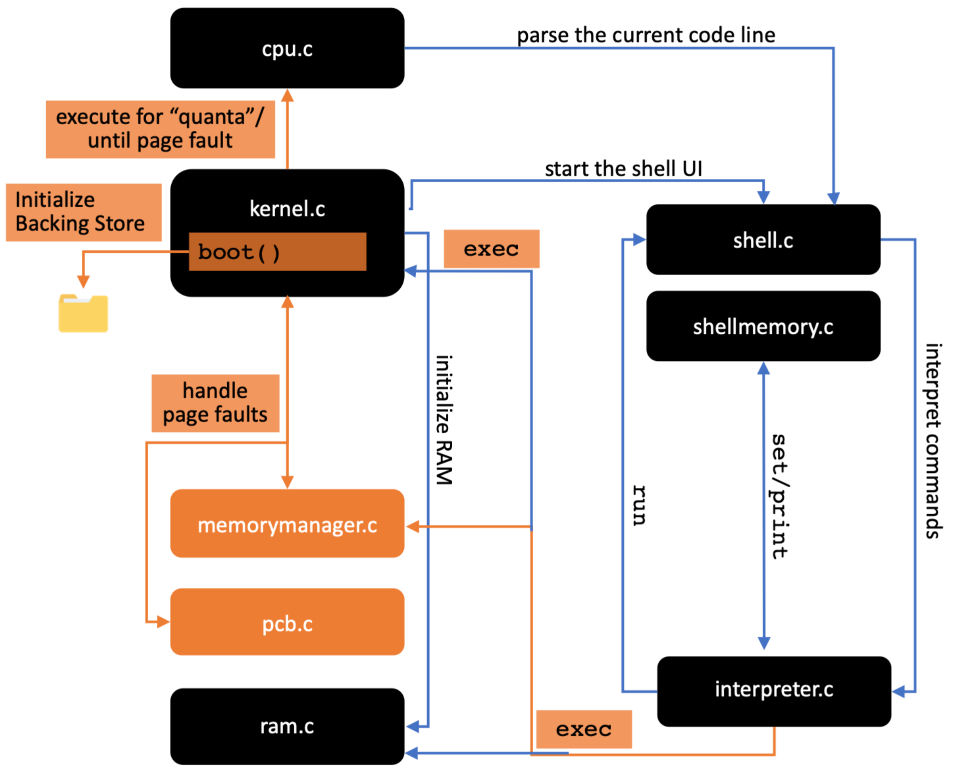

Fig. 1 High-level overview of our kernel. The orange highlights represent the new functionality for Assignment 3.

2.2 The OS boot sequence:

The boot sequence occurs as the very first task begun by the OS. In our simulation, this corresponds to the first thing in your `main()` function. Basically:

```
int main() {
    int error=0;
    boot();             // First : actions performed by boot
    error = kernel(); // Second: actions performed by kernel
    return error;
}
```

Notice that the `main()` function is very simple. The `boot()` function is a one-time call invoked by `main()` at the start to initialize and acquire the resources it needs to run. Place this function in `kernel.c`. Create a new function `kernel()` (in kernel.c) that contains the kernel main function code from Assignment 2.

In our kernel, `boot()` will perform only two activities.

1. It assumes that RAM is a global array of **40** `char*` pointers. This array is not instantiated (not malloced). **It assumes that each 4 cells of the array are a frame** (i.e., the RAM can hold a total of 10 frames). At boot time, there are no other programs running except the kernel, so it initializes every cell of the array to NULL. This indicates that there are no **pages** of code in RAM.
   Important: Note that the structure of the RAM changed from Assignment 2, where it was instantiated as an array of **1000** `char*` pointers.

2. The boot sequence also prepares the **Backing Store.** Typically, the Backing Store is a dedicated partition on the hard disk, but for us it will be simulated by a directory. Preparing the Backing Store means clearing it. Use the C `system()` command to delete the old backing store directory and then create a new directory. Name the directory **`BackingStore`**.
   Important: Note that the directory is only deleted when you run your kernel. This means, **when you exit your kernel the directory will still be present for the TA to look at.**

2.3 The Memory Manager:

The memory manager is the main piece of this assignment. Create a new C module called `memorymanager.c`. You may need to create a .h file. The memory manager is responsible with **launching** the programs that execute via the `exec` command (not the `run` command), **managing frames**, and **paging.**

2.3.1 Launching programs. Create a function called:

`int launcher(FILE *p)` in `memorymanager.c.`

Place the function call within the `exec` function after successfully opening a file.

Important: your `exec` command can now open the same file name multiple times (unlike Assignment 2). The `launcher()` function returns a 1 if it was successful launching the program, otherwise it returns 0. Launching a program consists of the following steps:

1. Copy the entire file into the Backing Store.
2. Close the file pointer pointing to the original file.
3. Open the file in the Backing Store.
4. Our launch paging technique **loads two pages of the program into RAM** when it is first launched. **A page is 4 lines of code.** If the program has 4 or less lines of code, then only one page is loaded. If the program has more than 8 lines of code, then only the first two pages are loaded. To do this, implement the following **helper functions** that exist in the `memorymanager.c` file:
   a. `int countTotalPages(FILE *f)`
   This function returns the total number of pages needed by the program. For example, if the program has L lines or less of code, if $L \leq 4$ the function returns 1. If $4 < L \leq 8$, it returns 2, etc.
   b. `void loadPage(int pageNumber, FILE *f, int frameNumber)`
   `FILE *f` points to the beginning of the file in the Backing Store. The variable `pageNumber` is the desired page from the Backing Store. The function loads the 4 lines of code from the page into the frame in `ram[]`.
   c. `int findFrame()`
   Use the FIFO technique to search `ram[]` for a free frame. If one exists then return its index number, otherwise return -1.
   d. `int findVictim(PCB *p)`
   This function is only invoke when `findFrame()` returns a -1. Use a random number generator to pick a frame number. If the frame number does not belong to the pages of the active PCB (i.e., it is not in its page table) then return that frame number as the victim, otherwise, starting from the

randomly selected frame, iteratively increment the frame number (modulo-wise) until you come to a frame number not belonging to the PCB's pages, and return that number.

e. `int updatePageTable(PCB *p, int pageNumber, int frameNumber, int victimFrame)`
   The page table must also be updated to reflect the changes. We do this once for the PCB asking for the page fault, and we might do it again for the victim PCB (if there was one). If a victim was selected, then the PCB page table of the victim must be updated.
   `p-> pageTable[pageNumber] = frameNumber` (or `= victimFrame`).

2.3.2 Modifying the PCB. The PCB now needs to contain a page table, and auxiliary helper variables to manage the page table. Add the page table array `int pageTable[10]` to the PCB:

- The index of the array is the page number.
- The values stored in the cell is the frame number. Note that the array is size 10 because the RAM has 10 frames in our simulator (4 lines of code per page).
- The PC must be the offset from the beginning of a frame, where offset is the line count starting from zero.
- **Helper variables.** Keep `int PC` as the pointer to the current position in the process.
- Add `int PC_page, PC_offset, pages_max`. This tracks which page and offset the program is currently at, and the total number of pages in this program respectively.

2.4 CPU Modifications:

In addition to switching tasks when the quanta expires, our CPU now needs to handle **page faults**. A page fault occurs when we run out of lines of code in our frame (each frame stores pointers to 4 lines of code). Remember that the CPU does a task switch when the quanta is done. We will leave the quanta to be 2 instructions (like in Assignment 2).

The CPU is modified in the following way: IP stays the same, but we add `int offset`. The new address is `IP+offset`. To access the next instruction, increment the offset to the next address (instruction), `offset++`.

Important: When the offset reaches 4, **generate a pseudo-interrupt**: regardless at what quanta count the program is at, execution stops because the CPU is at the end of the frame, and **the page fault operation must happen.**

The page fault operation follows these steps:

1. Determine the next page: `PC_page++.` (for simplicity, we assume that our scripts never loop.)
2. If `PC_page` is beyond `pages_max` then the program terminates, otherwise we check to see if the frame for that page exists in the `pageTable` array.
3. If `pageTable[PC_page]` is valid then we have the frame number and can do `PC=ram[frame]` and reset `PC_offset` to zero.
4. If `pageTable[PC_page]` is NOT valid then we need to find the page on disk and update the PCB page table (and possibly the victim's page table). Do the following:
   a. Find the page in the Backing Store
   b. Find space in RAM (either find a free cell or select a victim)

c. Update the page table(s).
d. Update the RAM frame with instructions.
e. Do `PC=ram[frame]` and reset `PC_offset` to zero.

5. Since the PCB was interrupted, it has lost it quanta, **even when the quanta was not finished**. Store everything back into the PCB. Place the PCB **at the back of the ready queue.**

2.5 Program termination:

Program termination is like in Assignment 2. This means, **when you exit your kernel the Backing Store directory will still be present for the TA to look at.**

2.6 Testing the kernel:

This assignment uses the same **testfile.txt** and scripts as Assignment 2.

The TAs will use and modify the provided text file to test your kernel. This text file will contain the same tests from Assignment 2. Since we are not implementing threads, **we will not be testing recursive exec calls**. However, note that now it should be possible to launch the same script multiple times in an exec command. For instance, `exec script1.txt script1.txt` is now a valid command, even though it was not the case in Assignment 2. 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. **You can assume that testfile.txt ends with a `quit` command, and hence that `mykernel` terminates when it finishes the testfile commands.**

Make sure your program works in the above way.

WHAT TO HAND IN

Your assignment has a **due date on March 19, 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 #3 submission box in myCourses. You need to submit the following:

- Make sure to ZIP your assignment submission into a single file called ass3.zip
- 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 with the gcc command to compile your program for mimi.**

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.

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.
- **1 point** – Modular programming. The application uses modular programming techniques.
- **1 point** – The student's compiled program must be named `mykernel`.
- **1 point** – A fully working Assignment 2 contained within `mykernel`.
- **1 point** – Ability to use the shell to input the exec command (and see error message).
- **1 point** – The ram[].
- **2 points** – The PCB.
- **1 point** – The CPU.
- **5 points** – Page Fault and Task Switch handling.
- **2 point** – Backing Store.
- **1 point** – Program Termination.
- **1 point** – Boot operation.
- **2 points** – Program can be tested with the testfile.txt file.
- **Programming expectations.** As software students at the 300 and 400 level at McGill University, it is expected that you code at the level of a beginner professional software engineer, even when a programming assignment does not expressly state it. Your code needs to meet the programming style posted on myCourses. **If not, your TA may remove up to 5 points, as they see fit.**

Good Luck!