Assignment #3

Memory Management

Due: March 22, 2019 at 23:55 on myCourses

Labs 3 to 6 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.

Build your assignment #3 using your solution from Assignment #2 or the official solution posted on myCourses.

Building Virtual Memory for the Kernel

The following point lists is a summary of what you will build for this assignment.

Assignment #3 expands on two parts of assignment #2:

- The Computer's Random-Access Memory, which is the following:
  - `FILE *ram[10];`
  - This is an array declared in `kernel.c` as a global variable.
  - For assignment #3 we will consider each cell of the array as a **page frame**
- The multi-processing execution command. Which is the following:
  - `exec prog1 prog2 prog3`
  - Each of the programs will be divided into pages and assigned to a frame in RAM using on-demand page fault requests.

Assignment #3 adds the following new elements that support the above:

- The OS Boot Sequence to create some necessary OS structures
  - Prepare the Backing Store
  - Prepare RAM for paging
- The Memory Manager to handle memory allocation for processes
  - Launcher
    - Loads new programs to the run-time environment
  - PCB Modifications
    - Addition of the page table
  - Page Fault
    - Find page, swap in, select victim, we will not do a swap out
  - Task Switch
    - Generates the Page Fault and properly assigns addresses

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() {
    boot();           // First action performed by kernel
    //Command line    // Second action performed by kernel
    :
    :
}
```

Where `boot()` is a one time call invoked by the kernel at the start to initialize and acquire the resources it needs to run. I do not know how your main function looks like, but the very first line of code (after declaring local variables) is `boot()`. Place this function in `kernel.c`.

The boot operation performs many activities, but for us, it will perform only two activities.

1. It assumes that RAM is a global array. This means it is not instantiated (not malloced). It assumes that each cell of the array is a **frame**. 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.
2. It prepares the Backing Store. This means that it clears the Backing Store. It makes sure there is nothing in the Backing Store. A Backing Store is a partition of the hard disk. For us, this will be simulated by a directory. Use the C `system()` command to delete the old directory and then create a new directory. Name the directory `BackingStore`. 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 it.

THE LAUNCHER & PCB MODIFICATIONS

Create a new C module called `memorymanager.c`. You may need to create a .h file.

The launcher procedure is associated only with your command-line `exec` command (not the `run` command). 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 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 defaults to loading 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. Function: `int countTotalPages(FILE *f)`
   It returns the total number of pages needed by the program. For example if the program has 4 lines or less of code, it returns 1. If greater than 4 and less that or equal to 8, it returns 2. Etc.
b. Function: `FILE *findPage(int pageNumber, FILE *f)`
   `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 returns a pointer that points at the first character of the desired page. A page is 4 lines of code. Since we do not know the length of a line use `fgets()` iteratively `4*pageNumber` times to move the pointer to the correct position. You will need to duplicate your `FILE *f` before using it:
   ```
   FILE *fp2 = fdopen (dup (fileno (f)), "r");
   ```
c. Function: `int findFrame(FILE *page)`
   Use the FIFO technique to search `ram[]` for a frame (not equal to NULL). If one exists then return its index number, otherwise return -1.
d. Function: `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 PCB (page table) then return that frame number, otherwise, starting from the randomly selected frame, iteratively increment the frame number (modulo-wise) until you come to a frame number not belong to the PCB's pages, and return that number.
e. Function:
   `int updateFrame(int frameNumber, int victimFrame, FILE *page)`
   Once we have a pointer to the page and the frame number (or victimFrame) then we can put the page into memory. Since we are not handling dirty pages, then this is easy. If the `frameNumber` is -1 then we use the `victimFrame` as the frame number: Overwrite `ram[frameNumber]` or `ram[victimFrame]` with `FILE *page`.
f. Function:
   ```
   int updatePageTable(PCB *p, int pageNumber,
        int frameNumber, int victimFrame)
   ```
   The PCB's page table must also be updated to reflect the changes. If a victim was selected then the PCB page table of the victim must also be updated. 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).
   `p->pageTable[pageNumber] = frameNumber` (or `= victimFrame`).

5. Modify the PCB by adding an array: `int pageTable[10];` The index of the array is the page number. The values stored in the cell is the frame number. The array is size 10 because RAM is size 10 in our simulator. The PC must be the offset from the beginning of a frame, where offset is the line count starting from zero. Keep `FILE *PC` as the pointer to the current position in the file. 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.

PAGE FAULT & TASK SWITCH & CPU MODIFICATIONS

A page fault occurs when we run out of lines of code in our frame. Each frame stores a pointer to 4 lines of code. When the quanta are done a task-switch occurs. The quanta are 2 instructions (as from assignment 2).

The CPU is modified in the following way: IP stays the same, but we add `int offset`. Theoretically the new address is `IP+offset`, however, IP is a file pointer and will update its position naturally. To complete the simulation, increment the offset to the next address (instruction), `offset++`. When the offset reaches 4 generate an 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++`.
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. Start by (a) finding the page in the backing store, then (b) finding space in RAM (either find a free cell or select a victim), finally (c) update the page tables, (d) RAM pointer, and do (e) `PC=ram[frame]` and (f) reset `PC_offset` to zero.
5. Since the PCB was interrupted, it has lost it quanta, even when there was some quanta left. Store everything back into the PCB (task switch). Place the PCB at the back of the Ready queue.

PROGRAM TERMINATION

Program termination is similar to assignment 2 with one addition: the file in the backing store is deleted.

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 2. 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 #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 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 the executable (compiled on the appropriate machine).
- 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.
    - 1 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 - The student's compiled program must be named mykernel.
    - 1 point – A fully working assignment 3.
    - 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
    - 2 points – Task Switch
    - 3 points – Page Fault
    - 2 point – Backing Store
    - 1 point – Program Termination
    - 1 point – Boot operation
    - 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.