Project Part #3: Memory Management

Due: April 4, 2025 at 23:59

1. Assignment Description

This is the final assignment. After implementing the memory manager, you will have built a simple, simulated OS. So far, our simulated OS supports simple Shell commands and is capable to do process management according to different scheduling techniques. The assumption in the second assignment was that processes fully fit in the Shell memory (like what we saw in the Virtual Memory lecture). In this assignment, we will extend the simulation with **demand paging.**

Similar to Project Part 2, this assignment is larger than Part 1 and MP.
Plan your time wisely and don't hesitate to ask questions **on Discord** if you get stuck.

1.1 Starter files description:

You have two options:

- **[Recommended]** Use your solution to Assignment 2 as starter code for this assignment. If your solution passes the public unit tests it is solid enough to use as a basis for the third assignment.
- Use the official solution to Assignment 2 provided by the OS team as starter code.

1.2 Your tasks:

Your tasks for this assignment are as follows:

- Add scaffolding for paging.
- Design and implement demand paging.
- Implement the LRU replacement policy in demand paging.

On a high level, in this assignment we will allow programs larger than the shell memory size to be run by our OS. We will split the program into pages; only the necessary pages will be loaded into memory and old pages will be switched out when the shell memory gets full. Programs executed through both the `source` and the `exec` commands need to use paging. In addition, we will further relax the assumptions of `exec` by allowing the same program to be executed multiple times by `exec` (remember that in Assignment 2 the programs run by `exec` needed to be different).

For simplicity, in our tests for this assignment we will **only consider the RR (round robin) policy, with a time slice of 2 instructions, with no multithreading and no background execution (the # parameter).** However, the paging mechanism will work with the other scheduling policies as well, if it is implemented correctly. The API for `exec` does not change, so you still need to specifically pass the RR argument.

More details on the *behavior* of your memory manager follow in the rest of this section. Even though we will make some recommendations, you have **full freedom for the implementation**. In particular:

- Unless we explicitly mention how to handle a corner case in the assignment, you are **free to handle corner cases as you wish**, without getting penalized by the TAs.
- You are free to craft your own error messages (please keep it polite).

- Just make sure that your output is the same as the expected output we provide in the test cases in Section 2.
- Formatting issues in the output such as tabs instead of spaces, new lines, etc. **will not be penalized.**

Let's start programming! ☺

1.2.1. Implement the paging infrastructure

We will start by building the basic paging infrastructure. For this intermediate step, you will modify the `source` and `exec` commands to use paging. Note that, even if this step is completed successfully, you will see no difference in output compared to the `source/exec` commands in Assignment 2. However, this step is crucial, as it sets up the scaffolding for demand paging in the following section.

As a reminder from Assignment 1, the `source` API is:

`source SCRIPT` *Executes the commands in the file SCRIPT*

`source` assumes that a file exists with the provided file name, *in the current directory*. It opens that text file and then sends each line one at a time to the interpreter. The interpreter treats each line of text as a command. At the end of the script, the file is closed, and the command line prompt is displayed.

The `exec` API is: `exec prog1 prog2 prog3 POLICY`

- As in Assignment 2, we will not be testing recursive `exec` calls.
- Unlike Assignment 2, `exec` now supports running the same script (i.e., `exec` can take identical arguments so `exec prog1 prog1` is a legal instruction).

You will need to do the following to implement the paging infrastructure.

1. **Set up process sharing.**
   - When exec is passed the same script name twice, we will want to share memory between them. Modify your existing code loading setup from A2 so that two scripts with the same name can share their loaded code. If you loaded code contiguously, you have something like base+bounds virtual memory. Allow multiple processes with the same name to use the same base and bounds.

2. **Partitioning the Shell memory.** The shell memory will be split into two parts:
   - **Frame store.** A part that is reserved to load pages into the shell memory. The number of lines in the frame store should be a multiple of the size of one frame. In this assignment, each page consists of 3 lines of code. Therefore, **each *frame*** in the frame store **has 3 lines**. If you implemented code memory separately from variable memory in A2, then there's probably very little to do here.
     - The frame size of 3 is an unfortunate complication (a power of two would usually make more sense). We have set it up this way so that we can test interesting cases of demand paging without having you implement more scheduling policies. For address translation, you can will probably want to use the / and % operators to separate the page number and offset respectively.
   - **Variable store.** A part that is reserved to store variables.

You are free to implement these two memory parts as you wish. For instance, you can opt to maintain two different data structures, one for loading pages and one for keeping track of variables. Alternatively, you can keep track of everything (pages + variables) in one data structure and keep track of the separation via the OS memory indices (e.g., you can have a convention that the last X lines of the memory are used for tracking variables). We think maintaining separate data structures is easier.

For now, the sizes of the frame store and variable store can be static. We will dynamically set their sizes at compile time in the next section.

3. **Code loading.** The shell will load script(s) into the frame memory as follows.
   - The given script files are used to load program pages into the frame store. If exec receives the same filename more than once, it is only loaded once.
   - At this point, you will load **all** the pages into the frame store for each program. This will change in the next section. Unlike Assignment 2, where you were encouraged to load the scripts contiguously into the Shell memory, in this assignment the pages will not be contiguously loaded. For example, you can load into memory as follows for 2 programs that each have 2 pages (i.e., 6 lines of code per program). Experiment with different orders to check that your paging implementation is working.

Frame store

0	prog2-page0
1	prog2-page0
2	prog2-page0
3	prog1-page0
4	prog1-page0
5	prog1-page0
6	prog2-page1
7	prog2-page1
8	prog2-page1
9	prog1-page1
10	prog1-page1
11	prog1-page1
12	

   - **However, for grading purposes, when a page is loaded into the frame store, it must be placed in the first free spot (i.e., the first available hole).**

4. **Creating the page table.** For each script, a **page table** needs to be added to its PCB to keep track of the loaded pages and their corresponding frames in memory. You are free to implement the page table however you wish. A possible implementation is adding a page table array, where the values stored in each cell of the array represent the frame number in the frame store. For instance, in the example above, the page tables would be:

```
Prog 1:
   pagetable[0] = 1 //frame 1 starts at line 3 in the frame store
   pagetable[1] = 3 //frame 3 starts at line 9 in the frame store
Prog 2:
   pagetable[0] = 0 //frame 0 starts at line 0 in the frame store
```

`pagetable[1] = 2` //frame 2 starts at line 6 in the frame store

You may want to also keep validity information separately, or you could use an unreasonable frame number like -1 to indicate that an entry is invalid.

Note that you will also need to modify the program counter to be able to navigate through the frames correctly. For instance, to execute prog 1, the PC needs to make the transitions between the 2 frames correctly, accessing lines: 3,4,5,9,10,11.

Assumptions (for now):

- The frame store is large enough to hold all the pages for all the programs.
- The variable store has at least 10 entries.
- An exec/run command will not allocate more variables than the size of the variable store.
- Each command (i.e., line) in the scripts will not be larger than a shell memory line (i.e., 100 characters in the reference implementation).
- A one-liner is considered as one command

If everything is correct so far, your `source/exec` commands should have the same behavior as in Assignment 2. You can use the existing unit tests from Assignment 2 to make sure your code works correctly.

1.2.2. Extend the OS Shell with demand paging

We are now ready to add demand paging to our shell. In Section 1.2.1, we assumed the all the pages of all the programs fit in the shell memory. Now, we will get rid of this assumption.

1. **Setting shell memory size at compile time.** First, you need to add two compilation flags to adjust the frame store size and variable store size at compile time as follows.
   - In gcc, you will need to use the -D compilation option, which replaces a macro by a value -D <macro>=<value>.
   - In make, you can pass the value of the variable from the command line.
     Example:
     At the command line: `make xval=42`
     In the Makefile: `gcc -D XVAL=$(xval) -c test.c`
     In test.c: `int x=XVAL;`
   - Using the technique described above, your shell will be compiled by running
     `make mysh framesize=X varmemsize=Y`
     where `X` and `Y` represent the number of lines in the frame store and in the variable store. You can assume that `X` will always be a multiple of 3 in our tests and that `X` will be large enough to hold at least 2 frames for each script in the test. The name of the executable remains `mysh.`

   - Print the following message at shell startup, **instead of** the version message:
     `"Frame Store Size = X; Variable Store Size = Y"`
     Where `X` and `Y` are the values passed to make from the command line.

     - To reiterate: this is printed **instead of** the version message.

Please make sure your program compiles this way and that the memory sizes are adjusted.

2. **Code loading.** Unlike the previous Section, in this section the code pages will be loaded into the shell memory dynamically, as they become necessary.
   - In the beginning of the `source/exec` commands, only **the first two pages** of each program are loaded into the frame store. A page consists of **3 lines of code**. In case the program is smaller than 3 lines of code, only one page is loaded into the frame store. Each page is loaded in the first available hole.
   - The programs start executing, according to the selected scheduling policy (in our case, RR with time slice of 2 lines of code).
3. **Handling page faults.** When a program needs to execute the next line of code that resides in a page which is not yet in memory**, a page fault is triggered.** Upon a page fault:
   - The current process P is interrupted and placed at the back of the ready queue, even if it may still have code lines left in its "time slice". The scheduler selects the next process to run from the ready queue.
     - If you want your A3 solution to work with policies other than RR, you might not want to always place P at the back of the ready queue. Again, such implementation details are up to you!
   - The missing page for process P is brought into the frame store from the file. P's page table needs to be updated accordingly. The new page is loaded into the first free slot in the frame store **if a free slot exists in the frame store.**
   - **If the frame store is full,** we need to pick a victim frame to evict from the frame store. For now, **pick a random frame** in the frame store and evict it. We will adjust this policy in Section 1.2.3. Do not forget to update P's page table. You also need to update any page tables that were using the frame you evicted! To accomplish this, you will need a mapping from frames to the page table(s) that use them. Make sure you keep this bookkeeping structure up-to-date whenever you modify the contents of frames!

     Upon eviction, print the following to the terminal:
     ```
     "Page fault!
     Victim page contents:"
     <the contents of the page, line by line>
     "End of victim page contents."
     ```

     Upon page faults when the frame store is not full, print the following:
     ```
     "Page fault!"
     ```

   - P will resume running whenever it comes next in the ready queue, according to the scheduling policy.
   - When a process terminates**,** you should **not** clean up its corresponding pages in the frame store.

Note that, because the scripting language is very simple the pages can be loaded in order into the shell memory (i.e., for a program, you can assume that you will first load page 1, then pages 2, 3, 4 etc.). This greatly simplifies the implementation, but be aware that real paging systems also account for loops, jumps in the code etc. Also, our pages are always read-only, so you never have *to worry about backtracking in the backing files to overwrite updated data. Obviously, real paging systems have to account for that as well.*

Also note that if the next page is already loaded in memory, there is no page fault. The execution simply continues with the instruction from the next page, if the current process still has remaining lines in its "time slice". This will happen when the same program is used by multiple processes.

If you are especially astute and thinking very hard, you might notice that it's possible for one process to load some pages of a program, and then another process running the same program uses them so much later that they have been evicted. This would require backtracking in the backing file. This is very difficult to implement correctly, since our pages do not have a fixed number of bytes, and so ***the tests will not cause this to happen****. You can simply pretend that it will not happen. If you do choose to try and handle it, beware.*

1.2.3. Adding Page Replacement Policy

The final piece is adjusting the page replacement policy to Least Recently Used (LRU). As seen in class, you will need to keep track of the least recently used frame in the entire frame store and evict it. You must implement **accurate** LRU, not an approximation. Note that, with this policy, a page fault generated by process P1 may still cause the eviction of a page belonging to process P2, so all of the same bookkeeping is necessary.

2. TESTCASES

We provide 5 testcases and expected outputs in the starter code repository. These are the public tests. There **may be** hidden tests for this assignment. We have not decided yet. Please run the testcases to ensure your code runs as expected, and make sure you get similar results in the automatic tests. You are strongly encouraged to add more of your own tests as well. If there are hidden tests, it will be **highly unlikely** that you fail any hidden tests if you are passing all the public ones.

As with A2, you need to run the given test cases from the *A3/test-cases/* directory, and your code for the assignment should remain in the *project/src* directory as with the other project parts.

IMPORTANT: The grading infrastructure uses batch mode, so make sure your program produces the expected outputs when testcases run in batch mode. You can assume that the grading infrastructure will run one test at a time in batch mode, and that there is a fresh recompilation between two testcases.

3. WHAT TO HAND IN

The assignment is **due on April 4, 2025 at 23:59**.

Your final grade will be determined by running the code in the GitLab repository that is crawled by our grading infrastructure. We will take into account the most recent commit that happened before the deadline, taking any requested late days into account, on the main branch of your fork.

The project must compile on the our server by running `make clean; make mysh framesize=X varmemsize=Y`

The project must run in batch mode, i.e. `./mysh < testfile.txt`

Feel free to modify the Makefile to add more structure to your code, but make sure that the project compiles and runs using the commands above.

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.

4. HOW IT WILL BE GRADED

Your program must compile and run on our server to be graded. If the code does not compile/run using the commands in Section 3, in our grading infrastructure you will receive **0 points** for the entire assignment. If you think your code is correct and there is an issue with the grading infrastructure, contact the instructor.

Your assignment is graded out of 20 points. You were provided 5 testcases, with expected outputs. If your code matches the expected output, you will receive 2 points for each testcase. There might be an additional 5 hidden test cases, in which case there will be 2 points per hidden testcase. (If we decide not to have hidden tests, you will receive 4 points for each of the given test cases.) You will receive 0 points for each testcase where your output does not match the expected output. Differences in whitespace in the output, such as tabs instead of spaces, new lines, etc. **will not be penalized.** The TA will look at your source code only if the program runs (correctly or not). The TA looks at your code to verify that you implemented the requirement as requested. Specifically:

- **Hardcoded solutions will receive 0 points for the hardcoded testcase**, even if the output is correct.
- **You must write this assignment in the C Programming language**, otherwise the assignment will receive 0 points.

Congratulations! If you made it this far, you have implemented a simple simulated Operating System ☺